GPGPUs and their programming

Sima, Dezső
Szénási, Sándor
GPGPUs and their programming
irta Sima, Dezső és Szénási, Sándor
Szerzői jog © 2013 Typotex

Kivonat

The widespread use of GPGPUs in an increasing number of HPC (High Performance Computing) areas, such as scientific, engineering, financial and business applications, is one of recent major trends in using informatics.

The lecture slides worked out pursuit two goals. On the one side the lecture aims at presenting the principle of operation, the microarchitecture and main features of GPGPU cores, as well as their implementation as graphics cards or data parallel accelerators. On the other hand, the practical part of the lectures aims at acquainting students with data parallel programming, first of all with GPGPU programming and program optimization using the CUDA language and programming environment.
Tartalom

I. GPGPUs ................................................................................................................................. 1
1. Aim ................................................................................................................................... ix
1.1. Introduction to GPGPUs ................................................................................................. 10
1.2. Representation of objects by triangles ........................................................................ 10
1.3. Main types of shaders in GPUs ..................................................................................... 11
2. Convergence of important features of the vertex and pixel shader models ............... 12
3. Peak FP32/FP64 performance of Nvidia’s GPUs vs Intel’ P4 and Core2 processors [7] 14
4. Peak FP32 performance of AMD’s GPUs [8] ................................................................. 15
6. Evolution of the bandwidth of Nvidia’s GPU’s vs Intel’s P4 and Core2 processors [7] 16
7. Comparing main features of CPUs and GPUs [12] ......................................................... 17
2. The virtual machine concept of GPGPU computing ...................................................... 19
1. Introduction to the virtual machine concept of GPGPU computing ............................ 19
   1.1. The virtual machine concept as used for GPGPUs .................................................... 19
   1.2. Benefits of the portability of the pseudo assembly code .......................................... 19
   1.3. The process of developing applications-1 .................................................................. 19
   1.4. The process of executing applications-2 .................................................................... 20
   1.5. Formulation of application programs at different levels of abstraction ................. 20
2. Introduction to a GPGPU-based massively data parallel computational model .......... 21
   2.1. Specification of GPGPUs at different levels of abstraction ...................................... 21
   2.2. Specification of the GPGPU virtual machine ............................................................ 22
   2.3. The GPGPU-based massively data parallel computational model ....................... 22
   2.4. Emergence of GPGPU-based massively data parallel computational models .......... 22
   2.5. The choice of the computational model presented .................................................. 23
   2.6. The choice of the terminology used .......................................................................... 23
       2.6.1. Designation of the GPGPU-based massively data parallel computational model
             as the SIMT (Single instruction Multiple Threads) model of computation ... 23
       2.6.2. Designation of the terms related to GPGPU-based massively data parallel
             computational model ......................................................................................... 23
3. Introduction to the SIMT computational model ............................................................ 24
3.1. Main components of the SIMT computational models .............................................. 24
3.2. The model of the compute workload .......................................................................... 24
       3.2.1. The thread model ............................................................................................ 25
       3.2.2. The kernel concept .......................................................................................... 29
3.3. The model of the platform .......................................................................................... 31
       3.3.1. The model of computational resources ............................................................ 31
       3.3.2. The memory mode .......................................................................................... 35
3.4. The execution model ................................................................................................. 39
       3.4.1. The concept of SIMT execution ....................................................................... 39
       3.4.2. The concept of assigning work to the PEs ......................................................... 43
       3.4.3. The concept of data sharing .......................................................................... 47
       3.4.4. The concept of data dependent flow control ................................................... 48
       3.4.5. The concept of synchronization ...................................................................... 50
4. The Pseudo ISA of GPGPU computing ......................................................................... 51
4.1. Discussion of pseudo ISAs ........................................................................................ 51
4.2. Case example: The pseudo ISA of Nvidia’s PTX virtual machine ......................... 52
4.3. Case example: The pseudo ISA of Nvidia’s PTX virtual machine ........................... 53
4.4. Nvidia’s compute capability concept [31], [1] ......................................................... 54
4.5. a) Evolution of the functional features, called compute capabilities in subsequent
       versions of Nvidia’s pseudo ISA (PTX) [32] ......................................................... 54
4.6. b) Evolution of the device parameters bound to subsequent compute capability
       versions of Nvidia’s PTX [32] ................................................................................ 55
4.7. c) Architecture specifications bound to compute capability versions of Nvidia’s
       PTX [32] .................................................................................................................. 56
4.8. d) Throughput of native arithmetic instructions in subsequent compute capability versions of Nvidia’s PTX (operations per clock cycle/SM) [7] .................. 56
4.9. PTX ISA Versions generated by subsequent releases of CUDA SDKs and supported compute capability versions (sm[xx]) (designated as Supported Targets in the Table) [20] ................................................................. 57
4.10. Compute capability versions supported by Nvidia’s GPGPU cores and cards [32] 58
4.11. Forward portability of PTX code [31] ........................................ 58
4.12. Compatibility rules of object files (CUBIN files) compiled to a particular GPGPU compute capability version [31] ....................................... 58
3. Overview of GPGPU cores and cards .............................................. 59
1. General overview .................................................................................. 59
1.1. GPGPU cores ...................................................................................... 59
1.2. Main features of Nvidia’s dual-GPU graphics cards ............................. 62
1.3. Main features of AMD’s/ATI’s dual-GPU graphics cards .................. 62
1.4. Remarks on AMD-based graphics cards [36], [37] ............................... 62
2. Overview of the main features of Nvidia’s graphics cores and cards ...... 63
2.1. Main features of Nvidia’s graphics cores and cards ............................ 63
2.1.1. Structure of an SM of the G80 architecture ..................................... 64
2.1.2. Positioning Nvidia’s GPGPU cards in their entire product portfolio [41] 67
2.2. Examples for Nvidia’s graphics cards ................................................. 68
2.2.1. Nvidia GeForce GTX 480 (GF 100 based) [42] ............................... 68
2.2.2. A pair of Nvidia’s GeForce GTX 480 cards [42] (GF100 based) ....... 68
2.2.3. Nvidia’s GeForce GTX 480 and 580 cards [44] .............................. 69
2.2.4. Nvidia’s GTX 680 card [80] (GK104 based) .................................... 70
2.2.5. Nvidia’s GTX Titan card [81] (GTK110 based) ............................... 70
3. Overview of the main features of AMD’s graphics cores and cards ........ 71
3.1. Main features of AMD’s graphics cores and cards ............................. 71
3.2. Examples for AMD’s graphics cards .................................................. 75
3.2.1. ATI HD 5970 (actually RV870 based) [46] .................................... 76
3.2.2. ATI HD 5970 (RV870 based) [47] .................................................. 76
3.2.3. AMD HD 6990 (Cayman based) [48] .......................................... 77
3.2.4. AMD Radeon HD 7970 (Tahiti XT, Southern Islands based) - First GCN implementation [49] .......................................................... 77
3.2.5. Prices of GPGPUs as of Summer 2012 [50] .................................. 78
4. Nvidia’s Fermi family of graphics cores and cards ................................. 80
1. Overview of the Fermi family .................................................................. 80
1.1. Major sub-families of Fermi ................................................................. 81
2. Key innovations of Fermi’s PTX 2.0 ......................................................... 81
2.1. Overview of PTX 2.0 ........................................................................... 81
2.2. Key innovations of PTX 2.0 pseudo ISA ........................................... 82
2.2.1. a) Unified address space for all variables and pointers with a single set of load/store instructions-1 [11] ............................................... 82
2.2.2. a) Unified address space for all variables and pointers with a single set of load/store instructions-2 [11] ............................................... 82
2.2.3. b) 64-bit addressing capability ....................................................... 83
2.2.4. c) New instructions to support OpenCL and DirectCompute APIs 83
2.2.5. d) Full support of predication [51] .................................................. 83
2.2.6. e) Full IEEE 754-3008 support for 32-bit and 64-bit FP precision .. 84
2.2.7. Supporting program development for the Fermi line of GPGPUs [11] 84
3. Major innovations and enhancements of Fermi’s microarchitecture ......... 84
3.1. Major innovations ................................................................................ 84
3.2. Major enhancements .......................................................................... 84
3.3. Major architectural innovations of Fermi ............................................ 85
3.3.1. a) Concurrent kernel execution [52], [18] .................................... 85
3.3.2. b) True two level cache hierarchy [11] ......................................... 85
3.3.3. c) Configurable shared memory/L1 cache per SM [11] ................. 86
3.3.4. d) ECC support [11] ................................................................. 87
3.4. Major architectural enhancements of Fermi ........................................ 87
3.4.1. a) Vastly increased FP64 performance ........................................... 88
3.5. Throughput of arithmetic operations per clock cycle per SM [13] .......... 88
3.5.1. b) Greatly reduced context switching times [11] ............................. 88
3.5.2. c) 10-20 times faster atomic memory operations [11] ..................... 89
4. Microarchitecture of Fermi GF100 ............................................................. 89
4.2. High level microarchitecture of Fermi GT100 .................................. 89
4.3. Evolution of the high level microarchitecture of Nvidia’s GPGPUs [52] .... 90
4.4. Layout of a Cuda GF100 SM [19] ....................................................... 91
4.5. Evolution of the cores (SMs) in Nvidia’s GPGPUs -1 ......................... 91
4.6. Further evolution of the SMs in Nvidia’s GPGPUs -2 ......................... 93
4.7. Structure and operation of the Fermi GF100 GPGPU .......................... 94
4.7.1. Layout of a Cuda GF100 SM [19] ....................................................... 94
4.7.2. A single ALU (“CUDA core”) ....................................................... 95
4.8. Principle of the SIMT execution in case of serial kernel execution .......... 96
4.9. Principle of operation of a Fermi GF100 GPGPU .............................. 96
4.10. Subtasks of work scheduling ............................................................ 96
4.11. Scheduling kernels to SMs [25], [18] ............................................. 97
4.12. Scheduling thread blocks associated with the same kernel to the SMs .... 97
4.13. The notion and main features of thread blocks in CUDA [53] ............... 98
4.15. Scheduling warps for execution in SMs .......................................... 100
4.16. Assumed principle of operation-1 ................................................... 102
4.17. Assumed principle of operation-2 ................................................... 103
4.18. Assumed principle of operation-3 ................................................... 104
4.20. Scheduling policy of warps in an SM of Fermi GF100-1 ....................... 107
4.21. Scheduling policy of warps in an SM of Fermi GT100-2 ....................... 108
4.22. Scheduling policy of warps in an SM of the G80 indicating coarse grain warp scheduling .......................................................... 109
4.23. Scheduling policy of warps in an SM of the G80 indicating fine grain warp scheduling .......................................................... 109
4.24. Estimation of the peak performance of the Fermi GF100 -1 ............... 110
4.25. Estimation of the peak performance of the Fermi GF100 -2 ............... 112
5. Microarchitecture of Fermi GF104 ......................................................... 114
5.1. Available execution resources per SM in the GF104 vs the GF100 ......... 115
5.2. Throughput of arithmetic operations per clock cycle per SM in the GF104/114 [13] 116
5.3. Warp issue in the Fermi GF104 .......................................................... 116
5.4. Peak computational performance data for the Fermi GF104 based GTX 460 card 117
6. Microarchitecture of Fermi GF110 ........................................................... 118
6.1. Overall structure of the Fermi GF110 core [65] ................................ 118
6.2. Block diagrams of the cores (SM units) of the four sub-families of Fermi [66] 119
6.3. Throughput of arithmetic operations per clock cycle per SM in the GF104/114 [13] ........................................................................... 120
7. Microarchitecture of Fermi GF114 .......................................................... 121
7.1. Overall structure of the Fermi GF114 core in the GTX 560 card [67] ....... 121
7.2. Microarchitecture of Fermi GF114 .................................................... 122
7.3. Block diagrams of the cores (SM units) of the four sub-families of Fermi [66] 123
7.4. Throughput of arithmetic operations per clock cycle per SM in the GF104/114 [13] ........................................................................... 123
5. Nvidia’s Kepler family of graphics cores and cards .............................. 125
1. Overview of Nvidia’s Kepler family ....................................................... 125
1.1. Sub-families of Kepler ................................................................. 125
1.2. The GK104 Kepler core [76] ......................................................... 125
1.3. The GK110 Kepler core [70] ......................................................... 126
1.4. Main features of Nvidia’s Kepler GK104 and GK110 cores [70] ........... 126
2. Key innovations of Kepler’s PTX 3.x .................................................... 127
3. Major innovations of Kepler’s microarchitecture ................................. 127
3.1. a) Dynamic parallelism [70] .......................................................... 127
  3.1.1. Principle of dynamic parallelism in Kepler GPUs .................. 128
  3.1.2. Benefits of dynamic parallelism [70] ..................................... 128
3.2. b) The Hyper-Q scheduling mechanism ................................... 128
  3.2.1. Speed up potential of the Hyper-Q scheduling mechanism ...... 133
3.3. c) Grid Management Unit ....................................................... 134
3.4. d) The GPU Direct feature .................................................... 134
3.5. Major enhancements of Kepler’s microarchitecture .................... 135
  3.5.1. a) Setting down the operating frequency of the execution units (shader clock frequency) to the core frequency-1 ........................................... 136
  3.5.2. a) Setting down the operating frequency of the execution units (shader clock frequency) to the core frequency-2 ........................................... 136
  3.5.3. a) Setting down the operating frequency of the execution units (shader clock frequency) to the core frequency-3 ........................................... 136
  3.5.4. a) Setting down the operating frequency of the execution units (shader clock frequency) to the core frequency-4 ........................................... 137
  3.5.5. Comparing the silicon area needed and power consumption of the design approaches used to implement Fermi’s and Kepler’s execution units [76] 137
  3.5.6. b) Simplifying hardware dependency checks by introducing compiler hints ........................................................................................................ 139
  3.5.7. c) Introducing quad warp schedulers per SMX ....................... 139
  3.5.8. Available execution resources in a Kepler SMX vs. a Fermi SM 139
  3.5.9. d) Quadrupling the number of registers that are accessible per thread 140
  3.5.10. Consequence of quadrupling the number of registers that can be accessed per thread in the cores of Kepler ............................................. 141
  3.5.11. e) Introduction of a 48 KB Read-Only data cache for general use 141
  3.5.12. f) Doubled L2 cache size and bandwidth vs. Fermi ............. 142
4. Nvidia’s GK104 Kepler core and related graphics cards ................. 142
  4.1. Sub-families of Kepler ............................................................ 142
  4.2. The GK104 Kepler core [76] .................................................. 143
  4.3. Die photo of Nvidia’s Kepler GK104 core [78] ...................... 143
  4.4. Block diagram of an SMX of the Kepler GK104 core [76] ......... 144
4.5. Peak FP32 and FP34 performance of the GK104 based GeForce GTX 680 graphics card ................................................................. 145
  4.5.1. a) Peak FP32 performance of the GK104 based GeForce GTX 680 graphics card-1 .......................................................... 145
  4.5.2. a) Peak FP32 performance of the GK104 based GeForce GTX 680 graphics card-2 [70] .......................................................... 145
  4.5.3. a) Peak FP32 performance of the GK104 based GeForce GTX 680 graphics card-3 .......................................................... 146
  4.5.4. b) Peak FP64 performance of the GK104 based GeForce GTX 680 graphics card-1 .......................................................... 146
  4.5.5. b) Peak FP64 performance of the GK104 based GeForce GTX 680 graphics card-2 [70] .......................................................... 147
  4.5.6. b) Peak FP64 performance of the GK104 based GeForce GTX 680 graphics card-3 .......................................................... 147
5. Nvidia’s GK110 Kepler core and related graphics cards ................. 147
  5.1. Sub-families of Kepler ............................................................ 147
  5.2. The GK110 core ................................................................. 148
  5.3. The GK110 Kepler core [70] .................................................. 148
    5.3.1. Die photo of Nvidia’s Kepler GK110 core [70] .................. 148
    5.3.2. Die layout of Nvidia’s Kepler GK110 core [79] ............... 149
    5.3.3. Comparison of the die sizes of Nvidia’s GK110 and GK104 Kepler cores [79] .......................................................... 150
    5.3.4. Block diagram of an SMX of the Kepler GK110 core [70] .... 151
    5.3.5. a) Peak FP32 performance of the GK110 based GeForce GTX Titan graphics card .......................................................... 152
    5.3.6. b) Peak FP64 performance of the GK110 based GeForce GTX Titan graphics card .......................................................... 152
6. Evolution of Nvidia’s microarchitectures .................................. 152
6.1. a) Evolution of FP32 warp issue efficiency in Nvidia’s pre-Fermi GPGPUs .................................................. 153
6.2. b) Evolution of FP32 warp issue efficiency in Nvidia’s Fermi and Kepler GPGPUs .................................................. 153
6.3. c) FP64 performance increase in Nvidia’s pre-Fermi GPGPUs .......................... 153
6.4. d) FP64 performance increase in Nvidia’s Tesla and GPGPUs .......................... 154

6. AMD’s Heterogeneous Systems Architecture concept .................................... 155
6.1. Introduction to the HSA concept .......................................................... 155
6.1.1. First public introduction of the HSA concept (6/2011) [84], [85] ..................... 155
6.1.2. Aims of the Heterogeneous Systems Architecture (HSA) concept .................. 156
6.1.2.1. AMD’s view of the evolution of single core, multi-core and application performance (6/2011) [84] .................. 156
6.1.2.2. AMD’s view of the evolution path to heterogeneous computing (6/2011) [84] ............................. 156
6.1.3. Announcement and implementation of AMD’s GCN [90] .......................... 161
6.1.4. AMD’s roadmap for the implementation of HSA [90] .......................... 161
6.1.5. Heterogeneous System Architecture Foundation (HSAF) .......................... 161
6.1.6. AMD’s roadmap for the implementation of HSA - a more detailed view [85] 162

7. AMD’s Southern Islands family of graphics cores and cards .......................... 163
7.1. Overview of AMD’s Southern Island family .................................................. 163
7.1.1. AMD’s roadmap for the implementation of HSA (Financial Analyst Day 2/2012) [90] .......................................................... 163
7.1.2. AMD’s new GPU microarchitecture: the Graphics Core Next (GCN) ............ 163
7.1.3. Announcement and implementation of AMD’s GCN ............................. 164
7.1.4. AMD’s roadmap indicating the introduction of GCN along with the Southern Islands line (2/2012) [94] .................. 164
7.2. Major innovations of the Southern Islands microarchitecture .......................... 164
7.2.1. AMD’s move from the VLIW4-based GPU design to the SIMD-based GPU design .................................................. 164
7.2.1.1. Main steps of the evolution of the microarchitecture of ATI’s (acquired by AMD in 2006) and AMD’s GPU’s [95] .............. 165
7.2.1.2. Principle of using VLIW-based ALUs in GPUs ............................. 165
7.2.1.3. VLIW-based ALUs ................................. 166
7.2.1.4. Example for encoding wavefront queues in form of VLIW4 instructions [96] .......................................................... 166
7.2.1.5. Scheduling the wavefronts given previously while assuming VLIW4 encoding [96] .......................................................... 167
7.2.1.6. Graphics processing and VLIW-based ALUs ............................. 168
7.2.1.7. AMD’s motivation for using VLIW-based ALUs in their previous GPU’s 168
7.2.1.8. Number of execution units (EUs) in Nvidia’s and AMD’s GPUs 168
7.2.1.9. AMD’s motivation for using VLIW5-based ALUs in their first GPU’s [97] 169
7.2.1.10. AMD’s VLIW5 design .......................................................... 169
7.2.1.11. Replacing the original VLIW5 design by the Enhanced VLIW5 design 170
7.2.1.12. Replacing the Enhanced VLIW5 design by the VLIW4 design-1 171
7.2.1.13. Replacing the Enhanced VLIW5 design with the VLIW4 design-2 171
7.2.1.14. Contrasting the main features of AMD’s Enhanced VLIW5 and VLIW4 designs [17] .......................................................... 172
7.2.1.15. Shift to numeric computing and AMD’s move to the GCN architecture 174
7.2.1.16. Replacing the VLIW4-based design by SIMD-based design in AMD’s GCN architecture [91] .......................... 175
7.2.2. AMD ZeroCore technology [49] .......................................................... 175
2.2.1. Achieved reduction of power consumption in long idle periods in AMD’s GPUs due to using the AMD ZeroCore Power Technology [104] .......................... 176
2.2.2. The AMD PowerTune technology ............................................................. 176
2.2.3. The PowerTune technology as introduced in the Northern Island line (HD 6900Cayman, 12/2010) [106] .................................................................... 176
2.2.4. Benefit of using the PowerTune technology [105] ................................. 177

3. AMD’s Southern Islands card family ............................................................. 178

3.1. Overview of the Southern Islands family ................................................. 178
3.1.1. Announcement and implementation of AMD’s GCN ......................... 178
3.1.2. Key features of the Southern Island Family ........................................... 179
3.1.3. Subfamilies of the Southern Island Family-1 [107] ........................... 179
3.1.4. Comparison of the die sizes of AMD’s and Nvidia’s graphics cores [95] 180
3.1.5. Announcement and implementation of AMD’s GCN ......................... 180

3.2. Microarchitecture of the Tahiti XT core used in the Radeon HD7970 card 181
3.2.1. Overall architecture of the Tahiti XT core [95] ................................. 181
3.2.2. Difference between the cores of the 7950 (Tahiti Pro) and the 7970 (Tahiti XT) cards ............................................................... 181
3.2.3. Overall architecture of a Compute Unit (CU) – Simplified block diagram [95] ......................................................................................... 182
3.2.4. Block diagram of the Compute Unit – Detailed block diagram [108] 182
3.2.5. Main units of a CU .................................................................................. 183
3.2.6. The four 16-wide SIMD units of the CU [110] (Strongly simplified discussion) ......................................................................................... 183
3.2.7. The Scalar unit of the CU [110] ............................................................ 184
3.2.8. Peak FP32 and FP64 performance figures of the HD 7970 (Tahiti XT) 184
3.2.9. The cache hierarchy of the HD 7970 (Tahiti XT) [95] ........................ 184
3.2.10. Available data spaces for execution per CU and per GPU in the HD 7970 (Tahiti XT) [110] ................................................................. 185
3.2.11. Memory sizes and peak read bandwidth values in the HD 7970 (Tahiti XT) [16] ......................................................................................... 185

3.3. Principle of operation (repeated from Section 3.2) ................................. 186
3.3.1. Compute workloads .............................................................................. 186
3.3.2. Work items ............................................................................................ 187
3.3.3. The kernel .............................................................................................. 188
3.3.4. Work groups ......................................................................................... 188
3.3.5. Wavefronts-1 ......................................................................................... 189
3.3.6. Wavefronts-2 ......................................................................................... 189
3.3.7. Principle of operation ......................................................................... 190
3.3.8. Launching compute workloads to the GPU ........................................ 190
3.3.9. Launching compute workloads to the GPU ........................................ 190
3.3.10. Execution of compute workloads-1 .................................................. 191
3.3.11. Execution of compute workloads-2 ................................................... 191
3.3.12. Execution of compute workloads-3 ................................................... 191

3.4. Achieved performance figures and power consumption of the HD 7970 (Tahiti XT) ................................................................. 192
3.4.1. a) Gaming performance ...................................................................... 192
3.4.2. a) Gaming performance of high end graphics cards-1 [50] ............... 192
3.4.3. a) Gaming performance of high end graphics cards-2 [50] ............... 192
3.4.4. b) Performance figures of high end graphics cards measured for a compute workload [50] ................................................................. 193
3.4.5. c) Load power consumption (Power consumption at full load) [50] 194
3.4.6. d) Idle power consumption [50] ........................................................... 195

8. AMD’s Sea Islands family ................................................................. 197

1. The Sea Islands family on AMD’s Client and Graphics roadmap [111] ....... 197
2. Key features of the Sea Islands family ...................................................... 197
3. Expected introduction of AMD’s 2. generation GCN, termed as Sea Islands family 198
4. Key parameters of the HD 8950/70 graphics cards .................................. 198
9. Outlook ..................................................................................................... 199
I. Introduction ................................................................. 228
  1. Computational power of GPUs ........................................ 228
  2. Introduction (2) ........................................................ 228
    2.1. Real-world applications ........................................... 228
    2.2. Graphical Processing Units ....................................... 229
      2.2.1. Shaders ...................................................... 229
    2.3. Unified Shader Model ............................................. 229
      2.3.1. Unified Shader ............................................... 230
    2.4. What is GPGPU .................................................... 230
      2.4.1. Programmable graphics cards ................................ 230
    2.5. GPGPU advantages ................................................ 231
      2.5.1. GPU disadvantages ........................................... 231
    2.6. CPU-GPGPU comparison ............................................ 231
    2.7. Memory architecture ............................................. 231
    2.8. SIMT execution .................................................. 233
  2. Programming model .................................................. 234
    1. Basics of CUDA environment ....................................... 234
      1.1. CUDA environment ............................................... 234
        1.1.1. Releases .................................................... 234
        1.1.2. Supported GPUs ............................................ 234
      1.2. Requested components ......................................... 234
        1.2.1. Download CUDA .............................................. 235
II. GPGPUs and their programming ......................................... 222
  1. Introduction (1) ...................................................... 228
  1.1. Computational power of GPUs ....................................... 228
  1.3. The Volta processor with stacked DRAM [113] ....................... 200
  1.6. System layout of the Knights Corner (KCN) DPA [120] .............. 205
  1.7. Microarchitecture of Knights Corner [120] .......................... 205
  1.8. Layout of the ring interconnect on the die [122] ..................... 206
  1.9. Block diagram of a core of the Knights Corner [120] ............... 206
  1.10. Block diagram and pipelined operation of the Vector unit [120] ... 207
  1.11. System architecture of the Xeon Phi coprocessor [122] .......... 207
  1.12. The Xeon Phi 5110P coprocessor [124] ................................ 208
  1.13. The Xeon Phi coprocessor board (backside) [122] .................... 209
  1.15. Principle of Intel’s common software development platform for multicore clusters [125] ........................................ 210
  1.16. The power efficiency of Intel’s Knight Corner vs. competition [120] ... 210
  1.17. Peak performance of the Xeon Phi 5110P and SE10P/X vs. a 2-socket Intel Xeon server [126] ............................................... 211
  1.18. Performance of Xeon Phi 5110P vs a 2-socket Sandy Bridge based Xeon server [126] ............................................... 211
  1.19. Assessing the achieved speed-up of the Xeon Phi 5110P vs a 2-socket Xeon server ..................................................... 212
  1.20. Intel’s roadmap for the Xeon Phi line [127] ............................ 212
  1.21. Assessing the peak FP64 performance of the Knights Corner coprocessor with that of Nvidia’s and AMD’s recent devices ......................... 213
  1.22. d) FP64 performance increase in Nvidia’s Tesla and GPGPUs ........ 213
  1.23. Number of execution units (EUs) in Nvidia’s and AMD’s GPUs .......... 214

10. References ........................................................................ 215

1.1.1. releases  ...................................................................... 234
1.1.2. supported GPUs ....................................................... 234
1.2. requested components .................................................... 234
1.2.1. download cuda ......................................................... 235

GPGPUs and their programming

Created by XMLmind XSL-FO Converter.
1.3. CUDA platform overview ...................................................... 235
  1.3.1. Separate host and device code ........................................ 235
1.4. Parts of the CUDA programming interface ............................ 236
  1.4.1. C language extensions .................................................. 236
  1.4.2. Runtime library .......................................................... 236
1.5. CUDA software stack ....................................................... 236
1.6. Main steps of the CUDA development ................................... 237
  1.6.1. Analyzing of the task ................................................... 237
  1.6.2. Implementing the C/C++ code ........................................ 238
  1.6.3. Compiling and linking ................................................. 238
2. Compiling and linking ....................................................... 238
  2.1. CUDA compilation process details ...................................... 238
    2.1.1. Input .................................................................. 238
    2.1.2. Compilation ............................................................ 239
    2.1.3. Output .................................................................. 239
  2.2. Main parameters of the nvcc compiler (1) ............................. 240
    2.2.1. Usage of the compiler ................................................. 240
    2.2.2. Specifying the compilation phase: ................................ 240
  2.3. Main parameters of the nvcc compiler (2) ............................. 240
    2.3.1. Setting directory information ...................................... 240
  2.4. Main parameters of the nvcc compiler (3) ............................. 241
    2.4.1. Options for steering GPU code generations .................... 241
  2.5. Main parameters of the nvcc compiler (4) ............................. 241
    2.5.1. Miscellaneous options for guiding the compiler driver: ....... 241
  2.6. Compiling example .......................................................... 242
  2.7. Overview of compilation .................................................. 242
3. Platform model ...................................................................... 242
  3.1. CUDA platform model ....................................................... 242
    3.1.1. Asynch execution ....................................................... 243
  3.2. Inside one CUDA device .................................................... 245
    3.2.1. Inside one multiprocessor .............................................. 245
  3.3. Device management ........................................................... 246
    3.3.1. Number of CUDA compatible devices ............................. 246
    3.3.2. Select the active CUDA compatible device ...................... 247
  3.4. Detailed information about devices ...................................... 247
  3.5. Acquire the detailed information about devices ...................... 247
    3.5.1. Exam 2.3.1 ................................................................ 247
4. Memory model ...................................................................... 248
  4.1. The memory concept .......................................................... 248
    4.1.1. Device-host communication .......................................... 248
  4.2. CUDA memory model – global memory ............................... 249
    4.2.1. Declaration ............................................................... 249
  4.3. CUDA memory model – constant memory ............................ 250
    4.3.1. Declaration ............................................................... 250
  4.4. CUDA memory model – texture memory .............................. 251
    4.4.1. Declaration ............................................................... 251
  4.5. CUDA memory model – shared memory ............................... 252
    4.5.1. Declaration ............................................................... 252
  4.6. CUDA memory model - registers ....................................... 253
    4.6.1. Declaration ............................................................... 254
  4.7. CUDA memory model – local memory ................................ 254
    4.7.1. Declaration ............................................................... 255
  4.8. Physical implementation of the CUDA memory model ............ 255
    4.8.1. Dedicated hardware memory ........................................ 255
    4.8.2. Device memory without cache ...................................... 256
    4.8.3. Device memory with cache ......................................... 257
  4.9. Memory handling ............................................................. 257
    4.9.1. Static allocation ....................................................... 258
    4.9.2. Dynamic allocation .................................................... 258
  4.10. CUDA memory regions .................................................... 258
3.5. Device component - functions .................................................. 278
  3.5.1. Fast mathematical functions ............................................. 278
  3.5.2. Synchronization within a block ........................................ 278
3.6. Device component – atomic functions ...................................... 279
  3.6.1. Warp vote functions .......................................................... 279
3.7. Host component - functions .................................................. 279
  3.7.1. Error handling ................................................................. 279
4. Asynchronous Concurrent Execution ............................................. 280
  4.1. Streams .............................................................................. 280
  4.2. Creating/destroying streams .................................................. 280
  4.3. Using streams ..................................................................... 281
  4.4. Using streams example .......................................................... 281
  4.5. Stream synchronization .......................................................... 282
  4.6. Operations implicitly followed a synchronization ....................... 282
  4.7. Stream scheduling [12] ............................................................. 283
  4.8. Concurrency support .............................................................. 283
  4.9. Blocked Queue example ........................................................... 283
  4.10. Blocked Kernel example .......................................................... 284
5. CUDA events ............................................................................ 284
  5.1. Create and destroy a new event ................................................. 284
  5.2. Record an event ................................................................... 285
  5.3. Synchronize an event ............................................................... 286
  5.4. Check an event ...................................................................... 286
  5.5. Synchronization with events ...................................................... 287
  5.6. Synchronization with events (example) ...................................... 287
  5.7. Calculate elapsed time between two events ............................... 288
  5.8. Calculate elapsed time (example) .............................................. 288
6. Unified Virtual Address Space ..................................................... 289
  6.1. CUDA Unified Virtual Address Management ................................ 289
  6.2. Unified Virtual Address Space .................................................. 289
  6.3. Unified Virtual Address Space – check availability ...................... 290
  6.4. Peer to peer communication between devices ............................ 290
  6.5. Using Unified Addressing and P2P transfer ............................... 291
  6.6. Peer-to-peer memory transfer between GPUs ............................ 291
  6.7. Peer-to-peer memory access between GPUs .............................. 292
  6.8. Peer-to-peer memory access kernel .......................................... 293
  6.9. CUDA Unified Virtual Address summary ................................... 293
    6.9.1. More information about UVA ............................................. 293
4. Optimization techniques ............................................................. 295
  1. Using shared memory ............................................................... 295
    1.1. Optimization strategies .......................................................... 295
    1.2. Matrix multiplication .............................................................. 295
    1.2.1. Exam 4.1.1 .................................................................. 295
    1.3. Multi-dimensional matrix in global memory .............................. 296
    1.3.1. Access elements of a multi-dimensional array ........................ 296
    1.4. Multi-dimensional matrix multiplication ................................... 297
    1.5. Multi-dimensional matrix in the GPU memory ......................... 297
    1.6. Aligned arrays ................................................................... 298
    1.7. Access elements in case of aligned storage .............................. 299
    1.8. Aligned memory management ............................................... 299
    1.9. Copy aligned memory ........................................................... 299
    1.10. Matrix multiplication with aligned arrays .............................. 300
    1.10.1. Exam 4.1.2 ................................................................. 300
    1.11. Kernel with aligned arrays ..................................................... 300
    1.12. Invoke kernel with aligned arrays ......................................... 301
    1.13. Using shared memory ........................................................... 301
    1.13.1. Available solutions ........................................................... 302
    1.14. Tiled matrix multiplication ..................................................... 302
    1.15. Matrix multiplication ............................................................ 302
    1.15.1. Exam 4.1.3 ................................................................. 302
1.6. BLAS functions overview ........................................ 304
1.5. CUBLAS memory handling ...................................... 305
1.4. CUBLAS helper functions ........................................ 306
1.3. CUBLAS function result ........................................... 307
1.2. Developing CUBLAS based applications ....................... 308
1.1. CUBLAS Library .................................................... 309
1. CUDA libraries ..................................................... 310
2. Parallel Nsight ...................................................... 311
4.1. Parallel Nsight ..................................................... 312
4. Parallel Nsight ....................................................... 313
3.1. Execution overview ............................................... 314
3. Occupancy considerations .......................................... 315
3.10. CUDA Occupancy calculator – impact of varying shared memory ................................................. 316
3.9. CUDA Occupancy calculator – impact of varying register count ........................................ 317
3.8. CUDA Occupancy calculator – impact of varying block size ........................................... 318
3.7. CUDA Occupancy calculator - example ......................... 319
3.6. CUDA Occupancy calculator ...................................... 320
3.5. Occupancy and block size ......................................... 321
3.4. Occupancy and shared memory .................................. 322
3.3. Occupancy and registers .......................................... 323
3.2. Occupancy .......................................................... 324
3.1. Execution overview ............................................... 325
2.8. Parallelization inside the block ................................... 326
2.7. Find the minimal value of a vector – using shared memory ........................................ 327
2.6. Find the minimal value of a vector – using global memory ........................................ 328
2.5. CUDA atomic bitwise instructions ............................... 329
2.4. CUDA atomic instructions (2) ................................... 330
2.3. CUDA atomic instructions ......................................... 331
2.2. CUDA atomic instructions ......................................... 332
2.1. Atomic operations .................................................. 333
2. CUDA atomic instructions ............................................ 334
1.2. Comparing runtime of original and tiled algorithms .......... 335
1.1. Optimized matrix multiplication .................................. 336
1.1. Optimized matrix multiplication (2) ............................ 337
1.1. Optimized matrix multiplication (3) ............................ 338
1.1. Optimized matrix multiplication (4) ............................ 339
1.1. Optimized matrix multiplication (5) ............................ 340
1.1. Optimized matrix multiplication source code ................. 341
1.2. Comparing of CPU and GPU implementation .................. 342
1.3. Occupancy and registers .......................................... 343
1.4. Occupancy and shared memory .................................. 344
1.3. Occupancy considerations ........................................ 345
1.2. Occupancy .......................................................... 346
1.1. Execution overview ............................................... 347
5. CUDA libraries ..................................................... 348
4.9. Using the memory checker ........................................ 349
4.8. Debugging PTX code .............................................. 350
4.7. CUDA Device Summary - warp ................................ 351
4.6. CUDA Device Summary - grid ................................... 352
4.5. CUDA Device Summary ............................................ 353
4.4. CUDA Debug Focus ................................................ 354
4.3. Watch GPU memory regions ...................................... 355
4.2. Kernel debugging ................................................... 356
4.1. Parallel Nsight ..................................................... 357
4. CUDA atomic instructions ............................................ 358
3.10. CUDA Occupancy calculator – impact of varying shared memory ........................................ 359
3.9. CUDA Occupancy calculator – impact of varying register count ........................................... 360
3.8. CUDA Occupancy calculator – impact of varying block size ........................................... 361
3.7. CUDA Occupancy calculator - example ......................... 362
3.6. CUDA Occupancy calculator ...................................... 363
3.5. Occupancy and block size ......................................... 364
3.4. Occupancy and shared memory .................................. 365
3.3. Occupancy and registers .......................................... 366
3.2. Occupancy .......................................................... 367
3.1. Execution overview ............................................... 368
2.8. Parallelization inside the block ................................... 369
2.7. Find the minimal value of a vector – using shared memory ........................................ 370
2.6. Find the minimal value of a vector – using global memory ........................................ 371
2.5. CUDA atomic bitwise instructions ............................... 372
2.4. CUDA atomic instructions (2) ................................... 373
2.3. CUDA atomic instructions ......................................... 374
2.2. CUDA atomic instructions ......................................... 375
2.1. Atomic operations .................................................. 376
2. CUDA atomic instructions ............................................ 377
1. CUDA libraries ..................................................... 378
1.16. Optimized matrix multiplication .................................. 379
1.15. Optimized matrix multiplication .................................. 380
1.14. Comparing of CPU and GPU implementation .................. 381
1.13. Comparing runtime of atomic and parallel version ............ 382
1.12. Parallel minimum search ........................................ 383
1.11. Parallel minimum search - kernel ................................ 384
1.10. Parallel minimum search .......................................... 385
1.9. Parallel minimum search ........................................... 386
1.8. Parallelization inside the block ................................... 387
1.7. Find the minimal value of a vector – using shared memory ........................................ 388
1.6. BLAS functions overview ........................................ 389
1.5. CUBLAS memory handling ...................................... 390
1.4. CUBLAS helper functions ........................................ 391
1.3. CUBLAS function result .......................................... 392
1.2. Developing CUBLAS based applications ....................... 393
1.1. CUBLAS Library .................................................... 394
1. CUDA libraries ..................................................... 395

1.7. Some CUBLAS level 1 functions .............................................. 330
1.8. Some CUBLAS level 2 functions .............................................. 331
1.9. Some CUBLAS level 3 functions .............................................. 332
6. CUDA versions ............................................................................ 333
  1. CUDA 4 features ........................................................................ 333
    1.1. CUDA 4.0 features ................................................................. 333
       1.1.1. Share GPUs across multiple threads ................................. 333
       1.1.2. One thread can access all GPUs .................................... 333
    1.2. Set current device ................................................................. 333
    1.3. Current device - streams, events .......................................... 333
    1.4. Multi-GPU example ............................................................... 334
    1.5. Using multiple CPU threads ................................................. 334
    1.6. Vector multiplication with multiple GPUs - kernel .................. 335
    1.7. Vector multiplication with multiple GPUs – memory allocation ... 335
    1.8. Vector multiplication with multiple GPUs – kernel invocation .... 336
    1.9. Vector multiplication with multiple GPUs – kernel invocation .... 336
  2. CUDA 5 features ........................................................................ 337
    2.1. CUDA 5.0 features [26] ....................................................... 337
       2.1.1. Dynamic Parallelism ....................................................... 337
       2.1.2. GPU-Callable Libraries ............................................... 337
       2.1.3. GPUDirect Support for RDMA ...................................... 337
    2.2. Dynamic parallelism ............................................................... 337
       2.2.1. Dynamic parallelism example ...................................... 337
  7. References ................................................................................... 339
  1. References .................................................................................. 339
Az ábrák listája

1.1. Contrasting the utilization of the silicon area in CPUs and GPUs [10] ........................................ 17
1.13. .............................................................................................................................................. 17
2.1. .................................................................................................................................................. 19
2.2. .................................................................................................................................................. 20
2.3. .................................................................................................................................................. 20
2.4. .................................................................................................................................................. 20
2.5. .................................................................................................................................................. 21
2.6. Terminologies used with GPGPUs/Data parallel accelerators ...................................................... 23
2.7. .................................................................................................................................................. 24
2.8. .................................................................................................................................................. 24
2.9. .................................................................................................................................................. 25
2.10. The interpretation of N-Dimensional Ranges [12] ...................................................................... 25
2.11. The interpretation of a work-item (Based on [12]) .................................................................... 26
2.12. The interpretation of wavefronts [12] ...................................................................................... 27
2.13. The interpretation wavefronts [12] ......................................................................................... 28
2.14. .................................................................................................................................................. 29
2.15. Interpretation of the kernel concept .......................................................................................... 29
2.16. .................................................................................................................................................. 30
2.17. .................................................................................................................................................. 31
2.18. .................................................................................................................................................. 31
2.19. The model of computational resources [15] ............................................................................. 32
2.20. .................................................................................................................................................. 32
2.21. .................................................................................................................................................. 33
2.22. .................................................................................................................................................. 33
2.23. .................................................................................................................................................. 34
2.24. .................................................................................................................................................. 35
2.25. Overview of available memory spaces in GPGPUs ................................................................. 35
2.26. .................................................................................................................................................. 35
2.27. .................................................................................................................................................. 36
2.28. .................................................................................................................................................. 36
2.29. .................................................................................................................................................. 37
2.30. .................................................................................................................................................. 37
2.31. Available data spaces in AMD’s IL vers. 2.0 [22] ................................................................. 38
2.32. .................................................................................................................................................. 38
2.33. Key abstractions of the execution model ................................................................................. 39
2.34. Key abstractions of the execution model ................................................................................. 39
2.35. Scheduling wavefronts for hiding stalls due to device memory reads [24] ......................... 40
2.36. .................................................................................................................................................. 40
2.37. The assumed platform model [15] .......................................................................................... 41
2.38. Principle of operation of SIMD execution .............................................................................. 41
2.39. Principle of the SIMT execution .............................................................................................. 42
2.40. Principle of the SIMT execution .............................................................................................. 42
2.41. Contrasting key features of SIMD and SIMT execution ......................................................... 43
2.42. Key abstractions of the execution model ............................................................................... 43
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1. A single ALU [57]</td>
<td>95</td>
</tr>
<tr>
<td>4.2. Measuring the performance of the single ALU core [57]</td>
<td>95</td>
</tr>
<tr>
<td>4.3. Hierarchy of threads [58]</td>
<td>106</td>
</tr>
<tr>
<td>4.4. Warp scheduling in the G80 [59]</td>
<td>107</td>
</tr>
<tr>
<td>4.5. Warp scheduling in the G80 [59]</td>
<td>107</td>
</tr>
<tr>
<td>4.6. Warp scheduling in the G80 [59]</td>
<td>107</td>
</tr>
<tr>
<td>4.7. Warp scheduling in the G80 [59]</td>
<td>107</td>
</tr>
<tr>
<td>4.8. Warp scheduling in the G80 [59]</td>
<td>107</td>
</tr>
<tr>
<td>4.9. Warp scheduling in the G80 [59]</td>
<td>107</td>
</tr>
<tr>
<td>4.10. Warp scheduling in the G80 [59]</td>
<td>107</td>
</tr>
<tr>
<td>4.11. Warp scheduling in the G80 [59]</td>
<td>107</td>
</tr>
<tr>
<td>4.13. Fermi’s system architecture [52]</td>
<td>89</td>
</tr>
<tr>
<td>4.14. Fermi’s system architecture [52]</td>
<td>90</td>
</tr>
<tr>
<td>4.15. Fermi’s system architecture [52]</td>
<td>91</td>
</tr>
<tr>
<td>4.16. Fermi’s system architecture [52]</td>
<td>92</td>
</tr>
<tr>
<td>4.17. Fermi’s system architecture [52]</td>
<td>92</td>
</tr>
<tr>
<td>4.18. Fermi’s system architecture [52]</td>
<td>93</td>
</tr>
<tr>
<td>4.19. Fermi’s system architecture [52]</td>
<td>94</td>
</tr>
<tr>
<td>4.20. Fermi’s system architecture [52]</td>
<td>95</td>
</tr>
<tr>
<td>4.21. A single ALU [57]</td>
<td>95</td>
</tr>
<tr>
<td>4.22. Contrasting the Multiply-Add (MAD) and the Fused-Multiply-Add (FMA) operations [51]</td>
<td>96</td>
</tr>
<tr>
<td>4.23. Hierarchy of threads [58]</td>
<td>106</td>
</tr>
<tr>
<td>4.24. Warp scheduling in the G80 [59]</td>
<td>107</td>
</tr>
<tr>
<td>4.25. Warp scheduling in the G80 [59]</td>
<td>107</td>
</tr>
<tr>
<td>4.27. Warp scheduling in the G80 [59]</td>
<td>107</td>
</tr>
<tr>
<td>4.28. Warp scheduling in the G80 [59]</td>
<td>107</td>
</tr>
<tr>
<td>4.29. The Bulldozer core [60]</td>
<td>102</td>
</tr>
<tr>
<td>4.30. The Fermi core [52]</td>
<td>102</td>
</tr>
<tr>
<td>4.31. The Fermi core [52]</td>
<td>104</td>
</tr>
<tr>
<td>4.32. The Fermi core [52]</td>
<td>105</td>
</tr>
<tr>
<td>4.33. The Fermi core [52]</td>
<td>106</td>
</tr>
<tr>
<td>4.34. The Fermi core [52]</td>
<td>107</td>
</tr>
<tr>
<td>4.35. Warp scheduling in the G80 [59]</td>
<td>108</td>
</tr>
<tr>
<td>4.36. Warp scheduling in the G80 [59]</td>
<td>109</td>
</tr>
<tr>
<td>4.37. Warp scheduling in the G80 [59]</td>
<td>110</td>
</tr>
<tr>
<td>4.38. The Fermi core [52]</td>
<td>111</td>
</tr>
<tr>
<td>4.39. The Fermi core [52]</td>
<td>113</td>
</tr>
<tr>
<td>4.40. Contrasting the overall structures of the GF104 and the GF100 [62]</td>
<td>114</td>
</tr>
<tr>
<td>4.41. The Fermi core [52]</td>
<td>115</td>
</tr>
<tr>
<td>4.42. The Fermi core [52]</td>
<td>116</td>
</tr>
<tr>
<td>4.43. The Fermi core [52]</td>
<td>116</td>
</tr>
<tr>
<td>4.44. The Fermi core [52]</td>
<td>118</td>
</tr>
<tr>
<td>4.45. The Fermi core [52]</td>
<td>119</td>
</tr>
<tr>
<td>4.46. The Fermi core [52]</td>
<td>119</td>
</tr>
<tr>
<td>4.47. The Fermi core [52]</td>
<td>120</td>
</tr>
<tr>
<td>4.48. The Fermi core [52]</td>
<td>121</td>
</tr>
<tr>
<td>4.49. The Fermi core [52]</td>
<td>122</td>
</tr>
<tr>
<td>4.50. The Fermi core [52]</td>
<td>123</td>
</tr>
<tr>
<td>4.51. The Fermi core [52]</td>
<td>123</td>
</tr>
<tr>
<td>5.1. Overview of GPGPU cores and cards and their basic software support (3)</td>
<td>125</td>
</tr>
<tr>
<td>5.2. Overview of GPGPU cores and cards and their basic software support (3)</td>
<td>125</td>
</tr>
<tr>
<td>5.3. Block diagram of the GK104 Kepler core</td>
<td>125</td>
</tr>
<tr>
<td>5.4. Block diagram of the GK110 Kepler core</td>
<td>126</td>
</tr>
<tr>
<td>5.5. Block diagram of the GK110 Kepler core</td>
<td>126</td>
</tr>
<tr>
<td>5.6. Principle of dynamic parallelism in Kepler GPUs</td>
<td>128</td>
</tr>
<tr>
<td>5.7. Serial and Concurrent Kernel execution models of Fermi and pre-Fermi devices [25], [18]</td>
<td>128</td>
</tr>
<tr>
<td>5.8. Serial and Concurrent Kernel execution models of Fermi and pre-Fermi devices [25], [18]</td>
<td>129</td>
</tr>
<tr>
<td>5.9. Using a single hardware work queue to schedule multiple streams of kernels [70]</td>
<td>129</td>
</tr>
<tr>
<td>5.10. Fermi’s model of concurrent execution [70]</td>
<td>130</td>
</tr>
<tr>
<td>5.11. Principle of the Hyper-Q scheduling model [70]</td>
<td>131</td>
</tr>
<tr>
<td>5.12. Principle of Hyper-Q [70]</td>
<td>132</td>
</tr>
</tbody>
</table>
5.13. Kepler’s Hyper-Q scheduling mechanism used to issue simultaneous MPI tasks compared to the MPI task issue mechanism of Fermi [73] ................................................................. 133
5.14. Contrasting the scheduling subsysystems of the Fermi and Kepler architectures [70] ...... 134
5.15. The GPU Direct feature of Kepler [70] ........................................................................ 134
5.16. View of the Titan (Cray XK7) supercomputer [75] ....................................................... 135
5.17. Core clock frequency vs shader frequency in Nvidia’s major GPU families .................. 136
5.18. Available execution resources in a Kepler SMX vs. a Fermi SM .................................. 137
5.19. Core clock frequency vs shader frequency in Nvidia’s major GPU families ................. 137
5.20. ....................................................................................................................................... 137
5.21. Main features of Nvidia’s and AMD’s GPU cards [77] ................................................. 138
5.22. Comparing the power efficiency of the Fermi and Kepler designs [76] ......................... 138
5.23. Block diagram of Fermi’s hardware dependency checking [76] ................................. 139
5.24. Block diagram of Kepler’s hardware dependency checking [76] .............................. 139
5.25. ....................................................................................................................................... 139
5.26. ....................................................................................................................................... 139
5.27. ....................................................................................................................................... 140
5.28. Key device features bound to the compute capability versions [32] ............................ 140
5.29. Kepler’s cache architecture [70] ............................................................................... 141
5.30. ....................................................................................................................................... 142
5.31. Block diagram of the GK104 Kepler core ................................................................. 143
5.32. ....................................................................................................................................... 143
5.33. ....................................................................................................................................... 144
5.34. Main features of Nvidia’s Kepler-based GTX 680 GPGPU card [33] ...................... 145
5.35. ....................................................................................................................................... 145
5.36. ....................................................................................................................................... 147
5.37. ....................................................................................................................................... 147
5.38. Block diagram of the GK110 Kepler core ................................................................. 148
5.39. ....................................................................................................................................... 149
5.40. ....................................................................................................................................... 149
5.41. ....................................................................................................................................... 150
5.42. ....................................................................................................................................... 151
5.43. ....................................................................................................................................... 153
5.44. ....................................................................................................................................... 153
5.45. ....................................................................................................................................... 154
5.46. ....................................................................................................................................... 154
6.1. ........................................................................................................................................ 155
6.2. ........................................................................................................................................ 156
6.3. AMD’s view of the evolution to heterogeneous computing [84] ................................. 156
6.4. ........................................................................................................................................ 158
6.5. The compilation phase ................................................................................................... 158
6.6. The runtime phase ......................................................................................................... 159
6.7. Classical retargetable tree-phase compiler design [89] .............................................. 160
6.8. ........................................................................................................................................ 161
6.9. ........................................................................................................................................ 161
6.10. ...................................................................................................................................... 162
7.1. ........................................................................................................................................ 163
7.2. Overview of GPGPU cores and cards and their basic software support (3) ............ 164
7.3. ........................................................................................................................................ 164
7.4. Evolution of the microarchitecture of ATI’s (acquired by AMD in 2006) and AMD’s GPUs [95] 165
7.5. Block diagram of a VLIW5 ALU [17] ......................................................................... 165
7.6. ........................................................................................................................................ 166
7.7. Block diagram of a VLIW4 ALU (introduced in the Northern Island line (HD6900 line)) [17] 166
7.8. Simplified block diagram of a 16-wide SIMD unit (based on [95]) ............................. 167
7.9. ........................................................................................................................................ 167
7.10. ..................................................................................................................................... 168
7.11. Main steps of the evolution of AMD’s VLIW-based shader ALUs ......................... 169
7.12. The vertex shader pipeline in ATI’s first GPU supporting DX 9.0 (The R300 core) (2002) [102] 170
7.13. Main steps of the evolution of AMD’s VLIW-based shader ALUs ......................... 170
### GPGPUs and their programming

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2</td>
<td>228</td>
</tr>
<tr>
<td>1.3. [3]</td>
<td>230</td>
</tr>
<tr>
<td>1.4. [5]</td>
<td>231</td>
</tr>
<tr>
<td>1.5. [3]</td>
<td>232</td>
</tr>
<tr>
<td>1.6. [7]</td>
<td>233</td>
</tr>
<tr>
<td>2.1</td>
<td>234</td>
</tr>
<tr>
<td>2.2. [5]</td>
<td>235</td>
</tr>
<tr>
<td>2.3. [5]</td>
<td>237</td>
</tr>
<tr>
<td>2.4. [2]</td>
<td>238</td>
</tr>
<tr>
<td>2.5</td>
<td>242</td>
</tr>
<tr>
<td>2.6</td>
<td>242</td>
</tr>
<tr>
<td>2.7. [5]</td>
<td>243</td>
</tr>
<tr>
<td>2.8. [5]</td>
<td>245</td>
</tr>
<tr>
<td>2.9</td>
<td>246</td>
</tr>
<tr>
<td>2.10</td>
<td>247</td>
</tr>
<tr>
<td>2.11</td>
<td>247</td>
</tr>
<tr>
<td>2.12</td>
<td>248</td>
</tr>
<tr>
<td>2.14</td>
<td>249</td>
</tr>
<tr>
<td>2.15. [5]</td>
<td>249</td>
</tr>
<tr>
<td>2.16</td>
<td>250</td>
</tr>
<tr>
<td>2.17. [5]</td>
<td>250</td>
</tr>
<tr>
<td>2.18. [5]</td>
<td>251</td>
</tr>
<tr>
<td>2.19</td>
<td>252</td>
</tr>
<tr>
<td>2.20</td>
<td>253</td>
</tr>
<tr>
<td>2.22</td>
<td>254</td>
</tr>
<tr>
<td>2.23. [5]</td>
<td>254</td>
</tr>
<tr>
<td>2.24</td>
<td>255</td>
</tr>
<tr>
<td>2.25. [5]</td>
<td>255</td>
</tr>
<tr>
<td>2.27</td>
<td>258</td>
</tr>
<tr>
<td>2.28</td>
<td>258</td>
</tr>
<tr>
<td>2.29</td>
<td>259</td>
</tr>
<tr>
<td>2.30</td>
<td>259</td>
</tr>
<tr>
<td>2.31</td>
<td>259</td>
</tr>
<tr>
<td>2.32. [5]</td>
<td>260</td>
</tr>
<tr>
<td>2.33</td>
<td>261</td>
</tr>
<tr>
<td>2.34. [5]</td>
<td>263</td>
</tr>
<tr>
<td>2.35</td>
<td>266</td>
</tr>
<tr>
<td>2.36</td>
<td>267</td>
</tr>
<tr>
<td>2.37</td>
<td>267</td>
</tr>
<tr>
<td>2.38</td>
<td>268</td>
</tr>
<tr>
<td>2.39</td>
<td>268</td>
</tr>
<tr>
<td>3.1</td>
<td>270</td>
</tr>
<tr>
<td>3.2</td>
<td>271</td>
</tr>
<tr>
<td>3.3</td>
<td>272</td>
</tr>
<tr>
<td>3.4</td>
<td>272</td>
</tr>
<tr>
<td>3.5</td>
<td>275</td>
</tr>
<tr>
<td>3.6</td>
<td>276</td>
</tr>
<tr>
<td>3.7. [12]</td>
<td>280</td>
</tr>
<tr>
<td>3.8</td>
<td>280</td>
</tr>
<tr>
<td>3.9</td>
<td>280</td>
</tr>
<tr>
<td>3.10</td>
<td>281</td>
</tr>
<tr>
<td>3.11</td>
<td>281</td>
</tr>
<tr>
<td>3.12</td>
<td>282</td>
</tr>
<tr>
<td>3.13</td>
<td>282</td>
</tr>
<tr>
<td>3.15. [12]</td>
<td>284</td>
</tr>
</tbody>
</table>
4.41. .................................................................................................................. 327
5.1. .................................................................................................................. 329
6.1. .................................................................................................................. 334
6.2. .................................................................................................................. 335
6.3. .................................................................................................................. 335
6.4. .................................................................................................................. 336
6.5. .................................................................................................................. 337
6.6. .................................................................................................................. 337
I. rész - GPGPUs
4.9. PTX ISA Versions generated by subsequent releases of CUDA SDKs and supported compute capability versions (sm[xx]) (designated as Supported Targets in the Table) [20] 57

4.10. Compute capability versions supported by Nvidia’s GPGPU cores and cards [32] 58

4.11. Forward portability of PTX code [31] ........................................................................ 58

4.12. Compatibility rules of object files (CUBIN files) compiled to a particular GPGPU compute capability version [31] ........................................................................ 58

3. Overview of GPGPU cores and cards ........................................................................... 59

1. General overview ........................................................................................................... 59

1.1. GPGPU cores ........................................................................................................... 59

1.2. Main features of Nvidia’s dual-GPU graphics cards ................................................. 62

1.3. Main features of AMD’s/ATI’s dual-GPU graphics cards ....................................... 62

1.4. Remarks on AMD-based graphics cards [36], [37] .................................................. 62

2. Overview of the main features of Nvidia’s graphics cores and cards ......................... 63

2.1. Main features of Nvidia’s graphics cores and cards .................................................. 63

2.1.1. Structure of an SM of the G80 architecture .......................................................... 64

2.1.2. Positioning Nvidia’s GPGPU cards in their entire product portfolio [41] .......... 67

2.2. Examples for Nvidia’s graphics cards ....................................................................... 68

2.2.1. Nvidia GeForce GTX 480 (GF 100 based) [42] ................................................. 68

2.2.2. A pair of Nvidia’s GeForce GTX 480 cards [42] (GF100 based) ......................... 68

2.2.3. Nvidia’s GeForce GTX 480 and 580 cards [44] .................................................. 69

2.2.4. Nvidia’s GTX 680 card [80] (GK104 based) ......................................................... 70

2.2.5. Nvidia’s GTX Titan card [81] (GK110 based) ......................................................... 70

3. Overview of the main features of AMD’s graphics cores and cards ......................... 71

3.1. Main features of AMD’s graphics cores and cards ................................................... 71

3.2. Examples for AMD’s graphics cards ........................................................................ 75

3.2.1. ATI HD 5970 (actually RV870 based) [46] .......................................................... 76

3.2.2. ATI HD 5970 (RV870 based) [47] ....................................................................... 76

3.2.3. AMD HD 6990 (Cayman based) [48] ................................................................... 77

3.2.4. AMD Radeon HD 7970 (Tahiti XT, Southern Islands based) - First GCN implementation [49] ................................................................. 77

3.2.5. Prices of GPGPUs as of Summer 2012 [50] ......................................................... 78

4. Nvidia’s Fermi family of graphics cores and cards ......................................................... 80

1. Overview of the Fermi family ....................................................................................... 80

1.1. Major sub-families of Fermi ..................................................................................... 81

2. Key innovations of Fermi’s PTX 2.0 ........................................................................... 81

2.1. Overview of PTX 2.0 .............................................................................................. 81

2.2. Key innovations of PTX 2.0 pseudo ISA ............................................................... 82

2.2.1. a) Unified address space for all variables and pointers with a single set of load/store instructions-1 [11] ................................................................. 82

2.2.2. a) Unified address space for all variables and pointers with a single set of load/store instructions-2 [11] ................................................................. 82

2.2.3. b) 64-bit addressing capability .......................................................................... 83

2.2.4. c) New instructions to support OpenCL and DirectCompute APIs .................. 83

2.2.5. d) Full support of predication [51] ....................................................................... 83

2.2.6. e) Full IEEE 754-3008 support for 32-bit and 64-bit FP precision .................... 84

2.2.7. Supporting program development for the Fermi line of GPGPUs [11] .............. 84

3. Major innovations and enhancements of Fermi’s microarchitecture ......................... 84

3.1. Major innovations .................................................................................................... 84

3.2. Major enhancements ............................................................................................... 84

3.3. Major architectural innovations of Fermi ............................................................... 85

3.3.1. a) Concurrent kernel execution [52], [18] .............................................................. 85

3.3.2. b) True two level cache hierarchy [11] ................................................................. 85

3.3.3. c) Configurable shared memory/L1 cache per SM [11] ......................................... 86

3.3.4. d) ECC support [11] ......................................................................................... 87

3.4. Major architectural enhancements of Fermi ............................................................. 87

3.4.1. a) Vastly increased FP64 performance ............................................................... 88

3.5. Throughput of arithmetic operations per clock cycle per SM [13] ......................... 88

3.5.1. b) Greatly reduced context switching times [11] .................................................. 88

3.5.2. c) 10-20 times faster atomic memory operations [11] ......................................... 89
4. Microarchitecture of Fermi GF100 ................................. 89
   4.2. High level microarchitecture of Fermi GT100 .......... 89
   4.3. Evolution of the high level microarchitecture of Nvidia’s GPGPUs [52] .......................... 90
   4.4. Layout of a Cuda GF100 SM [19] .......................... 91
   4.5. Evolution of the cores (SMs) in Nvidia’s GPGPUs -1 ............................................. 91
   4.6. Further evolution of the SMs in Nvidia’s GPGPUs -2 ............................................. 93
   4.7. Structure and operation of the Fermi GF100 GPGPU ............................................. 94
       4.7.1. Layout of a Cuda GF100 SM [19] ...................................................... 94
       4.7.2. A single ALU (“CUDA core”) ...................................................... 95
   4.8. Principle of the SIMT execution in case of serial kernel execution ............................. 96
   4.9. Principle of operation of a Fermi GF100 GPGPU ............................................. 96
   4.10. Subtasks of work scheduling ...................................................... 96
   4.11. Scheduling kernels to SMs [25], [18] ............................................. 97
   4.12. Scheduling thread blocks associated with the same kernel to the SMs .......................... 97
   4.13. The notion and main features of thread blocks in CUDA [53] ...................................... 98
   4.15. Scheduling warps for execution in SMs ............................................. 100
   4.16. Assumed principle of operation-1 ............................................. 102
   4.17. Assumed principle of operation-2 ............................................. 103
   4.18. Assumed principle of operation-3 ............................................. 104
   4.20. Scheduling policy of warps in an SM of Fermi GF100-1 ............................................. 107
   4.21. Scheduling policy of warps in an SM of Fermi GT100-2 ............................................. 108
   4.22. Scheduling policy of warps in an SM of the G80 indicating coarse grain warp scheduling ............................................. 109
   4.23. Scheduling policy of warps in an SM of the G80 indicating fine grain warp scheduling ............................................. 109
   4.24. Estimation of the peak performance of the Fermi GF100 -1 ............................................. 110
   4.25. Estimation of the peak performance of the Fermi GF100 -2 ............................................. 112
5. Microarchitecture of Fermi GF104 ............................................. 114
   5.1. Available execution resources per SM in the GF104 vs the GF100 ............................................. 115
   5.2. Throughput of arithmetic operations per clock cycle per SM in the GF104/114 [13] 116
   5.3. Warp issue in the Fermi GF104 ............................................. 116
   5.4. Peak computational performance data for the Fermi GF104 based GTX 460 card 117
6. Microarchitecture of Fermi GF110 ............................................. 118
   6.1. Overall structure of the Fermi GF110 core [65] ............................................. 118
   6.2. Block diagrams of the cores (SM units) of the four sub-families of Fermi [66] .......................... 119
   6.3. Throughput of arithmetic operations per clock cycle per SM in the GF104/114 [13] 120
7. Microarchitecture of Fermi GF114 ............................................. 121
   7.1. Overall structure of the Fermi GF114 core in the GTX 560 card [67] .......................... 121
   7.2. Microarchitecture of Fermi GF114 ............................................. 122
   7.3. Block diagrams of the cores (SM units) of the four sub-families of Fermi [66] .......................... 123
   7.4. Throughput of arithmetic operations per clock cycle per SM in the GF104/114 [13] 123
5. Nvidia’s Kepler family of graphics cores and cards ............................................. 125
1. Overview of Nvidia’s Kepler family ............................................. 125
   1.1. Sub-families of Kepler ............................................. 125
   1.2. The GK104 Kepler core [76] ............................................. 125
   1.3. The GK110 Kepler core [70] ............................................. 126
   1.4. Main features of Nvidia’s Kepler GK104 and GK110 cores [70] ............................................. 126
2. Key innovations of Kepler’s PTX 3.x ............................................. 127
3. Major innovations of Kepler’s microarchitecture ............................................. 127
   3.1. a) Dynamic parallelism [70] ............................................. 127
       3.1.1. Principle of dynamic parallelism in Kepler GPUs ............................................. 128
       3.1.2. Benefits of dynamic parallelism [70] ............................................. 128
   3.2. b) The Hyper-Q scheduling mechanism ............................................. 128
       3.2.1. Speed up potential of the Hyper-Q scheduling mechanism ............................................. 133
   3.3. c) Grid Management Unit ............................................. 134
   3.4. d) The GPU Direct feature ............................................. 134
   3.5. Major enhancements of Kepler’s microarchitecture ............................................. 135
3.5.1. a) Setting down the operating frequency of the execution units (shader clock frequency) to the core frequency-1 ............................................. 136
3.5.2. a) Setting down the operating frequency of the execution units (shader clock frequency) to the core frequency-2 .................................... 136
3.5.3. a) Setting down the operating frequency of the execution units (shader clock frequency) to the core frequency-3 .................................... 136
3.5.4. a) Setting down the operating frequency of the execution units (shader clock frequency) to the core frequency-4 .................................... 136
3.5.5. Comparing the silicon area needed and power consumption of the design approaches used to implement Fermi’s and Kepler’s execution units [76] .... 137
3.5.6. b) Simplifying hardware dependency checks by introducing compiler hints 139
3.5.7. c) Introducing quad warp schedulers per SMX .................................. 139
3.5.8. Available execution resources in a Kepler SMX vs. a Fermi SM ......... 139
3.5.9. d) Quadrupling the number of registers that are accessible per thread ... 140
3.5.10. Consequence of quadrupling the number of registers that can be accessed per thread in the cores of Kepler ............................................. 141
3.5.11. e) Introduction of a 48 KB Read-Only data cache for general use .......... 141
3.5.12. f) Doubled L2 cache size and bandwidth vs. Fermi .......................... 142
4. Nvidia’s GK104 Kepler core and related graphics cards .......................................................... 142
4.1. Sub-families of Kepler ................................................................. 142
4.2. The GK104 Kepler core [76] .......................................................... 143
4.3. Die photo of Nvidia’s Kepler GK104 core [78] .................................... 143
4.4. Block diagram of an SMX of the Kepler GK104 core [76] ...................... 144
4.5. Peak FP32 and FP34 performance of the GK104 based GeForce GTX 680 graphics card 145
4.5.1. a) Peak FP32 performance of the GK104 based GeForce GTX 680 graphics card-1 ................................................................. 145
4.5.2. a) Peak FP32 performance of the GK104 based GeForce GTX 680 graphics card-2 [70] ................................................................. 145
4.5.3. a) Peak FP32 performance of the GK104 based GeForce GTX 680 graphics card-3 ................................................................. 146
4.5.4. b) Peak FP64 performance of the GK104 based GeForce GTX 680 graphics card-1 ................................................................. 146
4.5.5. b) Peak FP64 performance of the GK104 based GeForce GTX 680 graphics card-2 [70] ................................................................. 147
4.5.6. b) Peak FP64 performance of the GK104 based GeForce GTX 680 graphics card-3 ................................................................. 147
5. Nvidia’s GK110 Kepler core and related graphics cards ......................................................... 147
5.1. Sub-families of Kepler ................................................................. 147
5.2. The GK110 core ......................................................................... 148
5.3. The GK110 Kepler core [70] .......................................................... 148
5.3.1. Die photo of Nvidia’s Kepler GK110 core [70] .................................. 148
5.3.2. Die layout of Nvidia’s Kepler GK110 core [79] .............................. 149
5.3.3. Comparison of the die sizes of Nvidia’s GK110 and GK104 Kepler cores [79] 150
5.3.4. Block diagram of an SMX of the Kepler GK110 core [70] ............... 151
5.3.5. a) Peak FP32 performance of the GK110 based GeForce GTX Titan graphics card ................................................................. 152
5.3.6. b) Peak FP64 performance of the GK110 based GeForce GTX Titan graphics card ................................................................. 152
6. Evolution of Nvidia’s microarchitectures .............................................................................. 152
6.1. a) Evolution of FP32 warp issue efficiency in Nvidia’s pre-Fermi GPGPUs .... 153
6.2. b) Evolution of FP32 warp issue efficiency in Nvidia’s Fermi and Kepler GPGPUs 153
6.3. c) FP64 performance increase in Nvidia’s pre-Fermi GPGPUs ............... 153
6.4. d) FP64 performance increase in Nvidia’s Tesla and GPGPUs ............... 154
6. AMD’s Heterogeneous Systems Architecture concept ....................................................... 155
1. Introduction to the HSA concept ...................................................................................... 155
1.1. First public introduction of the HSA concept (6/2011) [84], [85] ............... 155
1.2. Aims of the Heterogeneous Systems Architecture (HSA) concept .............. 156
1.2.1. AMD’s view of the evolution of single core, multi-core and application performance (6/2011) [84] ................................................................. 156
1.2.2. AMD’s view of the evolution path to heterogeneous computing (6/2011) [84] 156

2. The vision of the HSA concept ................................................................. 156
2.1. a) The memory model of HSA [87] ......................................................... 157
2.2. b) User-level command queuing [87] ...................................................... 157
2.3. d) Model of processing applications [88] ................................................. 158
2.4. Principle of processing applications in HSA [87] ....................................... 158
2.5. The runtime phase [87] ...................................................................... 159
2.6. Heterogeneous System Architecture Foundation (HSAF) ....................... 161
2.7. AMD’s roadmap for the implementation of HSA [90] ............................... 161
2.8. AMD’s roadmap for the implementation of HSA - a more detailed view [85] ... 162

7. AMD’s Southern Islands family of graphics cores and cards .......................... 163
1. Overview of AMD’s Southern Island family ............................................. 163
1.1. AMD’s roadmap for the implementation of HSA (Financial Analyst Day 2/2012) [90] 166
1.2. AMD’s new GPU microarchitecture: the Graphics Core Next (GCN) .......... 163
1.3. Announcement and implementation of AMD’s GCN ............................... 164
1.4. AMD’s roadmap indicating the introduction of GCN along with the Southern Islands line (2/2012) [94] ........................................................... 164

2. Major innovations of the Southern Islands microarchitecture ....................... 164
2.1. AMD’s move from the VLIW4-based GPU design to the SIMD-based GPU design 166
2.1.1. Main steps of the evolution of the microarchitecture of ATI’s (acquired by AMD in 2006) and AMD’s GPU’s [95] .............................................. 165
2.1.2. Principle of using VLIW-based ALUs in GPUs ................................... 165
2.1.3. VLIW-based ALUs ..................................................................... 166
2.1.4. Example for encoding wavefront queues in form of VLIW4 instructions [96] 166
2.1.5. Scheduling the wavefronts given previously while assuming VLIW4 encoding [96] ................................................................. 167
2.1.6. Graphics processing and VLIW-based ALUs ..................................... 168
2.1.7. AMD’s motivation for using VLIW-based ALUs in their previous GPU’s 168
2.1.8. Number of execution units (EUs) in Nvidia’s and AMD’s GPUs ............ 168
2.1.9. AMD’s motivation for using VLIW5-based ALUs in their first GPU’s [97] 169
2.1.10. AMD’s VLIW5 design .................................................................. 169
2.1.11. Replacing the original VLIW5 design by the Enhanced VLIW5 design 170
2.1.12. Replacing the Enhanced VLIW5 design by the VLIW4 design-1 ......... 171
2.1.13. Replacing the Enhanced VLIW5 design with the VLIW4 design-2 ....... 171
2.1.14. Contrasting the main features of AMD’s Enhanced VLIW5 and VLIW4 designs [17] ............................................................. 172
2.1.15. Shift to numeric computing and AMD’s move to the GCN architecture 174
2.1.16. Replacing the VLIW4-based design by SIMD-based design in AMD’s GCN architecture [91] .................................................. 175

2.2. AMD ZeroCore technology [49] .............................................................. 175
2.2.1. Achieved reduction of power consumption in long idle periods in AMD’s GPUs due to using the AMD ZeroCore Power Technology [104] ............... 176
2.2.2. The AMD PowerTune technology .................................................... 176
2.2.3. The PowerTune technology as introduced in the Northern Island line (HD 6900/Cayman), 12/2010) [106] ...................................................... 176
2.2.4. Benefit of using the PowerTune technology [105] ............................... 177

3. AMD’s Southern Islands card family .......................................................... 178
3.1. Overview of the Southern Islands family ............................................... 178
3.1.1. Announcement and implementation of AMD’s GCN ......................... 178
3.1.2. Key features of the Southern Island Family ..................................... 179
3.1.3. Subfamilies of the Southern Island Family-1 [107] .............................. 179
3.1.4. Comparison of the die sizes of AMD’s and Nvidia’s graphics cores [95] 180
3.1.5. Announcement and implementation of AMD’s GCN ....................... 180

3.2. Microarchitecture of the Tahiti XT core used in the Radeon HD7970 card ....... 181
3.2.1. Overall architecture of the Tahiti XT core [95] .................................. 181
3.2.2. Difference between the cores of the 7950 (Tahiti Pro) and the 7970 (Tahiti XT) cards ................................................................. 181
3.2.3. Overall architecture of a Compute Unit (CU) – Simplified block diagram [95] 182
3.2.4. Block diagram of the Compute Unit – Detailed block diagram [108] ... 182
3.2.5. Main units of a CU ........................................................................... 183
3.2.6. The four 16-wide SIMD units of the CU [110] (Strongly simplified discussion) 183
3.2.7. The Scalar unit of the CU [110] ........................................................... 184
3.2.8. Peak FP32 and FP64 performance figures of the HD 7970 (Tahiti XT) ..... 184
3.2.9. The cache hierarchy of the HD 7970 (Tahiti XT) [95] .......................... 184
3.2.10. Available data spaces for execution per CU and per GPU in the HD 7970 (Tahiti XT) [110] .......................... 185
3.2.11. Memory sizes and peak read bandwidth values in the HD 7970 (Tahiti XT) [16] .............................. 185
3.3. Principle of operation (repeated from Section 3.2) ................................. 186
  3.3.1. Compute workloads ............................................................................ 186
  3.3.2. Work items ........................................................................................ 187
  3.3.3. The kernel .......................................................................................... 188
  3.3.4. Work-groups ....................................................................................... 188
  3.3.5. Wavefronts-1 ..................................................................................... 189
  3.3.6. Wavefronts-2 ..................................................................................... 189
  3.3.7. Principle of operation ......................................................................... 190
  3.3.8. Launching compute workloads to the GPU ...................................... 190
  3.3.9. Launching compute workloads to the GPU ...................................... 190
  3.3.10. Execution of compute workloads-1 ............................................... 191
  3.3.11. Execution of compute workloads-2 ............................................... 191
  3.3.12. Execution of compute workloads-3 ............................................... 191
3.4. Achieved performance figures and power consumption of the HD 7970 (Tahiti XT) 192
  3.4.1. a) Gaming performance ................................................................. 192
  3.4.2. a) Gaming performance of high end graphics cards-1 [50] .............. 192
  3.4.3. a) Gaming performance of high end graphics cards-2 [50] .............. 192
  3.4.4. b) Performance figures of high end graphics cards measured for a compute workload [50] ................................................................. 193
  3.4.5. c) Load power consumption (Power consumption at full load) [50] ..... 194
  3.4.6. d) Idle power consumption [50] ..................................................... 195
8. AMD’s Sea Islands family ............................................................................. 197
  1. The Sea Islands family on AMD’s Client and Graphics roadmap [111] ........ 197
  2. Key features of the Sea Islands family ....................................................... 197
  3. Expected introduction of AMD’s 2. generation GCN, termed as Sea Islands family .... 198
  4. Key parameters of the HD 8950/70 graphics cards .................................. 198
9. Outlook ........................................................................................................ 199
  1. Nvidia’s roadmap for 2014 and beyond .................................................... 199
  1.1. The Maxwell CPU + GPU ................................................................. 199
  1.2. The Volta CPU + GPU ................................................................. 200
  1.3. The Volta processor with stacked DRAM [113] ..................................... 200
  2. AMD’s graphics roadmaps for 2013 ......................................................... 200
  3. Intel’s Xeon Phi line ............................................................................... 201
  3.1. The Knights Ferry prototype version .................................................... 202
  3.2. Renaming the MIC branding to Xeon Phi and providing open source software support 202
  3.3. The Knights Corner consumer processor ........................................... 203
  3.4. Key features related to the Xeon Phi family [120] .................................. 203
  3.5. First introduced subfamilies of the Xeon Phi coprocessor line [121] ......... 204
  3.6. System layout of the Knights Corner (KCN) DPA [120] ........................ 205
  3.7. Microarchitecture of Knights Corner [120] ........................................ 205
  3.8. Layout of the ring interconnect on the die [122] .................................... 206
  3.9. Block diagram of a core of the Knights Corner [120] ............................ 206
  3.10. Block diagram and pipelined operation of the Vector unit [120] .......... 207
  3.11. System architecture of the Xeon Phi coprocessor [122] ............ 207
3.12. The Xeon Phi 5110P coprocessor [124] ......................................................... 208
3.13. The Xeon Phi coprocessor board (backside) [122] ........................................ 209
3.15. Principle of Intel’s common software development platform for multicores, many-cores and clusters [125] ................................................................. 209
3.16. The power efficiency of Intel’s Knight Corner vs. competition [120] .......... 210
3.17. Peak performance of the Xeon Phi 5110P and SE10P/X vs. a 2-socket Intel Xeon server [126] ............................................................................................................. 210
3.18. Performance of Xeon Phi 5110P vs. a 2-socket Sandy Bridge based Xeon server [126] 211
3.19. Assessing the achieved speed-up of the Xeon Phi 5110P vs a 2-socket Xeon server 212
3.20. Intel’s roadmap for the Xeon Phi line [127] ..................................................... 212
3.21. Assessing the peak FP64 performance of the Knights Corner coprocessor with that of Nvidia’s and AMD’s recent devices ............................................. 213
3.22. d) FP64 performance increase in Nvidia’s Tesla and GPGPUs .................... 213
3.23. Number of execution units (EUs) in Nvidia’s and AMD’s GPUs ............... 214
10. References ........................................................................................................... 215
Aim

To provide an understanding about the architecture and principle of operation of GPGPUs and give case examples of up-to-date GPGPU families.

Remark

About 100 slides out of more than 400 are taken from a previous material worked out in the framework of the TÁMOP-4.1.2-08/2/A/KMR-2009-0053 project.
1. fejezet - Introduction to GPGPUs

1. Representation of objects by triangles

1.1. ábra -

Vertices

- have three spatial coordinates
- supplementary information necessary to render the object, such as
  - color
  - texture
  - reflectance properties
  - etc.

Example: Triangle representation of a dolphin [1]

1.2. ábra -
1.1. Main types of shaders in GPUs

1.3. ábra -

1.4. ábra - Pixel/vertex shader models (SM) supported by subsequent versions of DirectX and MS’s OSs [2], [3]
2. Convergence of important features of the vertex and pixel shader models

Early shader models (models 2 and 3) had different precision requirements, consequently data types, register sets and instruction sets (i.e. ISAs).

These models became integrated in the shader model 4, as indicated below.

Shader model 2 [4]
- Different precision requirements
  - Different data types
    - Vertex shader: FP32 (coordinates)
    - Pixel shader: FX24 (3 colors x 8)
  - Different instructions
  - Different programming resources (e.g. registers)

Shader model 3 [4]
- Unified precision requirements for both shaders (FP32)
  - with the option to specify partial precision (FP16 or FP24) by adding a modifier to the shader code
- Different instructions
- Different programming resources (e.g. registers)

Shader model 4 (introduced with DirectX10) [5]
- Unified precision requirements for both shaders (FP32)
with the possibility to use new data formats.

- Unified instruction set
- Unified programming resources (e.g. temporary and constant registers)

Shader architectures of GPUs prior to SM4

GPUs prior to SM4 (DirectX 10):

have separate vertex and pixel units with different features.

Drawback of having separate units for vertex and pixel shading

- Inefficiency of the hardware implementation
- (Vertex shaders and pixel shaders often have complementary load patterns [3]).

1.5. ábra -

1.6. ábra - Principle of the unified shader architecture [6]
1.7. ábra -

Based on its FP32 computing capability, the unified shader is a prospect.

3. Peak FP32/FP64 performance of Nvidia’s GPUs vs Intel’ P4 and Core2 processors [7]
4. Peak FP32 performance of AMD’s GPUs [8]

1.9. ábra -
5. Evolution of the FP-32 performance of GPUs [9]

6. Evolution of the bandwidth of Nvidia’s GPU’s vs Intel’s P4 and Core2 processors [7]
1.12. ábra - Contrasting the utilization of the silicon area in CPUs and GPUs [10]

- Less area for control since GPGPUs have simplified control (same instruction for all ALUs)
- Less area for caches since GPGPUs support massive multithreading to hide latency of long operations, such as memory accesses in case of cache misses.

7. Comparing main features of CPUs and GPUs [12]

1.13. ábra -
<table>
<thead>
<tr>
<th>Example Device</th>
<th>AMD Phenom II X6 1055T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Frequency</td>
<td>2.80 GHz</td>
</tr>
<tr>
<td>Compute Units</td>
<td>68</td>
</tr>
<tr>
<td>Approx. Power(^1)</td>
<td>98</td>
</tr>
<tr>
<td>Approx. Power/Compute Unit</td>
<td>14</td>
</tr>
<tr>
<td>Peak Single-Precision Billion Floating-Point Ops/Sec</td>
<td>66</td>
</tr>
<tr>
<td>Approx GFLOPS/Watt</td>
<td>0</td>
</tr>
<tr>
<td>Max In-flight HW Threads</td>
<td>128</td>
</tr>
<tr>
<td>Simultaneous Executing Threads</td>
<td>-</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>256</td>
</tr>
<tr>
<td>Int Add latency</td>
<td>0.4</td>
</tr>
<tr>
<td>FP Add Latency</td>
<td>1.2</td>
</tr>
<tr>
<td>Approx DRAM Latency</td>
<td>50</td>
</tr>
<tr>
<td>L2+L3 (GPU only L2) cache capacity</td>
<td>8196</td>
</tr>
<tr>
<td>Approx Kernel Launch Latency</td>
<td>25</td>
</tr>
</tbody>
</table>

\(^1\) For the power specifications of the AMD Phenom II X6 1055T, see the AMD website at https://www.amd.com/processors/phenom-ii/Pages/phenom-ii-motherboards.aspx.
2. fejezet - The virtual machine concept of GPGPU computing

1. Introduction to the virtual machine concept of GPGPU computing

1.1. The virtual machine concept as used for GPGPUs

Program development for GPGPUs is based on the virtual machine concept, as shown below.

2.1. ábra -

1.2. Benefits of the portability of the pseudo assembly code

- The compiled pseudo ISA code (PTX code/IL code) remains independent from the actual hardware implementation of a target GPGPU, i.e. it is portable over subsequent GPGPU families.

- Forward portability of the object code (GPGPU code, e.g. CUBIN code) is provided, however, typically only within major versions (e.g. 1.x or 2.x).

- Compiling a PTX/IL file to a GPGPU that misses features supported by the particular PTX/IL version however, need emulation for features not implemented in hardware.

  This slows down execution.

- Portability of pseudo assembly code (Nvidia’s PTX code or AMD’s IL code) is highly advantageous in the recent rapid evolution phase of GPGPU technology as it results in less costs for code refactoring.

- Code refactoring costs are a kind of software maintenance costs that arise when the user switches from a given generation to a subsequent GPGPU generation (like from GT200 based devices to GF100 or GF110-based devices) or to a new software environment (like from CUDA 1.x SDK to CUDA 2.x or from CUDA 3.x SDK to CUDA 4.x SDK).

1.3. The process of developing applications-1

With the virtual machine concept used application development becomes a two-phase process.
The virtual machine concept of GPGPU computing

• Phase 1: Compiling the HLL application to pseudo assembly code

2.2. ábra -

The compiled pseudo ISA code (PTX code/IL code) remains independent from the actual hardware implementation of a target GPGPU, i.e. it is portable over different GPGPU families.

1.4. The process of executing applications-2

• Phase 2: Compiling the pseudo assembly code to GPU specific binary code

2.3. ábra -

The object code (GPGPU code, e.g. a CUBIN file) is forward portable, but forward portability is provided typically only within major GPGPU versions, such as Nvidia’s compute capability versions 1.x or 2.x.

1.5. Formulation of application programs at different levels of abstraction

• Application programs may be formulated at different levels of abstractions while using appropriate language tools (development kits), as indicated below.

2.4. ábra -
The virtual machine concept of GPGPU computing

Also computing devices may be specified at different levels of abstractions, as shown later.

Remark

The virtual machine concept underlying both Nvidia’s and AMD’s GPGPUs is similar to the virtual machine concept underlying Java.

- For Java there is also an inherent pseudo ISA definition, called the Java bytecode.
- Applications written in Java will first be compiled to the platform independent Java bytecode.
- The Java bytecode will then either be interpreted by the Java Runtime Environment (JRE) installed on the end user’s computer or compiled at runtime to the object code by the Just-In-Time (JIT) compiler of the user.

2. Introduction to a GPGPU-based massively data parallel computational model

2.1. Specification of GPGPUs at different levels of abstraction

Also GPGPUs may be specified at two levels

- at a virtual machine level (pseudo ISA level, pseudo assembly level, intermediate level) and
- at the object code level (real GPGPU ISA level).

2.5. ábra -
Subsequently, we will focus on the specification of GPGPUs on the virtual machine level as it provides more stable and usable knowledge about GPGPUs than the more dynamically changing hardware specification level (including the real ISA).

2.2. Specification of the GPGPU virtual machine

It includes two related concepts

- a GPGPU-based massively data parallel computational model and
- the related pseudo ISA of GPGPU computing.

The GPGPU-based massively data parallel computational model underlies the pseudo ISA by specifying the operational environment and principles of operation taken for granted for the pseudo ISA.

First we will be concerned with the GPGPU-based massively data parallel computational model, the related pseudo ISA of GPGPUs is discussed subsequently, in Section 2.3.

2.3. The GPGPU-based massively data parallel computational model

The problem of discussing the GPGPU-based massively data parallel computational model is that there is no consensus on such a model, as

- both major firms developing GPGPUs (Nvidia and AMD) maintain somewhat different GPGPU-based massively parallel computational models and
- in addition, also their computational models evolve; partly to follow the evolution of the HLL (High Level Language) computational environment (e.g. subsequent CUDA or OpenCL versions) and partly to support new GPGPU hardware features.

2.4. Emergence of GPGPU-based massively data parallel computational models

Both major firms designing GPGPUs developed and published GPGPU-based massively data parallel computational models, as the subsequent brief overview shows.

- The term SIMT (Single Instruction Multiple Threads) was introduced by Nvidia in 2006 in conjunction with launching their G80 GPGPU.

Nvidia used this term to designate a massively data parallel execution model where a large number of independent threads execute concurrently on a GPGPU [11].
In 2007 also AMD introduced a similar concept for massively data parallel execution and designated it the ATI Stream Technology.

Later, in 10/2010 AMD renamed it to AMD Accelerated Parallel Processing technology (APP) at the same time when they introduced the HD 68xx (Northern Island series) of GPGPUs,

Nevertheless, the real incentive of renaming their GPGPU related technology was replacing the original Brook+ HLL programming environment by the OpenCL environment already in their previous Evergreen family (HD 58xx) in 2009.

- In addition, we note that OpenCL’s GPGPU-based massively data parallel computational model is designated as the SPMD (Single Program Multiple Data) model (announced in 12/2008).

### 2.5. The choice of the computational model presented

- As there is no real reason to prefer one or the other computational model of the two major firms leading the development of GPGPUs (Nvidia, AMD) we will discuss GPGPU-based massively data parallel computational models in a slightly more general way by describing a computational model that includes key features of both computational models developed and continuously enhanced by both Nvidia and AMD.

### 2.6. The choice of the terminology used

#### 2.6.1. Designation of the GPGPU-based massively data parallel computational model as the SIMT (Single instruction Multiple Threads) model of computation

- In these slides we prefer to refer to this model as the SIMT (Single Instruction Multiple Threads) model of computation (as done by Nvidia) as it appropriately expresses both the multi-threaded feature and the similarity to the SIMD computational model.

  Note that albeit we use the same designation as Nvidia we interpret this model differently.

#### 2.6.2. Designation of the terms related to GPGPU-based massively data parallel computational model

Both Nvidia and AMD make use for the most part different terms related to their GPGPU-based massively data parallel computational models.

Moreover, while AMD replaced their Brook+ high level programming environment by OpenCL they changed almost completely their related terminology.

In addition, in the associated documentation there is no consistent use of the terminology, documentations targeting different fields, such as hardware, pseudo assembly programming or high level programming, often make use of different designations for the same notion, as shown in the Table below.

As far as the terminology of the present slides concerns, we typically make use of the terminology associated with OpenCL due to the more and more widespread use of this programming environment.

#### 2.6. ábra - Terminologies used with GPGPUs/Data parallel accelerators
3. Introduction to the SIMT computational model

3.1. Main components of the SIMT computational models

The SIMT computational model involves the following three key abstractions.

2.7. ábra -

The model of the compute workload

(Section 2.3.2)

3.2. The model of the compute workload

2.8. ábra -
3.2.1. The thread model

2.9. ábra -

The thread (Section)

3.2.1.1. The notion of the compute workload

A compute workload for the GPU is to execute a data parallel program segment, called kernel on up to 3-dimensional data structures, i.e. to perform computations in an N-Dimensional Range, (called also as the Domain of execution) like the one shown in the Figure below.

2.10. ábra - The interpretation of N-Dimensional Ranges [12]
3.2.1.2. The thread model

It is a hierarchical model, based on the notions work-items, work-groups and wavefronts, to be discussed next.

3.2.1.3. Work items

Work-items represent basic units of computation, they typically are given as index points in the data space.

A work-item, or more precisely the processing on a work item according to the flow of instructions will be interpreted as a thread.

2.11. ábra - The interpretation of a work-item (Based on [12])
3.2.1.4. Work-groups

Domains of execution (N-Dimensional Ranges) are segmented by the programmer to work-groups, termed also as thread blocks, as indicated in the Figure below, for efficient execution.

A work-group has an identifier and will be scheduled for execution onto a particular Compute Unit (CU).

2.12. ábra - The interpretation of wavefronts [12]
Remark

Work-groups are designated differently,

- as Thread blocks by Nvidia and
- as Thread blocks (Pre-OpenCL term) or Work Groups (sometimes as Workgroups) (OpenCL term) by AMD.

3.2.1.5. Wavefronts-1

For efficient execution work-groups are segmented by the GPGPU to wavefronts (warps) that will be processed in lock-step on the same CU.

2.13. ábra - The interpretation wavefronts [12]

Wavefronts belonging to the same work-group can share data and their run can be synchronized by appropriate instructions to force each wavefront to wait until all other wavefronts reach the same instruction, as discussed later.
3.2.1.6. Wavefronts-2

- Wavefronts represent the smallest unit of work that will be scheduled for execution by the CU for its Processing Elements.

- Wavefronts are called as such by AMD and as warps by Nvidia.

- Wavefront sizes are hardware specific.
  - In Nvidia’s GPGPUs the wavefronts (called warps) include 32 work-items.
  - AMD’s GPGPUs have different wavefront sizes;
    - High performance GPGPU cards, like those of the Southern Island lines (HD 7950/70 cards) have typically wavefront sizes of 64, whereas
    - Lower performance cards may have wavefront sizes of 32, 24 or even 16.

- Each wavefront has a single Program Counter (PC) that points to the next instruction to be executed by the wavefront.
  When a wavefront is created, the PC is initialized to the first instruction of the program.

- Wavefronts will be sent for execution to the Processing Elements of the CU by the scheduler of the CU.
  Wavefronts are collections of work-items grouped together for efficient processing on the CU.

3.2.2. The kernel concept

2.14. ábra -

The n

The thre

3.2.2.1. The kernel concept-1

The programmer describes the set of operations to be done over the entire Domain of Execution by kernels.

2.15. ábra - Interpretation of the kernel concept
Kernels are specified at the HLL level and compiled to the intermediate level.

### 3.2.2.2. The kernel concept-2

The kernel is the data parallel program to be executed on the Domain of Execution (N-Dimensional Range).

It consists of instructions, like

- Vector instructions that instruct e.g. all 16 EUs of a CU,
- Vector memory instructions that transfer data (read or write data) between the GPU memory and the vector register file (VGPR) of a CU, etc.

Remark

Kernels designate compute oriented data parallel programs whereas shaders are graphics oriented programs to be executed on GPUs.

### 3.2.2.3. The kernel concept-3

Dedicated HLLs like OpenCL or CUDA C allow the programmer to define kernels, that, when called are executed in parallel by n different threads, as opposed to only once like for regular C functions.

### 3.2.2.4. Specification of kernels

- A kernel is defined by
  - using a declaration specifier (like `_kernel` in OpenCL or `_global_` in CUDA C) and
  - declaring the instructions to be executed.

### 3.2.2.5. Sample codes for kernels

The subsequent sample codes illustrate two kernels that adds two vectors (a/A) and (b/B) and store the result into vector (c/C).

2.16. ábra -
The virtual machine concept of
GPGPU computing

CUDA C [13]

// Kernel definition
_global_ void VecAdd(float* A, float* B,
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

Remark
During execution each thread is identified by a unique identifier that is

- int i in case of CUDA C, accessible through the threadIdx variable, and
- int id in case of OpenCL accessible through the built-in get_global_id() function.

3.3. The model of the platform

2.17. ábra -

The computational resources available at the virtual machine level.

3.3.1. The model of computational resources

2.18. ábra -
It is structured in a hierarchical manner, as shown below.

2.19. ábra - The model of computational resources [15]

| The set of PEs allows a “wide” SIMD type execution on multiple data, e.g. on 16 FX/FP data pairs | Proces: Element (PE) |

3.3.1.1. Compute Units (CU)

They have different implementations/designations, as shown subsequently, e.g.

- 16 VLIW5/VLIW4 ALUs in AMDs pre Southern Islands lines or
- 2x16 FX/FP32+16 L/S+4 SFU (Special Functions Unit) EUs in Nvidia’s Fermi100/110 GPUs.

The special kind of the implementation of a CU isn’t belonging to the platform model, but is part of the microarchitecture of a particular GPGPU core.

3.3.1.2. Example 1: Structure of AMD’s Cypress cores in their Evergreen series [16]

2.20. ábra -

3.3.1.3. Layout of AMD’s Enhanced VLIW5 designs [17]
2.21. ábra -

Enhanced VLIW5 design

RV770 core/HD4870 card, (2008)

Evergreen line (RV870 core/HD5870 card), (2009)

- 5 FX32 or 5 FP32 operations or
- 1 FP64 operation or
- 1 transcendental + 4 FX/FP 32 operations

per cycle.

3.3.1.4. Example 2: Structure of Nvidia’s GF100/GF110 cores in their Fermi series [11], [18]

2.22. ábra -
3.3.1.5. Layout of a CU of the GF100 core [19]

2.23. ábra -

<table>
<thead>
<tr>
<th>Available execution resources per CU in the GF100/110 core</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GF100</strong></td>
</tr>
<tr>
<td>No. of FX/FP32 EUs(^1)</td>
</tr>
<tr>
<td>No. of L/S units</td>
</tr>
<tr>
<td>No. of SFUs</td>
</tr>
<tr>
<td>No. of DP FP ALUs</td>
</tr>
</tbody>
</table>

\(^1\) An FP64 instruction will be forwarded to two FP32 execution units of both groups of 16 FP32 units in the same time, thus FP64 instructions enforce single issue.

Remark

The FX/FP32 EUs are designated as “Core”s by Nvidia

3.3.1.6. Different designations for the term Computing Unit (CU)

Streaming multiprocessor (SM/SMX) (Nvidia).
Superscalar shader processor, Compute Unit (CU) (AMD),
Wide SIMD processor, CPU core (Intel).

### 3.3.2. The memory mode

#### 2.24. ábra -

The memory model at the virtual machine level declares all data spaces available at this level along with their features, like their accessibility, access mode (read or write) access width etc.

#### 2.25. ábra - Overview of available memory spaces in GPGPUs

As key components of the memory model we consider the following data spaces available at the virtual machine level:

#### 3.3.2.1. b) Interpretation of the memory model in OpenCL [15]

#### 2.26. ábra -
3.3.2.2. Accessibility of memory spaces by the host and the kernels in OpenCI's memory model [15]

2.27. ábra -

<table>
<thead>
<tr>
<th>Host access</th>
<th>Kernel access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read / Write access</td>
<td>Read / Write access</td>
</tr>
</tbody>
</table>

Remarks

1. AMD introduced Local memories, designated as Local Data Share, only along with their RV770-based HD 4xxx line in 2008.

2. Beyond the key data space elements discussed so far, there may be also other kinds of memories declared at the virtual machine level, such as AMD’s Global Data Share, an on-chip Global memory introduced with their RV770-based HD 4xxx line in 2008).

3. Max. sizes of particular data spaces are specified by the related instruction formats of the intermediate language.

4. Actual sizes of particular data spaces are implementation dependent.

5. Traditional caches are not visible at the virtual machine level, as they are typically transparent during program execution.

Nevertheless, more advanced GPGPUs allow an explicit cache management at the virtual machine level, by providing e.g. data prefetching instructions.

In these cases the memory model needs to be extended with these caches, accordingly.

3.3.2.3. Example 1: Nvidia’s memory model underlying PTX 3.1 (2012) [20]

(The Constant memory space although available not indicated in the Figure)

2.28. ábra -
3.3.2.4. Principle of the implementation of Nvidia’s memory model underlying PTX 3.1 [20]

2.29. ábra -

3.3.2.5. Nvidia’s memory model – OpenCL vs. CUDA terminology and implementation details in Nvidia’s Fermi and Tesla lines [21]

2.30. ábra -
3.3.2.6. Example 2: Data spaces in AMD’s memory model underlying IL vers. 2.0 (simplified)

2.31. ábra - Available data spaces in AMD’s IL vers. 2.0 [22]

<table>
<thead>
<tr>
<th>Data space</th>
<th>Access type</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Purpose Registers</td>
<td>R/W</td>
</tr>
<tr>
<td>Local Data Share (LDS)</td>
<td>R/W</td>
</tr>
<tr>
<td>Constant Register (CR)</td>
<td>R</td>
</tr>
<tr>
<td>Global Data Share</td>
<td>R/W</td>
</tr>
<tr>
<td>Device Memory</td>
<td>R/W</td>
</tr>
</tbody>
</table>

Remarks

- Max. sizes of data spaces are specified along with the instructions formats of the intermediate language.
- The actual sizes of the data spaces are implementation dependent.

3.3.2.7. Implementation of AMD’s memory model in their Cayman core (Northern Islands/HD 69xx) [23]

2.32. ábra -
3.4. The execution model

We interpret the execution model of the SIMT model of computation as built up of the following five key abstractions:

2.33. ábra - Key abstractions of the execution model

3.4.1. The concept of SIMT execution

2.34. ábra - Key abstractions of the execution model

The concept of SIMT execution units three concepts

a) massive multithreading to hide stalls during execution

b) SIMD-like execution to utilize data parallelism and

c) zero-penalty multithreading in order to implement an effective switch among wavefronts.

3.4.1.1. a) Massive multithreading
Wile GPGPUs process wavefronts long latency operations, such as loads from the device memory may cause long stalls in the order of 100 cycles.

Such stalls can be hidden when stalled wavefronts will be suspended and a new wavefront that is ready for execution will be sent to available execution resources (PEs), as shown below.

2.35. ábra - Scheduling wavefronts for hiding stalls due to device memory reads [24]

This kind of operation can be implemented e.g. when after issuing a memory read to a wavefront the scheduler simply switches to another runnable wavefront.

Remark

The type of scheduling shown above is called coarse grained scheduling, as the wavefronts are allowed to run as long as they do not stall.

By contrast, in case of fine grain scheduling the scheduler would select the next wavefront to be run in every new cycle.

Note

When there are typically enough ready to run wavefronts available, massive multithreading can effectively hide stalls due to memory loads.

To achieve this, e.g. Nvidia’s Fermi-based GPGPUs can maintain up to 48 wavefronts (warps) per CU.

With massive multithreading less emphasis can be laid on the implementation of the cache system and the redeemed silicon area (used otherwise for implementing caches) may be utilized for performing computations, as indicated below.

2.36. ábra -

3.4.1.2. b) SIMD-like execution-1
Massive multithreading is based on single instruction multiple data (SIMD) execution, in line with the assumed platform model.

2.37. ábra - The assumed platform model [15]

3.4.1.3. Principle of SIMD execution

In case of the SIMD execution the PEs operate in conjunction with a SIMD register file that has a small number of long registers, e.g. 16 - 128 or 256 bit long - registers.

For executing a SIMD instruction the unit will fetch the referenced source operands (FX or FP data vectors) from a common register file (the SIMD register file), the PEs perform the required operation and the result vector will be written back into the given destination address of the SIMD register file, as indicated in the Figure below.

2.38. ábra - Principle of operation of SIMD execution
3.4.1.4. b) SIMD-like execution-2

By contrast, in case of SIMT processing, each thread is allocated a particular RF (per thread RF). A kernel instruction will be executed in SIMD fashion on all PEs available for the execution of the given wavefront.

For each kernel instruction the operands are fetched from the actual Per thread Register Files (RF), the same operation is performed by all PEs available for the execution of the wavefront and the results are written back into the actual Per thread Register Files (RF).

2.39. ábra - Principle of the SIMT execution

3.4.1.5. c) Zero penalty multithreading

If each thread has a private register file for maintaining its actual data space called context, a context switch needed to execute another wavefront can be performed simply by modifying the pointer that identifies the actual register file, as shown in the next Figure.

In this way a context switch does not cause any cycle penalty.

2.40. ábra - Principle of the SIMT execution
3.4.1.6. Contrasting key features of SIMD and SIMT execution

2.41. ábra - Contrasting key features of SIMD and SIMT execution

3.4.2. The concept of assigning work to the PEs

2.42. ábra - Key abstractions of the execution model

It is typically a four step process:

- a) Segmenting the domain of execution to work-groups by the programmer
- b) Assigning work-groups to CUs by the global scheduler of the GPGPU for execution
• c) Segmenting work-groups into wavefronts (warps) by the global scheduler execution pipelines of the SIMT cores

• d) Scheduling wavefronts (warps) to Processing Elements (PEs) (warps) for execution by the scheduler of the PE.

Steps a) to c) belong to the concept of assigning work to the PEs of the GPGPU, nevertheless step d) is implementation dependent, i.e. it will be performed in different GPGPU families differently, according to the actual implementation of the PEs and does not belong to the model of computation at the virtual machine level.

3.4.2.1. a) Segmenting the domain of execution to work-groups-1

The domain of execution will be broken down into equal sized ranges, called work-groups (thread blocks), i.e. into units of work that will be allocated to the CUs as entities.

2.43. ábra - Segmenting the domain of execution to work-groups

![Diagram of domain of execution segmented into work-groups](image)

E.g. Segmenting a 512 x 512 sized domain of execution into four 256 x 256 sized work-groups.

3.4.2.1.1. Segmenting the domain of execution to work-groups-2

2.44. ábra -

![Diagram of domain of execution segmented into work-groups](image)

• Work-groups may be executed in parallel on available CUs.

• The kind of how a given domain of execution will be segmented to work-groups is implementation specific, it can be done either by the programmer or the HLL compiler.

3.4.2.2. b) Assigning work-groups to CUs for execution
Work-groups will be assigned for execution to the available CUs as entities by the scheduler of the GPGPU/DPA.

3.4.2.2.1. Example: Assigning work-groups to CUs in AMD’s Cayman GPGPUs (Northern Islands family) [16]

2.45. ábra -

![Diagram showing work-groups assigned to CUs](image)

Work-groups will be assigned for execution to the same or to different CU.

3.4.2.2.2. Serial/concurrent kernel processing on the CUs

This is an implementation dependent feature of execution and does not belong to the model of computation defined at the virtual machine level, nevertheless due to its implication on the efficiency of processing we briefly discuss it.

2.46. ábra -

![Diagram showing serial/concurrent kernel processing](image)

The GPGPU scheduler assigns work-groups only from a single kernel to the available CUs, i.e. the scheduler distributes work-groups to available CUs for maximum parallel execution.

Examples

- Nvidia’s pre-Fermi GPGPUs
- AMD’s pre-Northern Islands GPGPUs

3.4.2.2.3. Example 1: Serial/concurrent kernel processing in Nvidia’s GPGPUs [25], [18]

- A global scheduler, called the Gigathread scheduler assigns work to each CU.
• In Nvidia’s pre-Fermi GPGPU generations (G80-, G92-, GT200-based GPGPUs) the global scheduler could only assign work-groups to the CUs from a single kernel (serial kernel execution).

• By contrast, in Fermi-based and later GPGPUs the global scheduler is able to run up to 16 different kernels concurrently, presumable, one per CU (concurrent kernel execution).

2.47. ábra -

3.4.2.2.4. Example 2: Serial/concurrent kernel processing in AMD’s GPGPUs

• In GPGPUs preceding Cayman-based systems (Northern Island family) (2010), only a single kernel was allowed to run on a GPGPU.

  In these systems, the work allocation units constituting the NDRange (domain of execution) were spread over all available CUs in order to speed up execution.

• In Cayman based systems (Northern Islands family) (2010) and later families multiple kernels may run on the same GPGPU simultaneously, each one on a single or multiple CUs, allowing a better utilization of the hardware resources for a more parallel execution.

3.4.2.2.5. Example 2: Assigning multiple kernels concurrently to the CUs in Cayman-based cores (Northern Island family)

2.48. ábra -
3.4.2.3. c) Segmenting work-groups into wavefronts (warps)

For execution the scheduler of the GPGPU segments work-groups to wavefronts. The wavefronts will then be scheduled for execution on the allocated CU as an entity.

Segmenting work-groups into wavefronts is considered as part of the computational model, nevertheless, both the size of the wavelengths and the way of segmenting is implementation dependent and thus do not belong to the computational model discussed.

As far as the size of wavefronts is concerned we refer to Section 2.3 stating that:

- In Nvidia’s GPGPUs the wavefronts (called warps) include 32 work-items.
- AMD’s GPGPUs have different wavefront sizes;
  - High performance GPGPU cards, like those of the Southern Island lines (HD 7950/70 cards) have typically wavefront sizes of 64, whereas
  - lower performance cards may have wavefront sizes of 32, 24 or even 16 work items.

3.4.3. The concept of data sharing

2.49. ábra - Key abstractions of the execution model

- The concept of data sharing declares how threads may communicate data to each other.
• This is not an orthogonal concept, but results from the declared memory concept, as indicated in the next example.

3.4.3.1. Example: The model of data sharing in Nvidia’s PTX

(Here we consider only key elements of the data space, based on [13])

2.50. ábra -

Notes

1. Work-groups are designated in the Figure as Blocks

2. The Constant Memory is not shown.
   It has the same data sharing scheme but provides only Read only accessibility.

3.4.4. The concept of data dependent flow control

2.51. ábra -
In SIMT processing all threads have to perform the same operation, nevertheless it is also feasible to mask the execution of particular threads.

It follows that a branch will be executed such that both paths of a branch will be executed in succession on all elements of a wavefront, nevertheless, on each path only those operations will be performed which fulfill the specified data condition for that path (e.g. $x_i > 0$).

This will be demonstrated by the following example.

### 3.4.4.1. Example for executing a data dependent branch-1 [26]

#### 2.52. ábra - Execution of branches-1 [26]

### 3.4.4.2. Example for executing a data dependent branch-2 [26]

#### 2.53. ábra - Execution of branches-2 [26]
First all ALUs meeting the condition execute the prescribed three operations, then all ALUs missing the condition execute the next two operations.

3.4.4.3. Example for executing a data dependent branch-3 [26]

2.54. ábra - Resuming instruction stream processing after executing a branch [26]

3.4.5. The concept of synchronization

2.55. ábra - Key abstractions of the execution model
The virtual machine concept of GPGPU computing

GPGPUs make use of the barrier synchronization due to their wavefront based principle of execution.

Barrier synchronization may however, refer either to the synchronization of the thread execution or to the synchronization of memory read/writes, as shown below.

2.56. ábra -

Synchronization of thread execution

3.4.5.1. Barrier synchronization of thread execution

It allows to synchronize wavefronts in a work-group such that in a work-group all work-items must reach the synchronization point (marked by the barrier synchronization instruction) before execution proceeds.

It is implemented

- in Nvidia’s PTX by the “bar” instruction [27] or
- in AMD’s IL by the “fence thread” instruction [28].

3.4.5.2. Barrier synchronization of memory read/writes

- It ensures that no read/write instructions can be re-ordered or moved across the memory barrier instruction in the specified data space (Local Data Space/Global memory/System memory).

- Thread execution resumes when all prior memory writes of the threads have been completed and thus the data became visible to other threads in the specified data space.

It is implemented

- in Nvidia’s PTX by the “membar” instruction [27] or
- in AMD’s IL by the “fence lds”/“fence memory” instructions [28].

4. The Pseudo ISA of GPGPU computing

- The pseudo ISA of GPGPU computing specifies the instruction set available at the virtual machine level.

- The pseudo ISA evolves in line width the real ISA in form of subsequent releases.

- The evolution comprises both the enhancement of the qualitative (functional) and the quantitative features of the pseudo architecture.

There are two pseudo ISA families associated with the two leading firms developing GPGPU technology, as indicated below.
The virtual machine concept of GPGPU computing

2.57. ábra -

*PTX*

The pseudo ISA of Nvidia’s Parallel Thread execution virtual machine

The pseudo ISAs evolve dynamically following the evolution of both the HLL software environment and the underlying hardware microarchitectures.

Their evolution can be segmented into major releases associated to particular GPGPU families, as indicated below.

2.58. ábra -

*Pseudo ISA’s of G*

*PTX*

The pseudo ISA of Nvidia’s Parallel Thread execution virtual machine

*PTX 1.x*

Pre-Fermi implementations

(Before 2010)

*PTX 2.x*

Fermi implementations

(2010)

*PTX 3.x*

Kepler implementations

(2012)

Remark

Beyond the two vendor specific pseudo ISA’s considered so far the organization responsible for OpenCL (Khronos Group) introduced in 2012 a proposal for a standard (vendor independent) intermediate representation of OpenCL programs, called the OpenCL Standard Portable Intermediate Representation (SPIR).

SPIR is based on OpenCL 1.2 and LLVM 3.1 [29].

(LLVM is a widely used language independent intermediate format for compilers.)

At the time being it is not foreseeable how far both major vendors leading the GPGPU technology will accept SPIR.

4.1. Discussion of pseudo ISAs

Considering the fact that pseudo ISAs relate to a particular GPGPU implementation, it has more sense to give a case example of a pseudo ISA than try to provide a vendor agnostic general description.

As Nvidia’s PTX has a more concise and lucid documentation than AMD’s IL, subsequently, we give a brief overview of pseudo ISAs by the case example of Nvidia’s PTX focusing only on the evolution of its major features.

4.2. Case example: The pseudo ISA of Nvidia’s PTX virtual machine
Often abbreviated as PTX pseudo ISA.

4.3. Case example: The pseudo ISA of Nvidia’s PTX virtual machine

It is a pseudo ISA for GPGPUs that

• is close to the “metal” (i.e. to the actual ISA of GPGPUs) and
• serves as the hardware independent target code format for CUDA, OpenCL etc. compilers.

The PTX virtual machine concept gives rise to a two phase compilation process.

1) First, the application, e.g. a CUDA or OpenCL program will be compiled to a pseudo code, called also as the PTX ISA code or PTX code by the appropriate compiler.

The PTX code is a pseudo code since it is not directly executable and needs to be translated to the actual ISA of a given GPGPU to become executable.
2) In order to become executable the PTX code needs to be compiled to the actual ISA code of a particular GPGPU, called the CUBIN file.

This compilation is performed by the CUDA driver during loading the program (Just-In-Time).

2.61. ábra -

4.4. Nvidia’s compute capability concept [31], [1]

- Nvidia addresses the evolution of their devices and programming environment by maintaining compute capability versions of both
The virtual machine concept of GPGPU computing

- their intermediate language versions (PTX versions) and
- their real architectures (GPGPU ISA).

Designation of the compute capability versions

- Subsequent versions of GPGPU ISAs are designated as sm_1x/sm_2x or sm_3x.
- The first digit 1, 2 or 3 denotes the major version number, the second digit denotes the minor version.
- Major versions of 1.x (or 1x) relate to pre-Fermi solutions whereas those of 2.x (or 2x) to Fermi based solutions and those of 3.x (or 3x) to Kepler based implementations.

Remarks [1]

Correspondence of the pseudo PTX ISA and real GPGPU ISA compute capability versions

Until now there is a one-to-one correspondence between the pseudo PTX ISA versions and the real GPGPU ISA versions, i.e. the PTX ISA versions and the GPGPU ISA versions with the same major and minor version number have the same compute capability.

However, there is no guarantee that this one-to-one correspondence will remain valid in the future.

Main facts concerning the compute capability versions are summarized in the subsequent Tables.

4.5. a) Evolution of the functional features, called compute capabilities in subsequent versions of Nvidia’s pseudo ISA (PTX) [32]

2.62. ábra -

<table>
<thead>
<tr>
<th>Feature support (unlisted features are supported for all compute capabilities)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer atomic functions operating on 32-bit words in global memory</td>
</tr>
<tr>
<td>atomicExch() operating on 32-bit floating point values in global memory</td>
</tr>
<tr>
<td>Integer atomic functions operating on 32-bit words in shared memory</td>
</tr>
<tr>
<td>atomicExch() operating on 32-bit floating point values in shared memory</td>
</tr>
<tr>
<td>Integer atomic functions operating on 64-bit words in global memory</td>
</tr>
<tr>
<td>Warp vote functions</td>
</tr>
<tr>
<td>Double-precision floating-point operations</td>
</tr>
<tr>
<td>Atomic functions operating on 64-bit integer values in shared memory</td>
</tr>
<tr>
<td>Floating-point atomic addition operating on 32-bit words in global and shared memory</td>
</tr>
<tr>
<td>__ballot()</td>
</tr>
<tr>
<td>__threadfence_system()</td>
</tr>
<tr>
<td>__syncthreads_count(), __syncthreads_and(), __syncthreads_or()</td>
</tr>
<tr>
<td>Surface functions</td>
</tr>
<tr>
<td>3D grid of thread block</td>
</tr>
<tr>
<td>Funnel shift</td>
</tr>
</tbody>
</table>

4.6. b) Evolution of the device parameters bound to subsequent compute capability versions of Nvidia’s PTX [32]
2.63. ábra -

<table>
<thead>
<tr>
<th>Technical specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum dimensionality of grid of thread blocks</td>
</tr>
<tr>
<td>Maximum x-, y-, or z-dimension of a grid of thread blocks</td>
</tr>
<tr>
<td>Maximum dimensionality of thread block</td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a block</td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
</tr>
<tr>
<td>Warp size</td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
</tr>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
</tr>
<tr>
<td>Maximum number of 32-bit registers per thread</td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
</tr>
<tr>
<td>Number of shared memory banks</td>
</tr>
<tr>
<td>Amount of local memory per thread</td>
</tr>
<tr>
<td>Constant memory size</td>
</tr>
<tr>
<td>Cache working set per multiprocessor for constant memory</td>
</tr>
</tbody>
</table>

4.7. c) Architecture specifications bound to compute capability versions of Nvidia’s PTX [32]

2.64. ábra -

<table>
<thead>
<tr>
<th>Architecture specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of FX32/FP32 EUs</td>
</tr>
<tr>
<td>Number of special function units for single-precision floating-point transcendental functions</td>
</tr>
<tr>
<td>Number of texture filtering units for every texture address unit or render output unit (ROP)</td>
</tr>
<tr>
<td>Number of warp schedulers</td>
</tr>
<tr>
<td>Number of instructions issued at once by scheduler</td>
</tr>
</tbody>
</table>

4.8. d) Throughput of native arithmetic instructions in subsequent compute capability versions of Nvidia’s PTX (operations per clock cycle/SM) [7]

2.65. ábra -
Given CUDA (and OpenCL) releases generate PTX code of a specified PTX version (PTXu.v) that supports a given compute capability version (sm.x.y), as indicated in the next Table.

### 4.9. PTX ISA Versions generated by subsequent releases of CUDA SDKs and supported compute capability versions (sm[xx]) (designated as Supported Targets in the Table) [20]

<table>
<thead>
<tr>
<th>PTX ISA Version</th>
<th>CUDA Release</th>
<th>Supported Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTX ISA 1.0</td>
<td>CUDA 1.0</td>
<td>sm_{10,11}</td>
</tr>
<tr>
<td>PTX ISA 1.1</td>
<td>CUDA 1.1</td>
<td>sm_{10,11}</td>
</tr>
<tr>
<td>PTX ISA 1.2</td>
<td>CUDA 2.0</td>
<td>sm_{10,11,12,1}</td>
</tr>
<tr>
<td>PTX ISA 1.3</td>
<td>CUDA 2.1</td>
<td>sm_{10,11,12,1}</td>
</tr>
<tr>
<td>PTX ISA 1.4</td>
<td>CUDA 2.2</td>
<td>sm_{10,11,12,1}</td>
</tr>
<tr>
<td>PTX ISA 1.5</td>
<td>driver r190</td>
<td>sm_{10,11,12,1}</td>
</tr>
<tr>
<td>PTX ISA 2.0</td>
<td>CUDA 3.0, driver r195</td>
<td>sm_{10,11,12,1}</td>
</tr>
<tr>
<td>PTX ISA 2.1</td>
<td>CUDA 3.1, driver r256</td>
<td>sm_{10,11,12,1}</td>
</tr>
<tr>
<td>PTX ISA 2.2</td>
<td>CUDA 3.2, driver r260</td>
<td>sm_{10,11,12,1}</td>
</tr>
<tr>
<td>PTX ISA 2.3</td>
<td>CUDA 4.0, driver r270</td>
<td>sm_{10,11,12,1}</td>
</tr>
<tr>
<td>PTX ISA 3.0</td>
<td>CUDA 4.1, driver r285</td>
<td>sm_{10,11,12,1}</td>
</tr>
<tr>
<td>PTX ISA 3.1</td>
<td>CUDA 4.2, driver r295</td>
<td>sm_{10,11,12,1}</td>
</tr>
<tr>
<td>PTX ISA 3.1</td>
<td>CUDA 5.0, driver r302</td>
<td>sm_{10,11,12,1}</td>
</tr>
</tbody>
</table>
The virtual machine concept of
GPGPU computing

The discussed compute capability versions reveal the compute capabilities of Nvidia’s GPGPU cores and cards, as shown below.

4.10. Compute capability versions supported by Nvidia’s GPGPU cores and cards [32]

2.67. ábra -

<table>
<thead>
<tr>
<th>Compute capability versions supported by Nvidia’s GPGPU cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capability vers.</td>
</tr>
<tr>
<td>(sm_xy)</td>
</tr>
<tr>
<td>1.0</td>
</tr>
<tr>
<td>1.1</td>
</tr>
<tr>
<td>1.2</td>
</tr>
<tr>
<td>1.3</td>
</tr>
<tr>
<td>2.0</td>
</tr>
<tr>
<td>2.1</td>
</tr>
<tr>
<td>3.0</td>
</tr>
<tr>
<td>3.5</td>
</tr>
</tbody>
</table>

4.11. Forward portability of PTX code [31]

Applications compiled for pre-Fermi GPGPUs that include PTX versions of their kernels should work as-is on Fermi GPGPUs as well.

4.12. Compatibility rules of object files (CUBIN files) compiled to a particular GPGPU compute capability version [31]

The basic rule is forward compatibility within the main versions (e.g. within versions sm_1x or sm_2x) but not across main versions.

This is interpreted as follows:

Object files (called CUBIN files) compiled to a particular GPGPU compute capability version are supported on all devices having the same or higher subversion number within the same main version.

E.g. object files compiled to the compute capability 1.0 are supported on all 1.x devices but not supported on compute capability 2.0 (Fermi) devices.

For more details see [31].
3. fejezet - Overview of GPGPU cores and cards

1. General overview

3.1. ábra - Basic implementation alternatives of the SIMT execution

1.1. GPGPU cores

Both GPGPU cards and data parallel accelerators are based on GPGPU cores (chips).

3.2. ábra - Overview of Nvidia’s and AMD/ATI’s GPGPU lines
Overview of GPGPU cores and cards

3.3. ábra - Overview of GPGPU cores and cards and their basic software support (1)

3.4. ábra - Overview of GPGPU cores and cards and their basic software support (2)
Remarks

1) beginning with their Northern Islands line (HD 6xxx) AMD has changed the branding of their graphics cards as follows:

- until the Evergreen family (HD 5xxx) AMD made use of the ATI HD xxxx branding,
but starting with the Northern Island family AMD changed their branding to AMD HD xxxx.

2) The above overview does not include dual-GPU graphics cards.

These are surveyed in the next two Tables.

### 1.2. Main features of Nvidia’s dual-GPU graphics cards

3.6. ábra - Main features of Nvidia’s dual-GPU graphics cards [33], [34]

<table>
<thead>
<tr>
<th>Model</th>
<th>Intro.</th>
<th>Dubbed as</th>
<th>No. of ALUs</th>
<th>PCIe support</th>
</tr>
</thead>
<tbody>
<tr>
<td>7900 GX2</td>
<td>3/2006</td>
<td></td>
<td>2x24</td>
<td>PCIe 1.0</td>
</tr>
<tr>
<td>7950 GX2</td>
<td>6/2006</td>
<td></td>
<td>2x24</td>
<td></td>
</tr>
<tr>
<td>9800 GX2</td>
<td>3/2008</td>
<td></td>
<td>2x128</td>
<td></td>
</tr>
<tr>
<td>GTX 295</td>
<td>1/2009</td>
<td>5/2009</td>
<td>2x240</td>
<td>PCIe 2.0</td>
</tr>
<tr>
<td>GTX 590</td>
<td>3/2011</td>
<td>Fermi</td>
<td>2x512</td>
<td></td>
</tr>
<tr>
<td>GTX 690</td>
<td>4/2012</td>
<td>Kepler</td>
<td>2x1536</td>
<td>PCIe 3.0</td>
</tr>
</tbody>
</table>

### 1.3. Main features of AMD’s/ATI’s dual-GPU graphics cards

3.7. ábra - Main features of AMD’s/ATI’s dual-GPU graphics cards [35]

<table>
<thead>
<tr>
<th>ModelID</th>
<th>Introduced</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATI Rage Fury MAXX</td>
<td>10/1999</td>
<td></td>
</tr>
<tr>
<td>Radeon HD 3870 X2</td>
<td>1/2008</td>
<td>2</td>
</tr>
<tr>
<td>Radeon HD 3850 X2</td>
<td>4/2008</td>
<td>2</td>
</tr>
<tr>
<td>Radeon HD 4870 X2</td>
<td>8/2008</td>
<td>2</td>
</tr>
<tr>
<td>Radeon HD 4850 X2</td>
<td>11/2008</td>
<td>2</td>
</tr>
<tr>
<td>Radeon HD 5970</td>
<td>11/2009</td>
<td>2</td>
</tr>
<tr>
<td>Radeon HD 6990</td>
<td>3/2011</td>
<td>2</td>
</tr>
<tr>
<td>Radeon HD 7990</td>
<td>8/2012</td>
<td>2</td>
</tr>
</tbody>
</table>

### 1.4. Remarks on AMD-based graphics cards [36], [37]

Beginning with their Cypress-based HD 5xxx line (Evergreen family) and SDK v.2.0 AMD left Brook+ and started supporting OpenCL as their basic HLL programming language.

3.8. ábra -
As a consequence AMD changed also

- both the microarchitecture of their GPGPUs (by introducing Local and Global Data Share memories) and their terminology by introducing Pre-OpenCL and OpenCL terminology, as discussed

- in Section 5.2.

2. Overview of the main features of Nvidia’s graphics cores and cards

2.1. Main features of Nvidia’s graphics cores and cards

3.9. ábra - Main features of Nvidia’s pre-Fermi graphics cards [33]

<table>
<thead>
<tr>
<th>Core</th>
<th>8800 GTS</th>
<th>8800 GTX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>11/06</td>
<td>11/06</td>
</tr>
<tr>
<td>IC technology</td>
<td>99 nm</td>
<td>96 nm</td>
</tr>
<tr>
<td>Nr. of transistors</td>
<td>681 mns</td>
<td>683 mns</td>
</tr>
<tr>
<td>Die area</td>
<td>446 mm²</td>
<td>446 mm²</td>
</tr>
<tr>
<td>Core frequency</td>
<td>506 MHz</td>
<td>575 MHz</td>
</tr>
<tr>
<td>Compute unit</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>No of FP32 EU</td>
<td>96</td>
<td>128</td>
</tr>
<tr>
<td>Shader frequency</td>
<td>1.2 GHz</td>
<td>1.31 GHz</td>
</tr>
<tr>
<td>No. FP32 operations/cycle</td>
<td>230.4 GFLOPS</td>
<td>345.6 GFLOPS</td>
</tr>
<tr>
<td>Peak FP64 performance</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Peak FP64 performance</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem. transfer rate (off)</td>
<td>1600 MB/s</td>
<td>1800 MB/s</td>
</tr>
<tr>
<td>Mem. interface</td>
<td>320-bit</td>
<td>384-bit</td>
</tr>
<tr>
<td>Mem. bandwidth</td>
<td>64 GB/s</td>
<td>64 GB/s</td>
</tr>
<tr>
<td>Mem. size</td>
<td>32 MB</td>
<td>768 MB</td>
</tr>
<tr>
<td>Mem. type</td>
<td>GDDR3</td>
<td>GDDR5</td>
</tr>
<tr>
<td>Mem. channel</td>
<td>64×4-bit</td>
<td>64×4-bit</td>
</tr>
<tr>
<td>System</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-CPUs/second</td>
<td>4 SL</td>
<td>4 SL</td>
</tr>
<tr>
<td>Interface</td>
<td>PCIe ×16</td>
<td>PCIe ×16</td>
</tr>
<tr>
<td>MS Direct X</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>TDP</td>
<td>146 W</td>
<td>155 W</td>
</tr>
</tbody>
</table>

1: Nvidia takes the FP32 capable Texture Processing Units also in

Remarks

In publications there are conflicting statements about whether or not the GT80 makes use of dual issue (including a MAD and a Mul operation) within a period of four shader cycles or not.
Overview of GPGPU cores and cards

Official specifications [6] declare the capability of dual issue, but other literature sources [38] and even a textbook, co-authored by one of the chief developers of the GT80 (D. Kirk [39]) deny it.

A clarification could be found in a blog [37], revealing that the higher figure given in Nvidia’s specifications includes calculations made both by the ALUs in the SMs and by the texture processing units TPU).

3.10. ábra - Interpretation of the MAD (Multiply-Add) operation [51]

2.1.1. Structure of an SM of the G80 architecture

3.11. ábra -

Texture processing Units consisting of

- TA: Texture Address units
• TF: Texture Filter Units

They are FP32 or FP16 capable [40]

Nevertheless, the TPUs can not be directly accessed by CUDA except for graphical tasks, such as texture filtering.

Accordingly, in our discussion focusing on numerical calculations it is fair to take only the MAD operations into account for specifying the peak numerical performance.

3.12. ábra - Main features of Nvidia’s Fermi-based graphics cards [33]

<table>
<thead>
<tr>
<th>Feature</th>
<th>GTX 470</th>
<th>GTX 480</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>GF100</td>
<td>GF100</td>
</tr>
<tr>
<td>IC technology</td>
<td>40 nm</td>
<td>40 nm</td>
</tr>
<tr>
<td>Nr. of transistors</td>
<td>3200 m^2</td>
<td>3200 m^2</td>
</tr>
<tr>
<td>Die area</td>
<td>528 m^2</td>
<td>528 m^2</td>
</tr>
<tr>
<td>Core frequency</td>
<td>60 MHz</td>
<td>70 MHz</td>
</tr>
</tbody>
</table>

**Computation**

- No of SMs (cores): 14, 15
- No of FP32 EUs: 148, 160
- Shader frequency: 121 MHz, 140 MHz
- No FP32 operations/cycle: 2, 2
- Peak FP32 performance: 1088 GFLOPS, 1344 GFLOPS
- Peak FP64 performance: 336 GFLOPS, 416 GFLOPS

**Memory**

- Mem. transfer rate (off): 334.4 MB/s, 6091 MB/s
- Mem. interface: 320-bit, 384-bit
- Mem. bandwidth: 133.9 GB/s, 177.4 GB/s
- Mem. size: 1.28 GB, 1.516 GB
- Mem. type: GDDR5, GDDR5
- Mem. channel: 5*64-bit, 6*64-bit

**System**

- Max. CPU/sec: 512, 512
- Interface: PCI 2.0x16, PCI 2.0x16
- MX Direct X: 11, 11
- TDP: 215 W, 250 W

Remarks on Fermi-based graphics cards [36], [37]

FP64 speed

- 1/2 of the FP32 speed for the Tesla 20-series
- 1/8 of the SP32 speed for the GeForce GTX 470/480/570/580 cards
- 1/12 for other GForce GTX4xx cards

ECC

available only on the Tesla 20-series

Number of DMA engines

Tesla 20-series has 2 DMA Engines (copy engines). GeForce cards have 1 DMA Engine.

This means that CUDA applications can overlap computation and communication on Tesla using bi-directional communication over PCI-e.

Memory size

Tesla 20 products have larger on board memory (3GB and 6GB)
Remarks on GDDR memories

1) The GDDR3 memory has a double clocked data transfer
   Effective memory transfer rate = 2 x memory frequency
The GDDR5 memory has a quad clocked data transfer
   Effective memory transfer rate = 4 x memory frequency

2) Both the GDDR3 and GDDR5 memories are 32-bit devices.

Nevertheless, memory controllers of GPGPUs may be designed either to control a single 32-bit memory channel or dual memory channels, providing a 64-bit channel width.

3.13. ábra - Main features of Nvidia’s Kepler-based graphics cards [33]
### Core
- Introduction
- IC technology
- No. of transistors
- Die area

### Computation
- No of SMXs (cores)
- No. of FP32 EUs
- Shader frequency
- No. FP32 operations/cycle
- Peak FP32 performance
- Peak FP64 performance

### Memory
- Mem. transfer rate (eff)
- Mem. interface
- Mem. bandwidth
- Mem. size
- Mem. type
- Mem. channel

### System
- Multi. CPU techn.
- Interface
- MS Direct X
- TDP

**2.1.2. Positioning Nvidia’s GPGPU cards in their entire product portfolio [41]**

3.14. ábra -
2.2. Examples for Nvidia’s graphics cards

2.2.1. Nvidia GeForce GTX 480 (GF 100 based) [42]

3.15. ábra -

2.2.2. A pair of Nvidia’s GeForce GTX 480 cards [42] (GF100 based)
3.16. ábra -

2.2.3. Nvidia’s GeForce GTX 480 and 580 cards [44]

3.17. ábra -
2.2.4. Nvidia’s GTX 680 card [80] (GK104 based)

3.18. ábra -

2.2.5. Nvidia’s GTX Titan card [81] (GK110 based)

3.19. ábra -
3. Overview of the main features of AMD’s graphics cores and cards

3.1. Main features of AMD’s graphics cores and cards

3.20. ábra - Main features of AMD/ATI’s early graphics cards-1 [35]
Overview of GPGPU cores and cards

3.21. ábra - Main features of AMD/ATI’s early graphics cards-2 [35]

<table>
<thead>
<tr>
<th>Evergreen series</th>
<th>HD 5870</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>CypressPR (RV)</td>
</tr>
<tr>
<td>Introduction</td>
<td>9/09</td>
</tr>
<tr>
<td>IC technology</td>
<td>40 nm</td>
</tr>
<tr>
<td>Nr. of transistors</td>
<td>2154 Mnt</td>
</tr>
<tr>
<td>Die are</td>
<td>334 mm</td>
</tr>
<tr>
<td>Core frequency</td>
<td>725 MHz</td>
</tr>
<tr>
<td>Computation</td>
<td></td>
</tr>
<tr>
<td>No. of SIMD cores/ VLIW 5 ALUs</td>
<td>18:16</td>
</tr>
<tr>
<td>No. of ELUs</td>
<td>1440</td>
</tr>
<tr>
<td>Shader frequency</td>
<td>725 MHz</td>
</tr>
<tr>
<td>No. FP32 inst./cycle</td>
<td>1</td>
</tr>
<tr>
<td>Peak FP32 performance</td>
<td>2050 GFLOP</td>
</tr>
<tr>
<td>Peak FP64 performance</td>
<td>417.6 GFLOP</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>Mem. transfer rate (eff)</td>
<td>4000 MB</td>
</tr>
<tr>
<td>Mem. interface</td>
<td>256-bit</td>
</tr>
<tr>
<td>Mem. bandwidth</td>
<td>128 JB</td>
</tr>
<tr>
<td>Mem. size</td>
<td>1.0 GB</td>
</tr>
<tr>
<td>Mem. type</td>
<td>GDDR3</td>
</tr>
<tr>
<td>Mem. channel</td>
<td>8*12-bit</td>
</tr>
<tr>
<td>System</td>
<td></td>
</tr>
<tr>
<td>Multi. CPU tec.</td>
<td>CrossFire</td>
</tr>
<tr>
<td>Interface</td>
<td>PCIe x16</td>
</tr>
<tr>
<td>MS Direct X</td>
<td>11</td>
</tr>
<tr>
<td>TDP Max./Idle</td>
<td>115 W / 75 W</td>
</tr>
</tbody>
</table>

3.22. ábra - Main features of AMD’s Southern Islands series graphics cards [35]
### Northern Islands series

<table>
<thead>
<tr>
<th>Core</th>
<th>Introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC technology</td>
<td></td>
</tr>
<tr>
<td>Nr. of transistors</td>
<td></td>
</tr>
<tr>
<td>Die area</td>
<td></td>
</tr>
<tr>
<td>Core frequency</td>
<td></td>
</tr>
</tbody>
</table>

#### Computation

- No. of SIMD cores /VLIW 8 A1
- No. of EUs
- Shader frequency
- No. FP32 inst./ cycle
- Peak FP32 performance
- Peak FP64 performance

#### Memory

- Mem. transfer rate (eff)
- Mem. interface
- Mem. bandwidth
- Mem. size
- Mem. type
- Mem. channel

#### System

- Multi. CPU techn.
- Interface
- MS Direct X
- TDP Max./Idle

---

3.23. ábra - Main features of AMD’s Northern Islands series graphics cards [35]
Overview of GPGPU cores and cards

<table>
<thead>
<tr>
<th>Northern Islands series</th>
<th>HD 6950</th>
<th>HD 7950</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>Cayman Pro</td>
<td>Cayman</td>
</tr>
<tr>
<td>Introduction</td>
<td>1210</td>
<td>1210</td>
</tr>
<tr>
<td>IC technology</td>
<td>40 nm</td>
<td>40 nm</td>
</tr>
<tr>
<td>Nr. of transistors</td>
<td>2.64 billion</td>
<td>2.64 billion</td>
</tr>
<tr>
<td>Die area</td>
<td>389 mm²</td>
<td>388 mm²</td>
</tr>
<tr>
<td>Core frequency</td>
<td>850 MHz</td>
<td>880 MHz</td>
</tr>
</tbody>
</table>

**Computations**

- No. of SIMD cores/VLIW/ALUs: 22/16/2
- No. of EUs: 1488
- Shader frequency: 800 MHz
- No. FP/2 operations/cycle / ALU: 4
- Peak FP64 performance: 2.91 TFLOPS
- Peak FP32 performance: 6.16 TFLOPS

**Memory**

- Mem. transfer rate (eff): 5900 MB/s
- Mem. interface: 256-bit
- Mem. bandwidth: 160 GB/s
- Mem. size: 2 GB
- Mem. type: DDR5
- Mem. channel: 19.2-bit

**System**

- ECC: -
- Multi CPU/tecin.: CrossFireX
- Interface: PCIe 2.1, x16
- MS Direct X: 11
- TDP Max./All: 200/20 W

**3.24. ábra - Main features of AMD’s Southern Islands series (GCN) graphics cards [35]**

| Southern Islands series (GCN) | HD 7950 | |
|-------------------------------|---------| |
| Core                          | Tahiti 7900 |
| Introduction                  | 01/32   |
| IC technology                 | 28 nm   |
| Nr. of transistors            | 4,313 billion |
| Die area                      | 312 mm² |
| Core frequency                | 800 MHz |

**Computations**

- No. of CUs x No. of SIMDs x No. of EUs: 28 CU x 16 SIMD x 16 EU / 32 CU x 24 SIMD x 24 EU
- No. of EUs: 1792
- Shader frequency: 800 MHz
- No. FP/2 operations/cycle / EU: 2
- Peak FP32 performance: 2997 TFLOPS
- Peak FP64 performance: 0.57 TFLOPS

**Memory**

- Mem. transfer rate (eff): 5900 MB/s
- Mem. interface: 384-bit
- Mem. bandwidth: 240 GB/s
- Mem. size: 3 GB
- Mem. type: DDR5
- Mem. channel: 128-bit

**System**

- ECC: -
- Multi CPU/tecin.: CrossFireX
- Interface: PCIe 3.0 x16
- TDP Max./All: 200 W

**3.25. ábra - Main features of AMD’s Sea Islands series graphics cards [45]**
### Sea Islands series

<table>
<thead>
<tr>
<th>Core</th>
<th>Introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>iC technology</td>
<td></td>
</tr>
<tr>
<td>Nr. of transistors</td>
<td></td>
</tr>
<tr>
<td>Die area</td>
<td></td>
</tr>
<tr>
<td>Core frequency</td>
<td></td>
</tr>
</tbody>
</table>

#### Computation

<table>
<thead>
<tr>
<th>No. CUs x No. of SIMDs x No.</th>
<th>No. of EUs</th>
<th>Shader frequency</th>
<th>No. FP32 inst./cycle</th>
<th>Peak FP32 performance</th>
<th>Peak FP64 performance</th>
</tr>
</thead>
</table>

#### Memory

<table>
<thead>
<tr>
<th>Mem. transfer rate (eff)</th>
<th>Mem. interface</th>
<th>Mem. bandwidth</th>
<th>Mem. size</th>
<th>Mem. type</th>
<th>Mem. channel</th>
</tr>
</thead>
</table>

#### System

<table>
<thead>
<tr>
<th>Multi. CPU tech.</th>
<th>Interface</th>
<th>MS Direct N</th>
<th>TDP Max/Idle</th>
<th></th>
</tr>
</thead>
</table>

### 3.2. Examples for AMD’s graphics cards

3.26. ábra - ATI HD 5870 (RV870 based) [43]
3.2.1. ATI HD 5970 (actually RV870 based) [46]

3.27. ábra - ATI HD 5970: 2 x ATI HD 5870 with slightly reduced memory clock

3.2.2. ATI HD 5970 (RV870 based) [47]

3.28. ábra - ATI HD 5970: 2 x ATI HD 5870 with slightly reduced memory clock
3.2.3. AMD HD 6990 (Cayman based) [48]

3.29. ábra - AMD HD 6990: 2 x ATI HD 6970 with slightly reduced memory and shader clock

3.2.4. AMD Radeon HD 7970 (Tahiti XT, Southern Islands based) - First GCN implementation [49]

3.30. ábra - AMD HD 7970
3.2.5. Prices of GPGPUs as of Summer 2012 [50]

3.31. ábra - Summer 2012 GPU Pricing Comparison
### Overview of GPGPU cores and cards

<table>
<thead>
<tr>
<th>AMIL</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Rade</td>
<td>HD 79</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GH</td>
<td>Editor</td>
</tr>
</tbody>
</table>

| Rade | HD 79  |        |

| Rade | HD 78  |        |

| Rade | HD 78  |        |
4. fejezet - Nvidia’s Fermi family of graphics cores and cards

1. Overview of the Fermi family

4.1. ábra - Overview of GPGPU cores and cards and their basic software support (2)


Available: 1Q 2010 [18]
1.1. Major sub-families of Fermi

Fermi includes four sub-families (cores) with the following key features:

<table>
<thead>
<tr>
<th>Core</th>
<th>Available since</th>
<th>Max. no. of SMs (cores)</th>
<th>Max. no. of 32-bit ALUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>GF100</td>
<td>3/2010</td>
<td>16(^1)</td>
<td>512(^1)</td>
</tr>
<tr>
<td>GF104</td>
<td>7/2010</td>
<td>8</td>
<td>336</td>
</tr>
<tr>
<td>GF110</td>
<td>11/2010</td>
<td>16</td>
<td>512</td>
</tr>
<tr>
<td>GF114</td>
<td>1/2011</td>
<td>8</td>
<td>384</td>
</tr>
<tr>
<td>GF114</td>
<td>5/2011</td>
<td>7</td>
<td>336</td>
</tr>
</tbody>
</table>

\(^1\) In the associated flagship card (GTX 480) however, problems, so it has actually only 15 SIMD cores, call FP32 ALUs [62]

2. Key innovations of Fermi’s PTX 2.0

2.1. Overview of PTX 2.0

- Fermi’s underlying pseudo ISA is the 2. generation PTX 2.x (Parallel Thread eXecution) ISA, introduced along with the Fermi line.

- PTX 2.x is a major redesign of the PTX 1.x ISA, towards a more RISC-like load/store architecture rather than being an x86 memory based architecture.
2.2. Key innovations of PTX 2.0 pseudo ISA

• a) Unified address space for all variables and pointers with a single set of load/store instructions
• b) 64-bit addressing capability.
• c) New instructions to support OpenCL and DirectCompute APIs.
• d) Full support of predication.
• e) Full IEEE 754-3008 support for 32-bit and 64-bit FP precision.

These new features greatly improve GPU programmability, accuracy and performance.

2.2.1. a) Unified address space for all variables and pointers with a single set of load/store instructions-1 [11]

In PTX 1.0 there are three separate address spaces (thread private local, block shared and global) with specific load/store instructions to each one of the three address spaces. Programs could load or store values in a particular target address space at addresses that become known at compile time.

It was difficult to fully implement C and C++ pointers since a pointer’s target address could only be determined dynamically at run time.

4.4. ábra -

2.2.2. a) Unified address space for all variables and pointers with a single set of load/store instructions-2 [11]

PTX 2.0 unifies all three address spaces into a single continuous address space that can be accessed by a single set of load/store instructions.
PTX 2.0 allows to use unified pointers to pass objects in any memory space and Fermi’s hardware automatically maps pointer references to the correct memory space.

Thus the concept of the unified address space enables Fermi to support C++ programs.

### 4.5. ábra -

**Separate Address Space**

- Nvidia’s previous generation GPGPUs (G80, G92, GT200) provide 32 bit addressing for load/store instructions,

- PTX 2.0 extends the addressing capability to 64-bit for future growth.

however, recent Fermi implementations use only 40-bit addresses allowing to access an address space of 1 Terabyte.

#### 2.2.3. b) 64-bit addressing capability

- Nvidia’s previous generation GPGPUs (G80, G92, GT200) provide 32 bit addressing for load/store instructions,

- PTX 2.0 extends the addressing capability to 64-bit for future growth.

#### 2.2.4. c) New instructions to support OpenCL and DirectCompute APIs

- PTX2.0 is optimized for the OpenCL and DirectCompute programming environments.

- It provides a set of new instructions allowing hardware support for these APIs.

#### 2.2.5. d) Full support of predication [51]

- PTX 2.0 supports predication for all instructions.

- Predicated instructions will be executed or skipped depending on the actual values of conditional codes.

- Predication allows each thread to perform different operations while execution continuous at full speed.

- Predication is a more efficient solution for streaming applications than using conventional conditional branches and branch prediction.
Remark

Principle of predication

• Predication is a program construct to eliminate conditional branches that reduce the efficiency of pipelined execution.

• Predication make use of one or more predication registers.

• The contents of the addressed predication register is set according to whether or not the associated condition is true or not (e.g. there is an overflow etc.).

• The predicated instruction is executed if the predicate is true else not.

2.2.6. e) Full IEEE 754-3008 support for 32-bit and 64-bit FP precision

• Fermi’s FP32 instruction semantics and implementation supports now

  • calculations with subnormal numbers

    (numbers that lie between zero and the smallest normalized number) and

  • all four rounding modes (nearest, zero, positive infinity, negative infinity).

• Fermi provides fused multiply-add (FMA) instructions for both single and double precision FP calculations (with retaining full precision in the intermediate stage) instead of using truncation between the multiplication and addition as done in previous generation GPGPUs for multiply-add instructions (MAD).

2.2.7. Supporting program development for the Fermi line of GPGPUs [11]

• Nvidia provides a development environment, called Nexus, designed specifically to support parallel CUDA C, OpenCL and DirectCompute applications.

• Nexus brings parallel-aware hardware source code debugging and performance analysis directly into Microsoft Visual Studio.

• Nexus allows Visual Studio developers to write and debug GPU source code using exactly the same tools and interfaces that are used when writing and debugging CPU code.

• Furthermore, Nexus extends Visual Studio functionality by offering tools to manage massive parallelism.

3. Major innovations and enhancements of Fermi’s microarchitecture

3.1. Major innovations

a) Concurrent kernel execution

b) True two level cache hierarchy

c) Configurable shared memory/L1 cache per SM

d) ECC support

3.2. Major enhancements

a) Vastly increased FP64 performance

b) Greatly reduced context switching times

c) 10-20 times faster atomic memory operations
3.3. Major architectural innovations of Fermi

3.3.1. a) Concurrent kernel execution [52], [18]

- In previous generations (G80, G92, GT200) the global scheduler could only assign work to the SMs from a single kernel (serial kernel execution).

- The global scheduler of Fermi is able to run up to 16 different kernels concurrently, one per SM.

- A large kernel may be spread over multiple SMs.

4.6. ábra -

3.3.2. b) True two level cache hierarchy [11]

- Traditional GPU architectures support a read-only “load path” for texture operations and a write-only “export path” for pixel data output.

- For computational tasks however, this impedes the ordering of read and write operations done usually for speeding up computations.

- To eliminate this deficiency Fermi implements a unified memory access path for both loads and stores.

- Fermi provides further on a unified L2 cache for speeding up loads, stores and texture requests.

4.7. ábra -
3.3.3. c) Configurable shared memory/L1 cache per SM [11]

- Fermi provides furthermore a configurable shared memory/L1 cache per SM.
• The shared memory/L1 cache unit is configurable to optimally support both shared memory and caching of local and global memory operations.

Supported options are 48 KB shared memory with 16 KB L1 cache or vice versa.

• The optimal configuration depends on the application to be run.

4.8. ábra -

3.3.4. d) ECC support [11]

• It protects
• DRAM memory
• register files
• shared memories
• L1 and L2 caches.

Remark

ECC support is provided only for Tesla devices.

3.4. Major architectural enhancements of Fermi
3.4.1. a) Vastly increased FP64 performance

Compared to the previous G80 and GT200-based generations, Fermi provides vastly increased FP64 performance over flagship GPGPU cards.

3.4.1.1. Flagship Tesla cards

4.9. ábra -

<table>
<thead>
<tr>
<th></th>
<th>C870 (G80-based)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP64 performance</td>
<td>--</td>
</tr>
</tbody>
</table>

3.4.1.2. Flagship GPGPU cards

4.10. ábra -

<table>
<thead>
<tr>
<th></th>
<th>8800 GTX (G80-based)</th>
<th>GTX 280 (GT200-based)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP64 performance</td>
<td>--</td>
<td>77.76 GFLO</td>
</tr>
</tbody>
</table>

3.5. Throughput of arithmetic operations per clock cycle per SM [13]

4.11. ábra -

<table>
<thead>
<tr>
<th></th>
<th>Compute Capability 1.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit floating-point add, multiply, multiply-add</td>
<td>8</td>
</tr>
<tr>
<td>64-bit floating-point add, multiply, multiply-add</td>
<td>1</td>
</tr>
<tr>
<td>32-bit integer add, logical operation</td>
<td>8</td>
</tr>
<tr>
<td>32-bit integer shift, compare</td>
<td>8</td>
</tr>
<tr>
<td>32-bit integer multiply, multiply-add, sum of absolute difference</td>
<td>Multiple instructions</td>
</tr>
<tr>
<td>24-bit integer multiply (___ulmul24)</td>
<td>8</td>
</tr>
<tr>
<td>32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (___log2f), base-2 exponential (exp2f), sine (___sinf), cosine (___cosf)</td>
<td>2</td>
</tr>
<tr>
<td>Type conversions</td>
<td>8</td>
</tr>
</tbody>
</table>

3.5.1. b) Greatly reduced context switching times [11]
Nvidia’s Fermi family of graphics cores and cards

- Fermi performs context switches between different applications in about 25 µs.
- This is about 10 times faster than context switches in the previous GT200 (200-250 µs).

3.5.2. c) 10-20 times faster atomic memory operations [11]

- Atomic operations are widely used in parallel programming to facilitate correct read-modify-write operations on shared data structures.
- Owing to its increased number of atomic units and the L2 cache added, Fermi performs atomic operations up to 20x faster than previous GT200 based devices.

4. Microarchitecture of Fermi GF100

4.1. Overall structure of Fermi GF100 [18], [11]

4.12. ábra -

Fermi:16 Streaming Multiprocessors (SMs)
Each SM: 32 ALUs
512 ALUs

Remark

In the associated flagship card (GTX 480) however, one SM has been disabled, due to overheating problems, so it has actually 15 SMs and 480 ALUs [a]

6x Dual Channel GDDR5 (6x 64 = 384 bit)

4.2. High level microarchitecture of Fermi GT100

4.13. ábra - Fermi’s system architecture [52]
4.3. Evolution of the high level microarchitecture of Nvidia’s GPGPUs [52]

4.14. ábra -
Note

The high level microarchitecture of Fermi evolved from a graphics oriented structure to a computation oriented one complemented with units needed for graphics processing.

4.4. Layout of a Cuda GF100 SM [19]

(SM: Streaming Multiprocessor)

4.15. ábra -

1 SM includes 32 ALUs
called “Cuda cores” by NVidia)

4.5. Evolution of the cores (SMs) in Nvidia’s GPGPUs -1
Nvidia’s Fermi family of graphics cores and cards

GT80 SM [53]

4.16. ábra -

- 16 KB Shared Memory
- 8 K registersx32-bit/SM
- up to 24 active warps/SM
- up to 768 active threads/SM
- 10 registers/SM on average

GT200 SM [54]

4.17. ábra -

- 16 KB Shared Memory

Created by XMLmind XSL-FO Converter.
Nvidia’s Fermi family of graphics cores and cards

- 16 K registersx32-bit/SM
- up to 32 active warps/SM
  - up to 1 K active threads/SM
  - 16 registers/thread on average
- 1 FMA FPU (not shown)

Fermi GT100/GT110 SM [19]

4.18. ábra -

- 64 KB Shared Memory/L1 Cache
- up to 48 active warps/SM
- 32 threads/warp
  - up to 1536 active threads/SM
  - 20 registers/thread on average

4.6. Further evolution of the SMs in Nvidia’s GPGPUs -2

GF104 [55]
Available specifications:

- 64 KB Shared Memory/L1 Cache
- 32 Kx32-bit registers/SM
- 32 threads/warp
- Up to 48 active warps/SM
- Up to 1536 threads/SM

4.19. ábra -

GF100 SM [56]

4.7. Structure and operation of the Fermi GF100 GPGPU

4.7.1. Layout of a Cuda GF100 SM [19]
Nvidia’s Fermi family of graphics cores and cards

4.20. ábra -

1 SM includes 32 ALUs called "CUDA cores" by NVIDIA

4.7.2. A single ALU ("CUDA core")

4.21. ábra - A single ALU [57]

FP64

- First implementation of the IEEE 754-2008 standard
- Needs 2 clock cycles to issue the entire warp for execution.

FP64 performance: ½ of FP32 performance!!
(Enabled only on Tesla devices!)

Remark

The Fermi line supports the Fused Multiply-Add (FMA) operation, rather than the Multiply-Add operation performed in previous generations.
4.22. ábra - Contrasting the Multiply-Add (MAD) and the Fused-Multiply-Add (FMA) operations [51]

4.8. Principle of the SIMT execution in case of serial kernel execution

4.23. ábra - Hierarchy of threads [58]

4.9. Principle of operation of a Fermi GF100 GPGPU

The key point of operation is work scheduling

4.10. Subtasks of work scheduling

• Scheduling kernels to SMs
• Scheduling thread blocks associated with the same kernel to the SMs
• Segmenting thread blocks into warps
• Scheduling warps for execution in SMs

4.11. Scheduling kernels to SMs [25], [18]

• A global scheduler, called the Gigathread scheduler assigns work to each SM.

• In previous generations (G80, G92, GT200) the global scheduler could only assign work to the SMs from a single kernel (serial kernel execution).

• The global scheduler of Fermi is able to run up to 16 different kernels concurrently, one per SM.

• A large kernel may be spread over multiple SMs.

4.24. ábra -

The context switch time occurring between kernel switches is greatly reduced compared to the previous generation, from about 250 µs to about 20 µs (needed for cleaning up TLBs, dirty data in caches, registers etc.) [52].

4.12. Scheduling thread blocks associated with the same kernel to the SMs

• The Gigathread scheduler assigns up to 8 thread blocks of the same kernel to each SM.

  (Thread blocks assigned to a particular SM must belong to the same kernel).

• Nevertheless, the Gigathread scheduler can assign different kernels to different SMs, so up to 16 concurrent kernels can run on 16 SMs.

4.25. ábra -
4.13. The notion and main features of thread blocks in CUDA [53]

- Programmer declares blocks:
  - Block shape: 1D, 2D, or 3D
  - Block size: 1 to 512 concurrent threads
  - Block dimensions in threads
• Each block can execute in any order relative to other blocks!
• All threads in a block execute the same kernel program (SPMD)
• Threads have thread id numbers within block
  • Thread program uses thread id to select work and address shared data
  • Threads in the same block share data and synchronize while doing their share of the work
  • Threads in different blocks cannot cooperate

4.26. ábra -

4.14. Segmenting thread blocks into warps [59]
• Threads are scheduled for execution in groups of 32 threads, called the warps.
• For this reason each thread block needs to be subdivided into warps.
• The scheduler of an SM can maintain up to 48 warps at any point of time.

Remark
The number of threads constituting a warp is an implementation decision and not part of the CUDA programming model.

E.g. in the G80 there are 24 warps per SM, whereas in the GT200 there are 32 warps per SM.

4.27. ábra -
4.15. Scheduling warps for execution in SMs

Nvidia did not reveal details of the microarchitecture of Fermi so the subsequent discussion of warp scheduling is based on assumptions given in the sources [52], [11].

Assumed block diagram of the Fermi GF100 microarchitecture and its operation

- Based on [52] and [11], Fermi’s front end can be assumed to be built up and operate as follows:

- The front end consist of dual execution pipelines or from another point of view of two tightly coupled thin cores with dedicated and shared resources.

- Dedicated resources per thin core are
  - the Warp Instruction Queues,
  - the Scoreboarded Warp Schedulers and
  - 16 FX32 and 16 FP32 ALUs.

- Shared resources include
  - the Instruction Cache,
  - the 128 KB (32 K registersx32-bit) Register File,
Nvidia’s Fermi family of graphics cores and cards

- the four SFUs and the
- 64 KB LiD Shared Memory.

**Remark**

Fermi’s SM’s front end is similar to the basic building block of AMD’s Bulldozer core (2011) that consists of two tightly coupled thin cores [60].
4.29. ábra - The Bulldozer core [60]

4.16. Assumed principle of operation-1

- Both warp schedulers are connected through a partial crossbar to five groups of execution units, as shown in the figure.
- Up to 48 warp instructions may be held in dual instruction queues waiting for issue.
- Warp instructions having all needed operands are eligible for issue.
- Scoreboarding tracks the availability of operands based on the expected latencies of issued instructions, and marks instructions whose operands became already computed as eligible for issue.
- Fermi’s dual warp schedulers select two eligible warp instruction for issue in every two shader cycles according to a given scheduling policy.
- Each Warp Scheduler issues one warp instruction to a group of 16 ALUs (including an FX32 and an FP32 unit), to 4 SFUs or 16 load/store units (not shown in the figure).

4.30. ábra - The Fermi core [52]
4.17. Assumed principle of operation-2

Warp instructions are issued to the appropriate group of execution units as follows:

FX and FP32 arithmetic instructions, including FP FMA instructions are forwarded to 16 32-bit ALUs, each of them incorporating an FX32 ALU (ALU) and an FP32 ALU (FPU)

FX instructions will be executed in the FX32 units whereas SP FP instructions in the SP FP units.
FP64 arithmetic instructions, including FP64 FMA instructions will be forwarded to both groups of 16 FP32 units (FPUs) in the same time, thus DP FMA instructions enforce single issue.

FP32 transcendental instructions will be issued to the 4 SPUs.

4.31. ábra - The Fermi core [52]
• A warp scheduler needs multiple shader cycles to issue the entire warp (i.e. 32 threads), to the available number of execution units of the target group.

The number of shader cycles needed is determined by the number of execution units available in a particular group, e.g.:

- FX or FP32 arithmetic instructions: 2 cycles
- FP64 arithmetic instructions: 2 cycles (but they prevent dual issue)
- FP32 transcendental instructions: 8 cycles
- Load/store instructions: 2 cycles.

Execution cycles of further operations are given in [13].

4.32. ábra - The Fermi core [52]
4.19. Example: Throughput of arithmetic operations per clock cycle per SM [13]

4.33. ábra -
4.20. Scheduling policy of warps in an SM of Fermi GF100-1

Official documentation reveals only that Fermi GF100 has dual issue zero overhead prioritized scheduling [11]

4.34. ábra -

![Diagram of warp scheduler and instruction dispatch](image-url)
4.21. Scheduling policy of warps in an SM of Fermi GT100-2

Official documentation reveals only that Fermi GT100 has dual issue zero overhead prioritized scheduling [11].

Nevertheless, based on further sources [61] and early slides discussing warp scheduling in the GT80 in a lecture held by D. Kirk, one of the key developers of Nvidia’s GPGPUs (ECE 498AL Illinois, [59] the following assumptions can be made for the warp scheduling in Fermi:

- Warps whose next instruction is ready for execution, that is all its operands are available, are eligible for scheduling.
- Eligible warps are selected for execution on a not revealed priority scheme that is based presumably on the warp type (e.g. pixel warp, computational warp), instructions type and age of the warp.
- Eligible warp instructions of the same priority are scheduled presumably according to a round robin policy.
- It is not unambiguous whether or not Fermi is using fine grained or coarse grained scheduling.

Early publications discussing warp scheduling in the GT80 [59] let assume that warps are scheduled coarse grained but figures in the same publication illustrating warp scheduling show to the contrary fine grain scheduling, as shown subsequently.

Remark

As discussed before, in case of coarse grain scheduling wavefronts are allowed to run as long as they do not stall, by contrast, in case of fine grain scheduling the scheduler selects in every new cycle a wavefront to run.

Remarks

D. Kirk, one of the developers of Nvidia’s GPGPUs details warp scheduling for the G80 in [59], but this publication includes two conflicting figures, one indicating to coarse grain and the other to fine grain warp scheduling as shown below.

Underlying microarchitecture of warp scheduling in an SM of the G80

- The G80 fetches one warp instruction/issue cycle
  - from the instruction L1 cache
  - into any instruction buffer slot.
- Operand scoreboarding is used to prevent hazards
  - An instruction becomes ready after all needed values are deposited.
  - It prevents hazards
  - Cleared instructions become eligible for issue
  - Issue selection is based on round-robin/age of warp.
  - SM broadcasts the same instruction to 32 threads of a warp.

4.35. ábra - Warp scheduling in the G80 [59]
4.22. Scheduling policy of warps in an SM of the G80 indicating coarse grain warp scheduling

- The G80 uses decoupled memory/processor pipelines
  - any thread can continue to issue instructions until scoreboarding prevents issue
  - it allows memory/processor ops to proceed in shadow of other waiting memory/processor ops.

4.36. ábra - Warp scheduling in the G80 [59]

Note

The given scheduling scheme reveals a coarse grain one.

4.23. Scheduling policy of warps in an SM of the G80 indicating fine grain warp scheduling
Nvidia’s Fermi family of graphics cores and cards

- SM hardware implements zero-overhead Warp scheduling
- Warps whose next instruction has its operands ready for consumption are eligible for execution.
- Eligible Warps are selected for execution on a prioritized scheduling policy.
- All threads in a Warp execute the same instruction when selected.

Note

The given scheme illustrates reveals fine grain scheduling, in contrast to the previous figure.

4.37. ábra - Warp scheduling in the G80 [59]

4.24. Estimation of the peak performance of the Fermi GF100 -1

a) Peak FP32 performance per SM

Max. throughput of warp instructions/SM:

- dual issue
Nvidia’s Fermi family of graphics cores and cards

- 2 cycles/issue
  
  \[2 \times \frac{1}{2} = 1 \text{ warp instruction/cycle}\]

b) Peak FP32 performance (\(P_{\text{FP32}}\)) of a GPGPU card

- 1 warp instructions/cycle
- 32 FMA/warp
- 2 operations/FMA
- at a shader frequency of \(f_s\)
- \(n\) SM units

\[
P_{\text{FP32}} = 1 \times 32 \times 2 \times 2 \times f_s \times n
\]

E.g. in case of the GTX580

- \(f_s = 1.401\) GHz
- \(n = 15\)

\[
P_{\text{FP32}} = 2 \times 32 \times 1401 \times 15 = 1344.96 \text{ GFLOPS}
\]

4.38. ábra - The Fermi core [52]
4.25. Estimation of the peak performance of the Fermi GF100 -2

c) Peak FP64 performance per SM

Max. throughput of warp instructions/SM:

• single issue
• 2 cycles/issue
Nvidia’s Fermi family of graphics cores and cards

1 x 1/2 = 1/2 warp instruction/cycle

d) Peak FP64 performance ($P_{FP64}$) of a GPGPU card

- 1 warp instruction/2 cycles
- 32 FMA/warp
  - 2 operations/FMA
- at a shader frequency of $f_s$
- $n$ SM units

$$P_{FP64} = \frac{1}{2} \times 32 \times 2 \times f_s \times n$$

E.g. in case of the GTX480

- $f_s = 1.401$ GHz
- $n = 15$ but only $1/4$th of the FP64 ALUs are activated

$$P_{FP64} = 32 \times \frac{1}{4} \times 1401 \times 15 = 168.12 \text{ GFLOPS}$$

(The full (4x) speed is provided only on Tesla devices.

4.39. ábra - The Fermi core [52]
5. Microarchitecture of Fermi GF104

Introduced in 7/2010 for graphic use

Key difference between the GF104 and the GF100: number of cores (SM units), whereas the GF100 has 16 cores (SM units) the GF104 includes only 8.

4.40. ábra - Contrasting the overall structures of the GF104 and the GF100 [62]
Note

In the GF104 based GTX 460 flagship card Nvidia activated only 7 SMs rather than all 8 SMs available, due to overheating.

5.1. Available execution resources per SM in the GF104 vs the GF100

4.41. ábra -

<table>
<thead>
<tr>
<th></th>
<th>GF100 /GF110</th>
<th>GF104</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of SP FX/FP ALUs</td>
<td>32</td>
<td>48</td>
</tr>
<tr>
<td>No. of L/S units</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>No. of SFUs</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>No. of DP FP ALUs</td>
<td>(16)³</td>
<td>4</td>
</tr>
</tbody>
</table>

¹ An FP64 instruction will be forwarded to two FP32 execution units of both groups of 16 FP32 units in the same time, thus FP64 instructions enforce single issue.

Note
The modifications done in the GF104 vs the GF100 aim at increasing graphics performance per SM at the expense of FP64 performance while halving the number of SMs in order to reduce power consumption and price.

5.2. Throughput of arithmetic operations per clock cycle per SM in the GF104/114 [13]

4.42. ábra -

<table>
<thead>
<tr>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit floating-point add, multiply, multiply-add</td>
</tr>
<tr>
<td>64-bit floating-point add, multiply, multiply-add</td>
</tr>
<tr>
<td>32-bit integer add, logical operation</td>
</tr>
<tr>
<td>32-bit integer shift, compare</td>
</tr>
<tr>
<td>32-bit integer multiply, multiply-add, sum of absolute difference</td>
</tr>
<tr>
<td>24-bit integer multiply (_[u]mul24)</td>
</tr>
<tr>
<td>32-bit floating-point reciprocal, reciprocal square root,</td>
</tr>
<tr>
<td>base-2 logarithm (_log2f), base-2 exponential (_exp2f),</td>
</tr>
<tr>
<td>sine (_sinf), cosine (_cosf)</td>
</tr>
<tr>
<td>Type conversions</td>
</tr>
</tbody>
</table>

5.3. Warp issue in the Fermi GF104

SMs of the GF104 have dual thin cores (execution pipelines), each with 2-wide superscalar issue [63]

It is officially not revealed how the issue slots (1a – 2b) are allocated to the groups of execution units.

4.43. ábra -
5.4. Peak computational performance data for the Fermi GF104 based GTX 460 card

According to the computational capability data [13] and in harmony with the figure on the previous slide:

Peak FP32 FMA performance per SM:

2 x 48 FMA operations/shader cycle per SM

Peak FP32 performance of a GTX460 card while it executes FMA warp instructions:

\[
P_{FP32} = 2
\]
Nvidia’s Fermi family of graphics cores and cards

with

• fs: shader frequency
• n: number of SM units

For the GTX 460 card

• fs = 1350 MHz
• n = 7

\[ P_{FP32} = 2 \times 48 \times 1350 \times 7 = 907.2 \text{ MFLOPS} \]

Peak FP64 performance of a GTX 460 card while it executes FMA instructions:

\[ P_{FP64} = 2 \times 4 \times 1350 \times 7 = 75.6 \text{ MFLOPS} \]

6. Microarchitecture of Fermi GF110

Microarchitecture of Fermi GF110 [64]

• Introduced in 11/2010 for general purpose use

The GF110 is a redesign of the GF100 resulting in less power consumption and higher speed.

As long as in the GF100-based flagship card (GTX 480) only 15 SMs could be activated due to overheating, the GF110 design allows Nvidia to activate all 16 SMs in the associated flagship card (GTX 580) and at the same time to increase clock speed.

Key differences between the GF100-based GTX 480 and the GF110-based GTX 580 cards

4.44. ábra -

<table>
<thead>
<tr>
<th>No. of SMs</th>
<th>No. SP ALUs</th>
<th>Shader frequency</th>
<th>TDP</th>
</tr>
</thead>
</table>

• Due to its larger shader frequency and increased number of SMs the GF110-based GTX 580 card achieves a ~ 10 % peak performance over the GF100 based GTX 480 card by a somewhat reduced power consumption.

6.1. Overall structure of the Fermi Gf110 core [65]
4.45. ábra -

6.2. Block diagrams of the cores (SM units) of the four sub-families of Fermi [66]

4.46. ábra -
6.3. Throughput of arithmetic operations per clock cycle per SM in the GF104/114 [13]

4.47. ábra -
### 7. Microarchitecture of Fermi GF114

#### 7.1. Overall structure of the Fermi GF114 core in the GTX 560 card [67]

4.48. ábra -

<table>
<thead>
<tr>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit floating-point</td>
</tr>
<tr>
<td>- add, multiply, multiply-add</td>
</tr>
<tr>
<td>64-bit floating-point</td>
</tr>
<tr>
<td>- add, multiply, multiply-add</td>
</tr>
<tr>
<td>32-bit integer</td>
</tr>
<tr>
<td>- add, logical operation</td>
</tr>
<tr>
<td>32-bit integer</td>
</tr>
<tr>
<td>- shift, compare</td>
</tr>
<tr>
<td>32-bit integer</td>
</tr>
<tr>
<td>- multiply, multiply-add, sum of absolute difference</td>
</tr>
<tr>
<td>24-bit integer multiply</td>
</tr>
<tr>
<td>- (\texttt{mulp24})</td>
</tr>
<tr>
<td>32-bit floating-point</td>
</tr>
<tr>
<td>- reciprocal, reciprocal square root</td>
</tr>
<tr>
<td>- base-2 logarithm (\texttt{log2f})</td>
</tr>
<tr>
<td>- base-2 exponential (\texttt{exp2f})</td>
</tr>
<tr>
<td>- sine (\texttt{sinf}), cosine (\texttt{cosf})</td>
</tr>
<tr>
<td>Type conversions</td>
</tr>
</tbody>
</table>
7.2. Microarchitecture of Fermi GF114

• Introduced in 1/2011 along with the GTX 560Ti card, followed by the lower performance GTX560 version in 5/2011 for graphics use.

The GF114 is a redesign of the GF104 resulting in less power consumption and higher speed.

As long as in the GF104-based GTX 460 card only 7 SMs were activated, the GF114 design activates 8 SMs in the associated higher performance GTX 560Ti card by a higher shader clock speed.

Key differences between the GF104-based GTX 460 and the GF110-based GTX 560/560Ti cards

<table>
<thead>
<tr>
<th></th>
<th>GTX (GF104)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of SMs</td>
<td>7</td>
</tr>
<tr>
<td>No. SP ALUs</td>
<td>33</td>
</tr>
<tr>
<td>Shader frequency</td>
<td>1350</td>
</tr>
<tr>
<td>TDP</td>
<td>160</td>
</tr>
</tbody>
</table>

Due to its larger shader frequency and increased number of SMs the GF114-based GTX 560Ti card achieves a ~10 % higher peak performance than the GF104 based GTX 460 card by a somewhat increased power consumption [68].
7.3. Block diagrams of the cores (SM units) of the four sub-families of Fermi [66]

4.50. ábra -

7.4. Throughput of arithmetic operations per clock cycle per SM in the GF104/114 [13]

4.51. ábra -
<table>
<thead>
<tr>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit floating-point</td>
</tr>
<tr>
<td>add, multiply, multiply-add</td>
</tr>
<tr>
<td>64-bit floating-point</td>
</tr>
<tr>
<td>add, multiply, multiply-add</td>
</tr>
<tr>
<td>32-bit integer</td>
</tr>
<tr>
<td>add, logical operation</td>
</tr>
<tr>
<td>32-bit integer</td>
</tr>
<tr>
<td>shift, compare</td>
</tr>
<tr>
<td>32-bit integer</td>
</tr>
<tr>
<td>multiply, multiply-add, sum of absolute difference</td>
</tr>
<tr>
<td>24-bit integer multiply</td>
</tr>
<tr>
<td>([u]mul24)</td>
</tr>
<tr>
<td>32-bit floating-point</td>
</tr>
<tr>
<td>reciprocal, reciprocal square root,</td>
</tr>
<tr>
<td>base-2 logarithm ([log2f]),</td>
</tr>
<tr>
<td>base-2 exponential ([exp2f]),</td>
</tr>
<tr>
<td>sine ([sinf]), cosine ([cosf])</td>
</tr>
<tr>
<td>Type conversions</td>
</tr>
</tbody>
</table>
5. fejezet - Nvidia’s Kepler family of graphics cores and cards

1. Overview of Nvidia’s Kepler family

5.1. ábra - Overview of GPGPU cores and cards and their basic software support (3)

1.1. Sub-families of Kepler

Kepler includes recently two sub-families with the following features:

5.2. ábra -

<table>
<thead>
<tr>
<th>GPGPU core</th>
<th>Available since</th>
<th>Max. no. of SMXs (cores)</th>
<th>Max. no. of 32-bit ALUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>GK104</td>
<td>3/2012</td>
<td>8</td>
<td>8x192 = 1536</td>
</tr>
<tr>
<td>GK1101,2</td>
<td>11/2012</td>
<td>15</td>
<td>15x192 = 2880</td>
</tr>
</tbody>
</table>

1. On the GK110 based K20 Tesla part Nvidia fused two SMX units off, thus the K20 has only 13 SMXs with 13x192 = 2496 FX/FP32 EUs [69].

2. On the GK110 based K20X Tesla and GTX Titan parts Nvidia fused only one SMX unit off, thus both parts have 14 SMXs with 14x192 = 2688 FX/FP32 EUs [69].

1.2. The GK104 Kepler core [76]

5.3. ábra - Block diagram of the GK104 Kepler core
1.3. The GK110 Kepler core [70]

5.4. ábra - Block diagram of the GK110 Kepler core

1.4. Main features of Nvidia’s Kepler GK104 and GK110 cores [70]

5.5. ábra -
2. Key innovations of Kepler’s PTX 3.x

Along with the introduction of a set of key innovations into their PTX 2.0 Nvidia promised to have entered a phase of relative consolidation of their pseudo ISA.

PTX 3.0 fulfilled this expectation as the only major innovation of this version was to support the related Kepler GK104 graphics cores (termed internally as the sm_30 target architecture) [71].

The subsequent PTX 3.1 supports additionally [20]:

• the related Kepler GK110 cores (termed internally as the sm_35 target architecture) and

• dynamic parallelism, to be discussed in the next Section.

3. Major innovations of Kepler’s microarchitecture

a) Dynamic parallelism

b) The Hyper-Q scheduling model

c) The Grid Management Unit

d) The GPU Direct feature

Note

These innovative features are available only for GK110 based devices.

3.1. a) Dynamic parallelism [70]

In Fermi all work is launched by the CPU, they are running to completion in the GPU and then the results are returned back to the CPU.

Intermediate results will be analyzed by the CPU and if needed they will be resent for further processing again to the GPU and so forth.
The dynamic parallelism feature of the Kepler architecture greatly simplifies and speeds up processing.

With dynamic parallelism any kernel in a Kepler GPU can launch another kernel and create the necessary stream, events as well as can manage dependencies without the need for CPU intervention, as indicated in the next Figure.

3.1.1. Principle of dynamic parallelism in Kepler GPUs

5.6. ábra - Principle of dynamic parallelism in Kepler GPUs

Dynamic parallelism allows a more efficient execution of kernels directly by itself with

3.1.2. Benefits of dynamic parallelism [70]

It allows developers to optimize the execution of parallel code first of all in recursive and data dependent applications.

With dynamic parallelism more parts of a program will run directly on the GPU and the host CPU becomes freed up for performing additional tasks or even a less powerful CPU can be used to run the same workload.

3.2. b) The Hyper-Q scheduling mechanism

The Fermi architecture supports up to 16-way Concurrent Kernel Execution from separate streams, as indicated in the right part of the next Figure.

5.7. ábra - Serial and Concurrent Kernel execution models of Fermi and pre-Fermi devices [25], [18]
Remarks

1) In CUDA a stream is understood as a sequence of commands, such as kernel launches, that has to be executed in order [72].

2) Fermi’s Concurrent kernel execution model was a major innovation vs. pre-Fermi devices, such as the G80, G92, G200, since the global scheduler (called the GigaThread scheduler) of pre-Fermi devices could assign work to the SMs only serially, that is only from a single kernel (Serial Kernel Execution), as shown in the right part of the Figure.

5.8. ábra - Serial and Concurrent Kernel execution models of Fermi and pre-Fermi devices [25], [18]

The drawback of Fermi’s execution model is however that ultimately all the streams were multiplexed into the same hardware work queue, as the next Figure shows.

5.9. ábra - Using a single hardware work queue to schedule multiple streams of kernels [70]
This execution model causes however false intra-stream dependencies requiring that dependent kernels within one stream should be completed before other kernels being in a separate stream can be launched for execution, as indicated in the next Figure [70].

5.10. ábra - Fermi’s model of concurrent execution [70]
As a consequence of the single hardware work queue only the kernels (C,P) and (R,X) can run concurrently, as intra-stream dependencies enforce serial execution within each stream [70].

By contrast, Kepler’s Hyper-Q scheduling mechanism allows to set up 32 simultaneous, hardware managed work queues between the host CPU and the CUDA Work Distributor Logic (WDL) (compared to the single work queue available with Fermi), as indicated below.

5.11. ábra - Principle of the Hyper-Q scheduling model [70]
Now, with multiple hardware work queues each stream of kernels can run simultaneously with other streams of kernels, as shown in the Figure.

5.12. ábra - Principle of Hyper-Q [70]
Hyper-Q offers significant benefits for MPI-based parallel computations by allowing up to 32 simultaneous, hardware managed streams (MPI tasks) compared to a single stream of MPI tasks allowed by Fermi’s scheduling scheme, as shown below.

5.13. ábra - Kepler’s Hyper-Q scheduling mechanism used to issue simultaneous MPI tasks compared to the MPI task issue mechanism of Fermi [73]

MPI: Message Passing Interface, used in parallel computations.

3.2.1. Speed up potential of the Hyper-Q scheduling mechanism

Applications that previously encountered false serialization across tasks, thereby limiting GPU utilization, can now be speeded up to 32 times without changing any existing code [70].
3.3. c) Grid Management Unit

As discussed above, Kepler’s GPU architecture introduced two new features in order to enhance the efficiency of scheduling kernels:

- In a Kepler GPU any running kernel is allowed to launch another kernel and create the necessary stream (called the Dynamic Parallelism feature) and
- also there are multiple hardware work queues available (Hyper-Q scheduling mechanism).

These enhancements made a complete redesign of the scheduling subsystem of the Fermi architecture necessary. Kernel point of the new design is the Grid Management Unit that takes the responsibility for handling the tasks mentioned.

A block diagram of the new complex scheduling subsystem is given in the next Figure.

Instead of going into details of its operation we refer to the related literature [70].

5.14. ábra - Contrasting the scheduling subsystems of the Fermi and Kepler architectures [70]

![Fermi Workflow Diagram](image)

3.4. d) The GPU Direct feature

It allows direct access to GPU memory from 3rd party devices, such as network adapters, to facilitate direct transfers between GPUs, as shown below.

5.15. ábra - The GPU Direct feature of Kepler [70]
This feature can be beneficial for building supercomputers by including GPGPU-based DPAs (Data Parallel Accelerators) into them, like done in the Titan supercomputer.

The Titan supercomputer (Cray XK7), installed at Oak Ridge National Laboratory (USA) is based on 16 888 nodes, each one including a 16-core AMD Opteron 6274 (Bulldozer based Interlagos) server processor and an Nvidia Tesla K20X data accelerator.

The Titan supercomputer achieved a performance record of 17.59 PetaFLOPS, as measured by the LINPACK benchmark and became the no.1 supercomputer in 11/2012 [74], [75].

5.16. ábra - View of the Titan (Cray XK7) supercomputer [75]

3.5. Major enhancements of Kepler’s microarchitecture

a) Setting down the operating frequency of the execution units (shader clock frequency) to the core frequency
b) Simplifying hardware dependency checks by introducing compiler hints

c) Introducing quad warp schedulers per SMX

d) Quadrupling the number of registers that are accessible per thread

e) Introduction of a 48 KB Read-Only data cache for general use

f) Doubled L2 cache size and bandwidth vs Fermi

3.5.1. a) Setting down the operating frequency of the execution units (shader clock frequency) to the core frequency-1

Basically, there are two design options to double performance potential,

• either to let run the execution units at twice the original clock frequency or

• to double the number of available execution units, and let run them at the original clock speed.

The first option needs about the same silicon area as before but has higher power consumption, whereas the second option requires roughly twice the silicon area but needs less power than the first option.

Remark

• Higher clock frequency (fc) requires higher supply voltage V).

• As the dynamic dissipation of a processor D = const x fc x V^2, higher fc implies higher power consumption.

• On the other hand larger silicon area results in higher fabrication cost, thus the real trade-off is less power consumption for higher fabrication cost or vice versa.

3.5.2. a) Setting down the operating frequency of the execution units (shader clock frequency) to the core frequency-2

Designers of the G80 to the Fermi family of GPUs opted for at least doubling the clock frequency of the execution units (termed as the shader frequency) to raise performance, as shown below.

5.17. ábra - Core clock frequency vs shader frequency in Nvidia’s major GPU families

<table>
<thead>
<tr>
<th>Core</th>
<th>G80</th>
<th>G92</th>
<th>GT200</th>
</tr>
</thead>
<tbody>
<tr>
<td>Card</td>
<td>88 GTX</td>
<td>8800 GT</td>
<td>GTX 280</td>
</tr>
<tr>
<td>Intro</td>
<td>11/06</td>
<td>10/07</td>
<td>6/08</td>
</tr>
<tr>
<td>Core frequency</td>
<td>575 MHz</td>
<td>600 MHz</td>
<td>602 MHz</td>
</tr>
<tr>
<td>Shader frequency</td>
<td>1.35 GHz</td>
<td>1.512 GHz</td>
<td>1.296 GHz</td>
</tr>
</tbody>
</table>

The benefit of the chosen option is that it needs less silicon area and thus reduces fabrication cost, but the drawback is higher power consumption, compared to the second option.

3.5.3. a) Setting down the operating frequency of the execution units (shader clock frequency) to the core frequency-3

By contrast, beginning with Kepler Nvidia opted for reducing power consumption, by optimizing the performance/Watt figure instead of rough performance.

Accordingly, designers switched to the other basic option i.e. they strongly increased the number of execution units and set down the shader frequency to the operating clock frequency, as the next tables show.
5.18. ábra - Available execution resources in a Kepler SMX vs. a Fermi SM

<table>
<thead>
<tr>
<th></th>
<th>GF100/GF110</th>
<th>GF104</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of FX/FP32 ALUs</td>
<td>32</td>
<td>48</td>
</tr>
<tr>
<td>No. of I/S units</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>No. of SPUs</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>No. of FP64 ALUs</td>
<td>(16)</td>
<td>4</td>
</tr>
</tbody>
</table>

1 An FP64 instruction will be forwarded to two FP32 same time, thus FP64 instructions enforce single

3.5.4. a) Setting down the operating frequency of the execution units (shader clock frequency) to the core frequency-4

5.19. ábra - Core clock frequency vs shader frequency in Nvidia’s major GPU families

<table>
<thead>
<tr>
<th>Core</th>
<th>G80</th>
<th>G92</th>
<th>GT200</th>
<th>GF100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Card</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intro</td>
<td>11/06</td>
<td>16/07</td>
<td>6/08</td>
<td>3/10</td>
</tr>
<tr>
<td>Core frequency</td>
<td>575 MHz</td>
<td>600 MHz</td>
<td>602 MHz</td>
<td>760 MHz</td>
</tr>
<tr>
<td>Shader frequency</td>
<td>1350 MHz</td>
<td>1512 MHz</td>
<td>1296 MHz</td>
<td>1401 MHz</td>
</tr>
</tbody>
</table>

3.5.5. Comparing the silicon area needed and power consumption of the design approaches used to implement Fermi’s and Kepler's execution units [76]

5.20. ábra -

Remark
As far as AMD’s GPU implementations concerns AMD stuck at clocking the EUs at the basic clock frequency, as indicated in the next Table.

### 5.21. ábra - Main features of Nvidia’s and AMD’s GPU cards [77]

<table>
<thead>
<tr>
<th>Graphics card</th>
<th>Geforce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>GTX 480</td>
</tr>
<tr>
<td></td>
<td>GTX 470</td>
</tr>
<tr>
<td></td>
<td>GTX 2</td>
</tr>
<tr>
<td>Price (Euro)</td>
<td>500,-</td>
</tr>
<tr>
<td></td>
<td>350,-</td>
</tr>
<tr>
<td></td>
<td>200,-</td>
</tr>
<tr>
<td>Market introduction</td>
<td>Mar 2010</td>
</tr>
<tr>
<td></td>
<td>Mar 2010</td>
</tr>
<tr>
<td></td>
<td>Jan 2009</td>
</tr>
<tr>
<td>Code name</td>
<td>GF100</td>
</tr>
<tr>
<td></td>
<td>GF100</td>
</tr>
<tr>
<td></td>
<td>GT200</td>
</tr>
<tr>
<td>DirectX / Shader Version</td>
<td>11/5.0</td>
</tr>
<tr>
<td></td>
<td>11/5.0</td>
</tr>
<tr>
<td></td>
<td>10.8/4</td>
</tr>
<tr>
<td>Process</td>
<td>40 nm</td>
</tr>
<tr>
<td></td>
<td>40 nm</td>
</tr>
<tr>
<td></td>
<td>55 nm</td>
</tr>
<tr>
<td>Transistors (Mio.)</td>
<td>3000</td>
</tr>
<tr>
<td></td>
<td>3000</td>
</tr>
<tr>
<td></td>
<td>1400</td>
</tr>
<tr>
<td>GFLOPS, SP</td>
<td>1244.96</td>
</tr>
<tr>
<td></td>
<td>1090.64</td>
</tr>
<tr>
<td></td>
<td>708.4</td>
</tr>
<tr>
<td>GFLOPS, DP</td>
<td>672.48</td>
</tr>
<tr>
<td></td>
<td>544.32</td>
</tr>
<tr>
<td></td>
<td>88.51</td>
</tr>
<tr>
<td>GPiK/s</td>
<td>33.6</td>
</tr>
<tr>
<td></td>
<td>24.28</td>
</tr>
<tr>
<td></td>
<td>20.73</td>
</tr>
<tr>
<td>GTex/s</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>32.992</td>
</tr>
<tr>
<td></td>
<td>51.0</td>
</tr>
<tr>
<td>Memory bandwidth (GByte/s)</td>
<td>177.4</td>
</tr>
<tr>
<td></td>
<td>133.92</td>
</tr>
<tr>
<td></td>
<td>158.9</td>
</tr>
<tr>
<td>Chip frequency (MHz)</td>
<td>700</td>
</tr>
<tr>
<td></td>
<td>607</td>
</tr>
<tr>
<td></td>
<td>648</td>
</tr>
<tr>
<td>Shader ALU frequency (MHz)</td>
<td>1401</td>
</tr>
<tr>
<td></td>
<td>1215</td>
</tr>
<tr>
<td></td>
<td>1476</td>
</tr>
<tr>
<td>VRAM frequency (MHz)</td>
<td>1848</td>
</tr>
<tr>
<td></td>
<td>1674</td>
</tr>
<tr>
<td></td>
<td>1242</td>
</tr>
</tbody>
</table>

#### 3.5.5.1. Resulting power efficiency of the Kepler design vs. the Fermi design

Due to the overall design approach taken to optimize for performance/watt resulted in considerable higher power efficiency compared to the Fermi approach, as indicated below [76].

### 5.22. ábra - Comparing the power efficiency of the Fermi and Kepler designs [76]
3.5.6. b) Simplifying hardware dependency checks by introducing compiler hints

The Fermi design includes a complex hardware scheduler to perform dependency checking for warp scheduling, as shown below.

5.23. ábra - Block diagram of Fermi’s hardware dependency checking [76]

In their Kepler line, Nvidia greatly simplified hardware dependency checking by letting the compiler to give scheduling information revealing when instructions will be ready to issue.

This information can be calculated based on the fixed latencies of the execution pipelines.

This results in a more straightforward scheduler design, as shown in the next Figure.

5.24. ábra - Block diagram of Kepler’s hardware dependency checking [76]

3.5.7. c) Introducing quad warp schedulers per SMX

As stated before, Nvidia introduced a multiple of execution resources in their Kepler architecture, compared to the Fermi line, as indicated below.

3.5.8. Available execution resources in a Kepler SMX vs. a Fermi SM

5.25. ábra -

<table>
<thead>
<tr>
<th></th>
<th>GF100</th>
<th>GF104</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of FX/FP32 ALUs</td>
<td>32</td>
<td>48</td>
</tr>
<tr>
<td>No. of L/S units</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>No. of SFUs</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>No. of FP64 ALUs</td>
<td>(16)(^1)</td>
<td>4</td>
</tr>
</tbody>
</table>

In order to utilize vastly increased execution resources (see the Tables above) Nvidia introduced quad warp schedulers per SMX units, as shown next.

5.26. ábra -
The 4 warp schedulers of an SMX unit select each a particular warp with two independent instructions per warp for dispatching each cycle, as shown below [70].

5.27. ábra -

3.5.9. d) Quadrupling the number of registers that are accessible per thread

Each thread in the GKxxx Kepler cores can access 255 registers, four times as many as in Fermi cores, as the Table below indicates.

5.28. ábra - Key device features bound to the compute capability versions [32]
3.5.10. Consequence of quadrupling the number of registers that can be accessed per thread in the cores of Kepler

Codes that exhibit high register pressure may benefit from having an extended number of registers [70].

3.5.11. e) Introduction of a 48 KB Read-Only data cache for general use

In the Fermi generation of GPGPUs there was already a 48 KB read-only data cache introduced.

Albeit this cache was accessible only by the Texture unit, experienced programmers often made use of this cache.

Kepler made the read-only cache available for general use, as shown below.

5.29. ábra - Kepler’s cache architecture [70]
3.5.12. f) Doubled L2 cache size and bandwidth vs. Fermi

Nvidia doubled the size of the L2 cache in the GK110 to 1.5 MB from 768 KB of the related previous Fermi cores (GF100/GF110) [70].

Note

The L2 size of the GK104 remained further on only 512 KB, actually the same size as that of the previous Fermi GF104/GF114 cores.

4. Nvidia’s GK104 Kepler core and related graphics cards

4.1. Sub-families of Kepler

Kepler includes recently two sub-families with the following features:

5.30. ábra -
Nvidia’s Kepler family of graphics cores and cards

<table>
<thead>
<tr>
<th>GPGPU core</th>
<th>Available since</th>
<th>Max. no. of SMXs (cores)</th>
<th>Max. no. of 32-bit ALUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>GK104</td>
<td>3/2012</td>
<td>8</td>
<td>8x192 = 1536</td>
</tr>
<tr>
<td>GK110-2</td>
<td>11/2012</td>
<td>15</td>
<td>15x192 = 2880</td>
</tr>
</tbody>
</table>

1 On the GK110 based K20 Tesla part Nvidia fused 13 SMXs with 13x192 = 2496 FX/FPU EUs [6]
2 On the GK110 based K20X Tesla and GTX Titan, both parts have 14 SMXs with 14x192 = 2688

4.2. The GK104 Kepler core [76]

5.31. ábra - Block diagram of the GK104 Kepler core

4.3. Die photo of Nvidia’s Kepler GK104 core [78]

5.32. ábra -
4.4. Block diagram of an SMX of the Kepler GK104 core [76]

5.33. ábra -

<table>
<thead>
<tr>
<th>Per SMX available execution resources in the GK104 vs the GK110</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>No. of SP FX/FP ALUS</td>
</tr>
<tr>
<td>No. of L/S units</td>
</tr>
<tr>
<td>No. of SFUs</td>
</tr>
<tr>
<td>No. of DP FP ALUs</td>
</tr>
</tbody>
</table>

\(^1\) Not shown in the Figure
Both the GeForce GTX 680 and the K10 incorporate 8 SMX units.

5.34. ábra - Main features of Nvidia’s Kepler-based GTX 680 GPGPU card [33]

<table>
<thead>
<tr>
<th>Core</th>
<th>Introduction</th>
<th>IC technology</th>
<th>Nr. of transistors</th>
<th>Die are</th>
<th>Core frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Computation</th>
<th>No of SMs (core)</th>
<th>No. of FP32 ALU</th>
<th>Shader frequency</th>
<th>No. FP32 operat</th>
<th>Peak FP32 perfom</th>
<th>Peak FP64 perfom</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th>Mem. transfer rate</th>
<th>Mem. interface</th>
<th>Mem. bandwidth</th>
<th>Mem. size</th>
<th>Mem. type</th>
<th>Mem. channel</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>System</th>
<th>Multi. CPU tech</th>
<th>Interface</th>
<th>MS Direct X</th>
<th>TDP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.5. Peak FP32 and FP34 performance of the GK104 based GeForce GTX 680 graphics card

4.5.1. a) Peak FP32 performance of the GK104 based GeForce GTX 680 graphics card-1

Each SMX of the GeForce GTX 680 has 192 FX/FP32 ALUs, as shown in the next Figure.

4.5.2. a) Peak FP32 performance of the GK104 based GeForce GTX 680 graphics card-2 [70]

5.35. ábra -
In the K20 Nvidia fused off two and in the K20X a single SMX unit.

4.5.3. a) Peak FP32 performance of the GK104 based GeForce GTX 680 graphics card-3

These ALUs perform FMA operations at a shader frequency of \( f_{SH} \) in a pipelined fashion. Consequently the max. FP32 performance of a single SMX unit is

\[
P_{FP32} = 2 \times n_{FP32} \times f_{SH}
\]

with

\( n_{FP32} \): Number of the FP32 ALUs in the SMX

\( f_{SH} \): Shader frequency

Based on the actual figures

\( n_{FP32} = 192 \) and

\( f_{SH} = 1006 \) MHz

the peak FP32 performance of an SMX unit equals to:

\[
P_{FP32} = 2 \times 192 \times 1006 = 386.3 \text{ GFLOPS}
\]

The GeForce GTX 680 cards have 8 SMX units, thus the peak FP32 performance of the GTX 680 card amounts to

\[
P_{FP32} = 2 \times 192 \times 1006 \times 8 = 3090 \text{ GFLOPS}
\]

4.5.4. b) Peak FP64 performance of the GK104 based GeForce GTX 680 graphics card-1

Each SMX of the GeForce GTX 680 has 8 FP64 ALUs, as shown in the next Figure.
Nvidia’s Kepler family of graphics cores and cards

4.5.5. b) Peak FP64 performance of the GK104 based GeForce GTX 680 graphics card-2 [70]

5.36. ábra -

![Per SMX available execution resources in the GK104 vs the GK110](image)

In the K20 Nvidia fused off two and in the K20X a single GK110 SMX unit.

4.5.6. b) Peak FP64 performance of the GK104 based GeForce GTX 680 graphics card-3

In the GTX 680 graphics card each SMX unit has 192 FX/FP32 and 8 FP64 ALUs. Thus the FP64 performance is $8/192 = 1/24$th of the FP32 performance of this card. Consequently, the FP64 performance of the GTX 680 card amounts to

$$P_{FP64} = \frac{3090\text{ GFLOPS}}{24} = 128.75\text{ GFLOPS}$$

5. Nvidia’s GK110 Kepler core and related graphics cards

5.1. Sub-families of Kepler

Kepler includes recently two sub-families with the following features:

5.37. ábra -
5.2. The GK110 core

It underlies the GTX Titan graphics card and the K20 as well as K20X data accelerator cards.

Although the GK110 core includes 15 SMX units

On the GK110 based K20 Tesla part Nvidia fused two SMX units off, thus the K20 has only 13 SMXs with 13x192 = 2496 FX/FP32 EUs [69], whereas

On the GK110 based K20X Tesla and GTX Titan parts Nvidia fused only one SMX unit off, thus both parts have 14 SMXs with 14x192 = 2688 FX/FP32 EUs [69], as already stated before.

5.3. The GK110 Kepler core [70]

5.38. ábra - Block diagram of the GK110 Kepler core

5.3.1. Die photo of Nvidia’s Kepler GK110 core [70]
5.39. ábra -

5.3.2. Die layout of Nvidia's Kepler GK110 core [79]

5.40. ábra -
5.3.3. Comparison of the die sizes of Nvidia’s GK110 and GK104 Kepler cores [79]

5.41. ábra -
5.3.4. Block diagram of an SMX of the Kepler GK110 core [70]

5.42. ábra -
Nvidia’s Kepler family of graphics cores and cards

5.3.5. a) Peak FP32 performance of the GK110 based GeForce GTX Titan graphics card

Titan’s ALUs perform FMA operations at a shader frequency of $f_{SH}$ in a pipelined fashion. Consequently the max. FP32 performance of a single SMX unit is

$$P_{FP32} = 2 \times n_{FP32} \times f_{SH}$$

Based on the actual figures the peak FP32 performance of a single SMX unit equals to:

The GeForce GTX Titan cards have 14 SMX units, thus the peak FP32 performance of the GTX Titan card amounts to

5.3.6. b) Peak FP64 performance of the GK110 based GeForce GTX Titan graphics card

In the GTX Titan graphics card each SMX unit has 192 FX/FP32 and 64 FP64 ALUs. Thus the FP64 performance is $8/64 = 1/8$th of the FP32 performance of this card. Consequently, the FP64 performance of the GTX Titan card amounts to

We note that in contrast to the peak FP64 performance calculated above, the published technical documents indicate only a $P_{FP64}$ figure of 1300 GFLOPS for the GTX Titan card.

A possible explanation is, that the GK110 based K20X compute unit has only a core frequency of 732 MHz resulting in a $P_{FP64}$ of 1300 GFLOPS but its price can be assumed to be much higher than that of the GTX Titan card ($\approx 1000$ $\), so Nvidia artificially reduced the peak $P_{FP64}$ performance of their GTX Titan card.

6. Evolution of Nvidia’s microarchitectures
6.1. a) Evolution of FP32 warp issue efficiency in Nvidia’s pre-Fermi GPGPUs

5.43. ábra -

6.2. b) Evolution of FP32 warp issue efficiency in Nvidia’s Fermi and Kepler GPGPUs

5.44. ábra -

6.3. c) FP64 performance increase in Nvidia’s pre-Fermi GPGPUs
Performance is bound by the number of available DP FP execution units.

### 5.45. ábra -

<table>
<thead>
<tr>
<th></th>
<th>G80/G92 (11/06, 10/07)</th>
<th>GT200 (06/08)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available FP64 units/SM operations</td>
<td>No FP64 support</td>
<td>1 FP64 unit (Add, Mul, MAD)</td>
</tr>
<tr>
<td>Peak FP64 issue/SM</td>
<td>1 FP64 MAD</td>
<td>1x2 operations/S</td>
</tr>
<tr>
<td>Peak FP64 perf./cycle/SM</td>
<td>30x1x2x1296</td>
<td>77.76 GFLOPS</td>
</tr>
</tbody>
</table>

**Tesla cards**
- Flagship Tesla card: C1060
- Peak FP64 perf./card: 30x1x2x1296
- Performance: 77.76 GFLOPS

**GPGPU cards**
- Flagship GPGPU card: GT280
- Peak FP64 perf./card: 30x1x2x1296
- Performance: 77.76 GFLOPS

1. In their GPGPU Fermi cards Nvidia activates only 4 FP64 units.
2. No FP64 units are available, an FP64 instruction will be of 16 FP32 units at the same time, thus FP64 instruction...  

### 6.4. d) FP64 performance increase in Nvidia’s Tesla and GPGPUs

Performance is bounded by the number of available DP FP execution units.

### 5.46. ábra -

<table>
<thead>
<tr>
<th></th>
<th>GF100 (03/10)</th>
<th>GF110 (11/10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available FP64 units/SM or SMX operations</td>
<td>(16)² FP64 units (Add, Mul, FMA)</td>
<td>(16)² FP64 units (Add, Mul, FMA)</td>
</tr>
<tr>
<td>Peak FP64 load/SM</td>
<td>16 FP64 FMA</td>
<td>16 FP64 MAD</td>
</tr>
<tr>
<td>Peak FP64 perf./cycle/SM</td>
<td>16x2 operations/SM</td>
<td>16x2 operations/SM</td>
</tr>
</tbody>
</table>

**Tesla cards**
- Flagship Tesla card: C2070
- Peak FP64 perf./card: 14x16x2x1150
- Performance: 515.2 GFLOPS

**GPGPU cards**
- Flagship GPGPU card: GTX 480
- Peak FP64 perf./card: 15x4x2x1401
- Performance: 168.12 GFLOPS

- Flagship GPGPU card: GTX 580
- Peak FP64 perf./card: 16x4x2x1544
- Performance: 197.632 GFLOPS

1. In their GPGPU Fermi cards Nvidia activates only 4 FP64 units.
2. In their GTX Titan GPGPU card Nvidia states only a peak F...
6. fejezet - AMD’s Heterogeneous Systems Architecture concept

1. Introduction to the HSA concept

AMD introduced their FSA (Fusion Systems Architecture) concept at the Fusion Developer Summit in 6/2011 but renamed it shortly afterwards to HSA (Heterogeneous Systems Architecture) in 1/2012, as indicated in the next Figure [82], [83].

1.1. First public introduction of the HSA concept (6/2011) [84], [85]

6.1. ábra -

Remarks

In 1/2012 AMD also renamed their Fusion branding (introduced in 2008) to Heterogeneous Systems Architecture branding.

In AMD’s terminology the Fusion branding designated processors with on-die integrated CPU and GPU, as implemented in their APU lines (Accelerated Processing Units), such as AMD’s Llano-based A4-A8 lines (2011).

2) The motivation behind renaming the Fusion branding to the Heterogeneous Systems Architecture branding was a lawsuit [86] that AMD encountered from the Swiss component maker Arctic who possessed the “Fusion” trademark and used it for a range of power supply products since 2006.

To settle the legal conflict AMD offered a license fee to Arctic and dropped the “Fusion” branding about 1/2012.
1.2. Aims of the Heterogeneous Systems Architecture (HSA) concept

1.2.1. AMD’s view of the evolution of single core, multi-core and application performance (6/2011) [84]

6.2. ábra -

1.2.2. AMD’s view of the evolution path to heterogeneous computing (6/2011) [84]

6.3. ábra - AMD’s view of the evolution to heterogeneous computing [84]

2. The vision of the HSA concept

- HAS is an open architecture specification for heterogeneous multi-core processors conceived to provide a more efficient processing than used recently (in the Standard drivers era).
- It provides a unified computing framework for CPUs and GPUs in a single system with common computing concepts.
• It is an optimized platform for OpenCL.
• It encompasses both hardware and software concepts.

Key components of the HSA
• HAS is an open architecture specification for heterogeneous multi-core processors conceived to provide a more efficient processing than used recently (in the Standard drivers era).
• It provides a unified computing framework for CPUs and GPUs in a single system with common computing concepts.
• It is an optimized platform for OpenCL.
• It encompasses both hardware and software concepts.

Key components of the HSA

a) Memory model, based on a single address space that is accessible to both the CPUs and the GPU in order to avoid data copying.

b) User-level command queuing to minimize communication overhead.

c) Virtual ISA (HSAIL i.e. HSA Intermediate Language) and
d) Model of processing applications.

HSA specifications should be guided by the HSA foundation, established in 6/2012.

2.1. a) The memory model of HSA [87]
• A key architectural feature of HSA is its unified memory model.

  It declares a single unified virtual address space for the CPUs and the GPU.

  All HSA-accessible memory regions are mapped into this virtual address space.

• Memory regions shared between the CPUs and the GPU are coherent.

  This simplifies programming by eliminating the need for explicit cache coherency management.

  The benefit of the unified address space is that it eliminates explicit data movements.

2.2. b) User-level command queuing [87]

HSA specifies a dispatch queue per application for dispatching work for the GPU and also allows to dispatch commands directly into these queues without OS kernel services (called user-level command queuing).

This eliminates waiting for OS kernel services and drastically reduces the time needed to dispatch work to the GPU.

c) The HSA IL (HSA Intermediate Layer) [87]. [88]
• The HSA IL is a virtual ISA for heterogeneous systems.

  Splitting processing into two phases with generating an intermediate code (HSA IL) in the first phase and converting the intermediate code to native GPU and CPU ISA code by a JIT compiler (called Finalizer) in a second phase allows rapid innovations in the native GPU architecture.

• The HSA IL is focused purely on computing and does not expose graphics-specific instructions [88].

• The HSA IL is expected to remain stable across a series of GPU implementations.
2.3. d) Model of processing applications [88]

6.4. ábra -

2.4. Principle of processing applications in HSA [87]

Overview

- The HSA software platform supports multiple high-level parallel programming languages, such as OpenCL, C++AMP, C++, C#, OpenMP, Python etc.
- HSA application processing consists of two phases
  - a compilation phase and
  - a runtime phase.

The compilation phase

It includes two components

- the compiler front ends and
- the compiler back and.

The Compiler Front Ends convert high level languages into the widely used intermediate representation LLVM IR.

The Compiler Back End converts then LLVM IR into HSAIL.

LLVM: Low Level Virtual Machine Intermediate Representation

6.5. ábra - The compilation phase
2.5. The runtime phase [87]

- The HSA runtime works as part of the actual environment.
  E.g. the HSA Runtime is called from the OpenCL Language Runtime.

- The HSA Runtime dispatches work to the HSA CPU and GPU devices.
  During dispatch the Finalizer translates HSAIL into the underlying ISA of the GPU device at runtime.
  The HSA Runtime also dispatches work to the HSA CPU device, whereby OS System services may be invoked.

6.6. ábra - The runtime phase
A classical retargetable three phase compiler design looks like the one indicated in the Figure below.

6.7. ábra - Classical retargetable tree-phase compiler design [89]
Example: The GCC compiler (GCC is interpreted now as the “GNU Compiler Collection”. It supports many front ends and back ends and has an active and broad community of contributors.

By contrast, the LLVM compiler design is also a three-phase design with a well specified and described intermediate representation form, called the LLVM IR.

6.8. ábra -

LLVM IR is a low-level RISC-like virtual instruction set including simple instructions like add, subtract, compare, or branch.

A further key feature of LLVM is that it is designed as a set of libraries, rather than as a monolithic command line compiler like GCC.

Due to its easy adaptability and performance, in the last few years LLVM has evolved from an academic project at the University of Illinois at Urbana–Champaign to the universal back-end of C, C++ and Objective C compilers.

2.6. Heterogeneous System Architecture Foundation (HSAF)

It is a nonprofit Domestic (USA) Corporation, initiated by AMD and filed in 6/2012..

Aims of HSAF

Define and deliver open standards and tools for hardware and software to take advantage of high performance parallel compute engines, such as GPGPUs, by the lowest possible power consumption.

Founding members

AMD, ARM, Imagination Technologies, MediaTek, Qualcomm, Samsung, and Texas Instruments.

2.7. AMD’s roadmap for the implementation of HSA [90]

6.9. ábra -
2.8. AMD’s roadmap for the implementation of HSA - a more detailed view [85]

6.10. ábra -
7. fejezet - AMD’s Southern Islands family of graphics cores and cards

1. Overview of AMD’s Southern Island family

1.1. AMD’s roadmap for the implementation of HSA (Financial Analyst Day 2/2012) [90]

7.1. ábra -

![Diagram showing HSA differentiation between 2011 and 2012, comparing Bobcat-based APUs and Southern Islands families.]

**Remark**

For the Southern Islands family (aka GCN) AMD’s roadmap indicates that the GPU should access the CPU memory already in 2012.

This is also manifested in AMD’s White Paper “AMD Graphics Cores Next (GCN) Architecture” [91].

Nevertheless, AMD’s HD 7970 specification for the first GPU card debuting the Southern Islands family (aka GCN) does not include any reference for this, by contrast there are some indications found on the internet (e.g. [93]) saying that virtual memory support will be provided first in the Sea Islands family, to be announced in 2013.

1.2. AMD’s new GPU microarchitecture: the Graphics Core Next (GCN)

Announced at the AMD Fusion Developer Summit in 6/2011.

Implemented until now in the

- Southern Islands family (Tahiti cores/7950/7970 GPGPU cards)
AMD’s Southern Islands family of graphics cores and cards

- Sea Islands family (Venus cores/8950/8970 GPGPU cards),

as the next Figure indicates.

1.3. Announcement and implementation of AMD’s GCN

7.2. ábra - Overview of GPGPU cores and cards and their basic software support (3)

1.4. AMD’s roadmap indicating the introduction of GCN along with the Southern Islands line (2/2012) [94]

7.3. ábra -

2. Major innovations of the Southern Islands microarchitecture

- AMD’s move from the VLIW4-based GPU design to the SIMD-based GPU design
- AMD ZeroCore Power technology

2.1. AMD’s move from the VLIW4-based GPU design to the SIMD-based GPU design
7.4. ábra - Evolution of the microarchitecture of ATI’s (acquired by AMD in 2006) and AMD’s GPUs [95]

2.1.1. Main steps of the evolution of the microarchitecture of ATI’s (acquired by AMD in 2006) and AMD’s GPU’s [95]

As the above Figure indicates, traditionally, AMD preferred to implement their GPUs based on VLIW ALUs, first on VLIW5 then on VLIW4 designs.

2.1.2. Principle of using VLIW-based ALUs in GPUs

In contrast to Nvidia, AMD based initially their GPUs on VLIW5 ALUs, that consist of 5 EUs and a Branch Unit, as shown below, whereas Nvidia preferred SIMD-based ALUs.

7.5. ábra - Block diagram of a VLIW5 ALU [17]
2.1.3. VLIW-based ALUs

- VLIW-based ALUs, like the one shown above are controlled by VLIW instructions that have multiple issue slots used to specify the operations for each EU.

- The operations issued at a time in a VLIW instruction need however, be free of dependencies.

- It is the responsibility of the compiler to generate a flow of VLIW instructions specifying as many dependency free operations for the issue slots as possible (static dependency resolution).

2.1.4. Example for encoding wavefront queues in form of VLIW4 instructions [96]

Let's assume that the compiler has to encode the following sequence of wavefronts, each with 64 work-items, with the given dependencies between wavefronts for execution as VLIW4 instructions.

7.6. ábra -

While using VLIW4 instructions the compiler packs operations from 4 wavefronts into the 4 slots of a VLIW4 instruction.

Each slot of the VLIW4 instruction is allocated for execution to one of four EUs of a 16-wide SIMD unit.

7.7. ábra - Block diagram of a VLIW4 ALU (introduced in the Northern Island line (HD6900 line)) [17]
7.8. ábra - Simplified block diagram of a 16-wide SIMD unit (based on [95])

2.1.5. Scheduling the wavefronts given previously while assuming VLIW4 encoding [96]

7.9. ábra -
Existing dependencies restrict the compiler from filling all four slots in each cycle, as indicated in the Figure.

The resulting encoding of the wavefronts into VLIW4 instructions is given in the Figure.

Note that scheduling all 64 work-items of the wavefront onto the 16-wide pipelined SIMD unit needs obviously 4 clock cycles each.

2.1.6. Graphics processing and VLIW-based ALUs

As early GPUs, discussed here were focused on graphics processing and graphics processing means the execution of fixed algorithms, compilers had a feasible task to generate a flow of VLIW instructions with high occupancy of the available issue slots.

To put it another way, graphics workloads do map well to VLIW-based GPU architectures [95].

2.1.7. AMD’s motivation for using VLIW-based ALUs in their previous GPU’s

• The motivation behind VLIW-based GPU architectures with static dependency resolution is to have less complexity due to the lack of hardware dynamic dependency resolution compared to traditional SIMD-based GPU architectures.

• As a consequence, for a given number of transistors, VLIW-based GPUs with static dependency resolution may devote more transistors for implementing EUs than SIMD-based GPUs with dynamic dependency resolution, thus VLIW-based GPUs have typically more EUs than SIMD-based GPUs.

The next Table demonstrates this for Nvidia’s SIMD-based GPUs using dynamic dependency resolution and AMD’s VLIW-based GPUs with static dependency resolution.

2.1.8. Number of execution units (EUs) in Nvidia’s and AMD’s GPUs

7.10. ábra -
2.1.9. AMD’s motivation for using VLIW5-based ALUs in their first GPU’s [97]

As far as the number of EUs in AMD’s VLIW ALUs concerns, AMD chose originally the VLIW5 design.

AMD made this decision in connection with DX9, as VLIW5 ALUs allow to calculate a 4 component dot product (e.g. for the RGBA color representation) and a scalar component (e.g. for lighting), often needed in the DX9 vertex shader, in parallel.

2.1.10. AMD’s VLIW5 design

In their R600 GPU core, introduced in 2007, AMD already made use of the VLIW5 design, nevertheless a bit differently than in later GPU designs, as the next Figure indicates.

7.11. ábra - Main steps of the evolution of AMD’s VLIW-based shader ALUs

<table>
<thead>
<tr>
<th>AMD</th>
<th>R600</th>
<th>R670</th>
<th>RV770</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nvidia</td>
<td>G80</td>
<td>GT200</td>
<td></td>
</tr>
<tr>
<td></td>
<td>90 nm</td>
<td>8800GTX</td>
<td>65 nm</td>
</tr>
<tr>
<td></td>
<td>880 nm</td>
<td>880GTX</td>
<td>55 nm</td>
</tr>
<tr>
<td></td>
<td>128 EU</td>
<td></td>
<td>240 EU</td>
</tr>
<tr>
<td></td>
<td>Peak FP32 MFLOPS</td>
<td>345</td>
<td>933</td>
</tr>
<tr>
<td></td>
<td>Peak FP64 MFLOPS</td>
<td>---</td>
<td>77.76</td>
</tr>
</tbody>
</table>

Remark
The introduction of VLIW5 design can even be traced back to ATI’s first GPU supporting the DX 9.0 set of graphics APIs, which was the Radeon 9700 GPU that was based on the R300 core), introduced about 2002 [101].

The R300 made use of the VLIW5 design in its programmable vertex shader pipeline [102].

7.12. ábra - The vertex shader pipeline in ATI’s first GPU supporting DX 9.0 (The R300 core) (2002) [102]

2.1.11. Replacing the original VLIW5 design by the Enhanced VLIW5 design

In their RV770 GPU core (Evergreen line), introduced in 2008, AMD replaced their original VLIW5 design by an Enhanced VLIW5 design to introduce FP64 MAD capability.

7.13. ábra - Main steps of the evolution of AMD’s VLIW-based shader ALUs
2.1.12. Replacing the Enhanced VLIW5 design by the VLIW4 design-1

7.14. ábra - Main steps of the evolution of AMD’s VLIW-based shader ALUs

Subsequently, in their Cayman GPU cores (Northern Islands line), introduced in 2010, AMD replaced also their Enhanced VLIW5 design by a VLIW4 design.

2.1.13. Replacing the Enhanced VLIW5 design with the VLIW4 design-2

The reason for replacing their VLIW5 design with a VLIW4 design revealed AMD at their Cayman launch (HD 6900/Northern Islands line) in 12/2010 saying that the average slot utilization rate of their VLIW5 architecture was only 3.4 out of 5 in gaming applications for DX10/11 shaders, i.e. on average the 5. EU remains unused [17].

In their new VLIW4 design AMD

• removed the T-unit (Transcendental unit),
• enhanced 3 of the new EUs such that these units together became capable of performing 1 transcendental operation per cycle as well as
• enhanced all 4 EUs to perform together an FP64 operation per cycle.

7.15. ábra - Block diagram of the VLIW4 ALU introduced in the Northern Island line (HD6900 line) [17]

2.1.14. Contrasting the main features of AMD’s Enhanced VLIW5 and VLIW4 designs [17]
Enhanced VLIW5 design

*RV770 core/HD4870 card, (2008)*

*Evergreen line (RV870 core/HD5870 card), (2009)*

- 5 FX32 or 5 FP32 operations or
- 1 FP64 operation or
• 1 transcendental + 4 FX/FP 32 operations per cycle.

VLIW4 design

Northern Island line (Barts core/HD 6870), (2010)

• 4 FX32 or 4 FP32 operations or
• 1 FP64 operation or
• 1 transcendental + 1 FX32 or 1 FP32 operation per cycle.

Remark

Benefits of replacing VLIW5 ALUs with VLIW4 ALUs-1 [103]

In their VLIW4 design AMD

• removed the T-unit and
• enhanced 3 of the new EUs such that these units together became capable of performing 1 transcendental operation per cycle as well as
• enhanced all 4 EUs to perform together an FP64 operation per cycle.

The new design can compute now

• 4 FX32 or 4 FP32 operations or
• 1 FP64 operation or
• 1 transcendental + 1 FX32 or 1 FP32 operation per cycle, whereas

the previous design was able to calculate

• 5 FX32 or 5 FP32 operations or
• 1 FP64 operation or
• 1 transcendental + 4 FX/FP 32 operations per cycle.

Benefits of replacing VLIW5 ALUs with VLIW4 ALUs-2 [103]

• With removing the T-unit but enhancing 3 of the EUs to perform transcendental functions as well as all 4 EUs to perform together an FP64 operation per cycle after all about 10 % less floor space is needed compared with the previous design.

• The symmetric ALU design simplifies largely the scheduling task for the VLIW compiler,

• In addition, FP64 calculations can now be performed by a ¼ rate of FP32 calculations, rather than by a 1/5 rate as before.

2.1.15. Shift to numeric computing and AMD’s move to the GCN architecture

• As already discussed, AMD’s first GPUs were designed with graphics workloads in view.
• Since for graphics workloads VLIW architectures are favorable, AMD’s first GPUs were VLIW-based designs (based on prior ATI designs).

• Nevertheless, as time went on, compute workloads gained more and more impetus, and GPUs as well as data parallel accelerators came to widespread use on a wide range of application areas, including HPC, financial computing, mining, physics etc.

• In addition, numeric computation oriented graphics cards and data accelerators provide a much higher profit margin and growth potential than graphics cards [95].

• This is the reason why Nvidia, AMD and also Intel laid more and more emphasis on compute oriented devices, like Nvidia’s Tesla line, AMD’s FireStream/FirePro lines or Intel’s (Larrabee architecture (meanwhile cancelled) or their MIC, renamed later to Xeon Phi line.

• On the other hand, VLIW-based GPUs are less suited for compute workloads since compilers can not fill VLIW slots always appropriately for a wide spectrum of different algorithms used in HPC [95], so AMD made a further move and replaced their VLIW4-based GPU design by the new GCN architecture (South Islands line), as the subsequent Figure indicates.

2.1.16. Replacing the VLIW4-based design by SIMD-based design in AMD’s GCN architecture [91]

2.1.16.1. Overview of the evolution of the microarchitecture of AMD’s GPUs

7.17. ábra - 

<table>
<thead>
<tr>
<th>2006</th>
<th>2007</th>
<th>2008</th>
</tr>
</thead>
<tbody>
<tr>
<td>R600</td>
<td>R670</td>
<td>RV770</td>
</tr>
<tr>
<td>80 nm HD 2900 XT 320 EU</td>
<td>55 nm HD 3870 320 EU</td>
<td>55 nm HD 4870 800 EU</td>
</tr>
<tr>
<td>VLIW5 ALU</td>
<td>Enhanced ALU</td>
<td></td>
</tr>
<tr>
<td>Graphics workload one</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.2. AMD ZeroCore technology [49]

• For years both Nvidia and AMD introduced various techniques to reduce idle power of the graphics cards, like clock gating, power gating or frame buffer compression.

• Idle periods, may be short or long.

For idle periods that are longer than a preset value (long idle periods) the screen will be shut down.

In this case, however, usually the entire GPU can’t be powered off because the OS and the BIOS must remain aware of the presence of the card.

• AMD coped with this problem by the ZeroCore technology.

With the ZeroCore technology AMD introduced power island (separately supplied circuit parts), and for long idle periods the GPU will be shut down except of a small bus control block that lets the GPU remain visible for the OS and the BIOS.
2.2.1. Achieved reduction of power consumption in long idle periods in AMD’s GPUs due to using the AMD ZeroCore Power Technology [104]

7.18. ábra -

Remark

In addition to the AMD ZeroCore technology, introduced along with the Southern Islands line, AMD made use in this line also the AMD PowerTune technology that was introduced previously in their Northern islands line (HD 6000/Cayman).

2.2.2. The AMD PowerTune technology

Aim

• Applications with high power consumption, like Furmark or OCCT SC8, that were built basically for stress tests cause a potential threat for GPU overheating.

Without PowerTune, designers address this by leaving a high enough power headroom for a GPU to avoid a potential damage due to overheating.

This results however in a appropriately reduced clock speed.

• With PowerTune a microcontroller monitors and calculates the power consumption.

• Then a suitable algorithm will keep the GPU from exceeding the threshold (set by the GPU vendor) by reducing clock speed to avoid potential damage, as indicated in the next Figure.

In this way PowerTune allows to set a higher max. clock speed than without using this technique.

2.2.3. The PowerTune technology as introduced in the Northern Island line (HD 6900/Cayman), 12/2010) [106]

7.19. ábra -
2.2.4. Benefit of using the PowerTune technology [105]

7.20. ábra -
3. AMD’s Southern Islands card family

3.1. Overview of the Southern Islands family

Termed also as the GCN (Graphics Core Next) line

- It is AMD’s first GPU architecture towards the implementation of HSA
- Announced in 6/2011
- Launched in 12/2011

3.1.1. Announcement and implementation of AMD’s GCN

7.21. ábra - Overview of GPGPU cores and cards and their basic software support (3)
Note

After the first implemented cards (HD 7950/70) AMD launched also the higher clock versions HD7970 GE (Gigahertz Edition) and HD 7950 Boost).

3.1.2. Key features of the Southern Island Family

- New basic architecture (GCN)
- Accordingly a new ISA (GCN ISA)
- AMD ZeroCore power reduction technology
- First GPU built on 28 nm technology
- Supported by Windows 8
- First official support of PCIe 3.0
- First official support of DX11.1
- Support of OpenCL 1.2
- Three performance scaled subfamilies

3.1.3. Subfamilies of the Southern Island Family-1 [107]

7.22. ábra - Subfamilies of the "Southern Islands" Family [107]
3.1.4. Comparison of the die sizes of AMD’s and Nvidia’s graphics cores [95]

Note

While discussing the Southern Islands Family we restrict ourselves to its highest performance subfamily (Tahiti/HD 79xx line).

3.1.5. Announcement and implementation of AMD’s GCN

7.24. ábra - Overview of GPGPU cores and cards and their basic software support (3)
3.2. Microarchitecture of the Tahiti XT core used in the Radeon HD7970 card

3.2.1. Overall architecture of the Tahiti XT core [95]

7.25. ábra -

3.2.2. Difference between the cores of the 7950 (Tahiti Pro) and the 7970 (Tahiti XT) cards

The core of the HD 7970 has 32 Compute Units (CUs) (termed as GCN in the Figure) whereas in the core of the HD 7950 four CUs are disabled.

7.26. ábra - Block diagram of the HD 5950 (Tahiti Pro) [109]
3.2.3. Overall architecture of a Compute Unit (CU) – Simplified block diagram [95]

7.27. ábra -

3.2.4. Block diagram of the Compute Unit – Detailed block diagram [108]

7.28. ábra -
3.2.5. Main units of a CU

As the block diagram in the Figure shows each CU has the following main units:

- the Front End unit for fetching and decoding the instructions
- four 16-wide SIMD units
- a Scalar unit
- 16 KB L1 Instruction cache shared by 4 CUs
- 16 KB L1 Data cache per CU
- 64 KB LDS (Local Data Share) per CU, accessible for all four SIMD units
- 32 KB Scalar read-only constant cache, shared by 4 CUs

3.2.6. The four 16-wide SIMD units of the CU [110] (Strongly simplified discussion)

Used primarily for executing kernels.

As the previous Figure shows, each of the four 16-wide SIMD units (called SIMD0 to SIMD3) include a 64 KB Register File and a Vector MP ALU that includes 16 Execution units (EUs), as shown below.

7.29. ábra - Simplified block diagram of a 16-wide SIMD unit (based on [95])
• Each EU is pipelined and can perform 32- and 64-bit operations on FP, FX and Boolean data.

• The EUs take operands typically from the associated 64 KB sized register file.

• Each EU is capable to accept a new 32-bit FX, Boolean or FP operation, including FP MAD) in every new cycle.

• Each EU is capable to accept a new 64-bit FP operation, including FP MAD) in every 4th cycle.

• The pipelined EUs produce results in four cycles.

3.2.7. The Scalar unit of the CU [110]

• The Scalar unit can perform integer arithmetic and is primarily used for program flow control.

• It operates on a single value per clock cycle.

• The Scalar Unit takes operands typically from the associated 8 KB sized register file.

Remark
A detailed discussion of the operation of both the SIMD units and the Scalar unit can be found in [110].

3.2.8. Peak FP32 and FP64 performance figures of the HD 7970 (Tahiti XT)

3.2.8.1. a) Peak FP32 performance

With 4 SIMD units per CU, 16 EUs per SIMD unit and 32 CUs per GPU there are altogether 2048 EUs available per GPU.

At a core frequency of 925 MHz this results in a peak performance of

Then with 2 operations per MAD instructions the peak FP32 performance amounts to

3.2.8.2. b) Peak FP64 performance

The peak FP64 performance is 1/4th of the peak FP32 performance, that is

3.2.9. The cache hierarchy of the HD 7970 (Tahiti XT) [95]

7.30. ábra
3.2.10. Available data spaces for execution per CU and per GPU in the HD 7970 (Tahiti XT) [110]

7.31. ábra -

Remark

OpenCL has four memory domains: private, local, global and constant, and GPUs also recognize host (CPU) memory, as follows [16]:

3.2.11. Memory sizes and peak read bandwidth values in the HD 7970 (Tahiti XT) [16]

7.32. ábra -
Stream core means here EU

All sizes are given in Kbyte (designated here as k, e.g. 64 k means 64 KB)

### 3.3. Principle of operation (repeated from Section 3.2)

#### 3.3.1. Compute workloads

A compute workload for the GPU is to execute a kernel on up to 3-dimensional data structures, i.e. to perform computations in an N-Dimensional Range, like in the one shown in the Figure below.

7.33. ábra - The interpretation of an N-Dimensional Range [12]
3.3.2. Work items

Work-items are the basic units of computation, they typically represent index points in the data space.

A work-item, or more precisely the processing on a work item according to the flow of instructions may also be interpreted as a thread.

7.34. ábra - The interpretation of a work-item (Based on [12])
3.3.3. The kernel

The kernel is the data parallel program to be executed on the N-Dimensional Range. It consists of instructions, such as

- Vector instructions that instruct all 16 EUs of a SIMD unit,
- Scalar instructions that instruct the Scalar unit,
- Vector memory instructions that let transfer data (read or write data) between the GPU memory and the vector register file (VGPR) of a SIMD unit, etc.

Remark

In contrast to kernels, interpreted as compute oriented data parallel programs, shaders are graphics oriented programs to be executed on a GPU.

3.3.4. Work-groups

N-Dimensional Ranges are segmented by the programmer to work-groups, as indicated in the Figure, for efficient execution.

A work-group will be scheduled for processing to a particular CU.

7.35. ábra - The interpretation of workgroups [12]
3.3.5. Wavefronts-1

For efficient execution work-groups are segmented by the Ultra-Threaded Dispatcher to wavefronts that will be processed on the same CU.

7.36. ábra - The interpretation wavefronts [12]

Wavefronts belonging to the same work-group can share data and their run can be synchronized by an S_BARRIER instruction to force each wavefront to wait until all other wavefronts reach the same instruction.

3.3.6. Wavefronts-2

- Wavefronts represent the smallest unit of work that will be scheduled for execution to a CU.
- Wavefronts have a hardware specific size.

High performance models of the Southern Island lines, like the HD 7950/70 models (Tahiti Pro/XT) and some newer models have a wavefront length of 64 work-items, whereas some of the low-end and older models, such as the HD 54xx have a wavelength of 32 work-items.
• Each wavefront is assigned a single Program Counter (PC) that points to the next instruction to be executed. When the wavefront is created, the PC is initialized to the first instruction in the program.

Wavefronts are collections of work-items grouped together for efficient processing on the CU.

3.3.7. Principle of operation

The principle of operation of the HD 7970 (Tahiti XT) can be described based on the following block diagram of a South Islands series processor.

7.37. ábra - Block diagram of a Southern Island Series processor [110]

3.3.8. Launching compute workloads to the GPU

• Before an application passes a compute workload to the GPU, it must first compile the kernel and load it into the memory.

• It also must bind buffers to the source and result data.

• Finally, it creates a command queue in a command buffer, for instructing the GPU how to execute its computational workload.

• An application typically performs this by invoking a set of APIs.

Execution of compute workloads-1

3.3.9. Launching compute workloads to the GPU

• Before an application passes a compute workload to the GPU, it must first compile the kernel and load it into the memory.

• It also must bind buffers to the source and result data.
• Finally, it creates a command queue in a command buffer, for instructing the GPU how to execute its computational workload.

• An application typically performs this by invoking a set of APIs.

3.3.10. Execution of compute workloads-1

• Upon receiving the command to begin execution, the Command Processor begins interpreting the commands being in the command queue sequentially.

• It forwards related commands to the Ultra-Threaded Dispatch Processor.

• The Ultra-Threaded Dispatch Processor allocates resources needed for a kernel run, subdivides the N-Dimensional Range into wavefronts of 64 work-items (in case of the HD 7970) and dispatches work-groups and wavefronts to the CU array.

• The kernel is fetched into the instruction cache or instruction caches of the CUs that are commissioned to execute the workload.

(Note that each four CUs share a 16 KB instruction cache).

• Then the CUs begin fetching instructions from the instruction cache into their instruction buffers and dispatching instructions to their respective execution units (SIMD units, Scalar unit, memory system).

3.3.11. Execution of compute workloads-2

Each of the four SIMD units of a CU can work on different wavefronts of the same work-group in parallel.

Consequently, each CU has four active PCs, as illustrated in the next Figure.

7.38. ábra - Block diagram of a CU [110]

![Block diagram of a CU](image)

Altogether, the CU provides 4x10 instruction buffers, as shown in the above Figure.

3.3.12. Execution of compute workloads-3

A SIMD unit with 16 EUs needs obviously four clock cycles to accept all 64 work-items of a wavefront.

• The pipelines have four pipeline stages, i.e. there are four clock cycles needed to provide the results.
• New FX32 or FP32 data can be issued to the pipelined EUs in each cycle, whereas FP64 data will be accepted only every 4 cycles.

• The wavefront continues executing until the end of the kernel is reached, at which time the wavefront becomes terminated and a new one can take its place on the GPU.

3.4. Achieved performance figures and power consumption of the HD 7970 (Tahiti XT)

3.4.1. a) Gaming performance

At the end of 2012 AMD’s and Nvidia’s flagship graphics cards (the HD 7970 (Southern Islands/GCN) from AMD and the Kepler-based GTX 680 from Nvidia have about the same performance figures for gaming, as the next two Figures demonstrate it.

3.4.2. a) Gaming performance of high end graphics cards-1 [50]

7.39. ábra -

The GE tag means overlocked, by about 15 %

3.4.3. a) Gaming performance of high end graphics cards-2 [50]

7.40. ábra -
3.4.4. b) Performance figures of high end graphics cards measured for a compute workload [50]

7.41. ábra -
As the Figure below shows, at the end of 2012 the flagship graphics cards of AMD and Nvidia have approximately the same compute performance.

**3.4.5. c) Load power consumption (Power consumption at full load)** [50]

**7.42. ábra -**
As indicated, Nvidia’s GTX 680 has a lower power consumption at full load than AMD’s HD 7970.

3.4.6. d) Idle power consumption [50]

Owing to AMD’s ZeroCore Power Technology introduced, the idle power consumption of AMD’s HD 7970 could be reduced to about the same value as Nvidia’s GTX 680 has.

7.43. ábra -
<table>
<thead>
<tr>
<th>Total System Power Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD Radeon HD 7970</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 689</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 689</td>
</tr>
<tr>
<td>AMD Radeon HD 7970</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 690</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 690</td>
</tr>
<tr>
<td>AMD Radeon HD 7970</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 690</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 690</td>
</tr>
<tr>
<td>AMD Radeon HD 689</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 690</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 690</td>
</tr>
<tr>
<td>AMD Radeon HD 690</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 690</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 690</td>
</tr>
<tr>
<td>AMD Radeon HD 690</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 690</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 690</td>
</tr>
<tr>
<td>AMD Radeon HD 690</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 690</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 690</td>
</tr>
<tr>
<td>AMD Radeon HD 480</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 690</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 690</td>
</tr>
<tr>
<td>AMD Radeon HD 480</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 690</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 690</td>
</tr>
<tr>
<td>AMD Radeon HD 480</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 690</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 690</td>
</tr>
<tr>
<td>AMD Radeon HD 480</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 690</td>
</tr>
</tbody>
</table>
8. fejezet - AMD’s Sea Islands family

• It is AMD’s second GPU architecture towards the implementation of HSA.

1. The Sea Islands family on AMD’s Client and Graphics roadmap [111]

8.1. ábra -

2. Key features of the Sea Islands family

At revealing their client and graphics roadmap in 2/2012 AMD wasn’t specific what HSA features will be supported by the Sea Islands family [112].

8.2. ábra -
3. Expected introduction of AMD’s 2. generation GCN, termed as Sea Islands family

4. Key parameters of the HD 8950/70 graphics cards

Already revealed by AMD and summarized in Section 3.3.
9. fejezet - Outlook

In this Section we round up our presentation of GPGPUs by briefly discussing published graphics roadmaps of AMD and Nvidia and also give an overview to Intel’s newly introduced Xeon Phi line (previously designated as the MIC (Many Integrated Cores) family, targeting HPC.

1. Nvidia’s roadmap for 2014 and beyond

9.1. ábra -

1.1. The Maxwell CPU + GPU

It is scheduled for 2014.

Key features [115]:

- Maxwell is not solely a GPU but Nvidia integrates a CPU also on the same die.
  
  The CPU is an ARM v8-compatible 64-bit core (called Denver) with enhancements, not yet revealed.

- Maxwell supports unified virtual memory.
  
  It allows for both the GPU and the CPU to access system memory.

  This feature is similar as AMD’s Unified Memory, as part of their HAS (Heterogeneous System Architecture) and makes GPU programming easier.

As discussed in Section 6., AMD intends to introduce unified memory in 2013.

Nvidia expects three major benefits from their Maxwell processor, as indicated.

- Improved graphics capability,

- simplified programmability and

- energy efficiency.
1.2. The Volta CPU + GPU

The cited roadmap indicates that Nvidia’s subsequent processor is designated as Volta and it will have stacked DRAM, as shown in the next Figure.

1.3. The Volta processor with stacked DRAM [113]

9.2. ábra -

Remarks

1) Here we note that AMD showed in 2011 a prototype device with stacked memory, but until now no information is available about a commercial implementation by the firm [114].

2) On the other hand Intel did implement stacked memory in their Atom line, first in their Clover Trail tablet platform, called as PoP (Package-on-Package) memory, introduced in 12/2012 [116].

In the Clover Trail platform Intel implemented LPDDR2 memory as Package-on-Package (PoP) design that stacks the LPDDR2 memory chip on top of the SOC die, as shown in the Figure below.

The memory chip is placed on top of the SoC that is soldered on a shared package substrate.

9.3. ábra - Intel’s Package-on-Package memory stacking used in the Clover Trail tablet platform [116]

2. AMD’s graphics roadmaps for 2013

At writing these slides no graphics roadmaps could be found for 2014 on the internet.
The latest graphics roadmaps found refer to 2013, and do not indicate a new graphics card family beyond the Sea Islands (HD 8000) series, as shown below.

9.4. ábra - AMD’s 2013 notebook graphics roadmap [117]

![AMD 2013 NOTEBOOK GRAPHICS ROADMAP](image)

9.5. ábra - AMD’s 2013 desktop graphics roadmap [117]

![AMD 2013 DESKTOP OEM GRAPHICS ROADMAP](image)

3. Intel’s Xeon Phi line

Introduced in 5/2010 as the MIC (Many Integrated Core) DPA (Data Parallel Accelerator) line [118].

It was based mainly on Intel’s ill-fated Larrabee project and partly on results of their SCC (Single Cloud Computer) development.

9.6. ábra -
The line is targeting exclusively HPC and was implemented as add-on PCIe cards.

### 3.1. The Knights Ferry prototype version

In 5/2010 Intel introduced a prototype version of the MIC line termed as the Knight Ferry coprocessor, and made it available for developers [119].

### 3.2. Renaming the MIC branding to Xeon Phi and providing open source software support

In 6/2012 Intel renamed the MIC branding to Xeon Phi to emphasize the coprocessor nature of their DPAs and also to emphasize the preferred type of the companion processor.

At the same time Intel also made open source software support available for the Xeon Phi line, as indicated in the next Figure.

### 9.7. ábra - 

Overview of Intel’s Xeon Phi line (previously designated as the MIC line)
3.3. The Knights Corner consumer processor

In 5/2010 Intel announced also the Knights Corner consumer product that became available in 11/2012 [119]

3.4. Key features related to the Xeon Phi family [120]

Target application area
Highly parallel HPC workloads

Programming environment
It is a general purpose programming environment
• Runs under Linux
• Runs applications written in Fortran, C, C++, OpenCL 1.2 (in 2/2013 Beta)…

• Supports x86 memory model

• x86 design tools (libraries, compilers, Intel’s VTune, debuggers etc.)

3.5. First introduced subfamilies of the Xeon Phi coprocessor line [121]

9.10. ábra -

<table>
<thead>
<tr>
<th>SKU #</th>
<th>Form Factor, Thermal</th>
<th>Peak Double Precision</th>
<th>Cores</th>
<th>Clock Speed (Ghz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SE10P</td>
<td>PCIe Card, Passively Cooled</td>
<td>1073 GF</td>
<td>61</td>
<td>1.1</td>
</tr>
<tr>
<td>SE10X</td>
<td>PCIe Card, No Thermal Solution</td>
<td>1073 GF</td>
<td>61</td>
<td>1.1</td>
</tr>
<tr>
<td>S110P</td>
<td>PCIe Card, Passively Cooled</td>
<td>1011 GF</td>
<td>60</td>
<td>1.053</td>
</tr>
<tr>
<td>3100 Series</td>
<td>PCIe Card, Actively Cooled</td>
<td>&gt;1 TF</td>
<td>60</td>
<td>Disclosed at 3100 series</td>
</tr>
<tr>
<td></td>
<td>PCIe Card, Passively Cooled</td>
<td>&gt;1 TF</td>
<td>60</td>
<td>launch (H1’13)</td>
</tr>
</tbody>
</table>

Remark
The SE10P/X subfamilies are intended for customized products, like those used in supercomputers, such as the TACC Stampede, built in Texas Advanced Computing Center (2012).

9.11. ábra - Main features of Intel’s Xeon Phi line [122], [123]
### 3.6. System layout of the Knights Corner (KCN) DPA [120]

#### 9.12. ábra -

![System layout of the Knights Corner (KCN) DPA](image)

### 3.7. Microarchitecture of Knights Corner [120]

It is a bidirectional, ring based architecture like the predecessors Larrabee and Knights Ferry, with an increased number (60/61) of significantly enhanced Pentium cores and a coherent L2 cache built up of 256 kB/core segments, as shown below.
9.13. ábra - The microarchitecture of Knights Corner [120]

3.8. Layout of the ring interconnect on the die [122]

9.14. ábra -

3.9. Block diagram of a core of the Knights Corner [120]

9.15. ábra -
3.10. Block diagram and pipelined operation of the Vector unit [120]

9.16. ábra -

3.11. System architecture of the Xeon Phi coprocessor [122]

9.17. ábra -

EMU: Extended Math Unit

It can execute transcendental operations such as reciprocal, square root, and log, thereby allowing these operations to be executed in a vector fashion [120]
Remark

The System Management Controller (SMC) has three I2C interfaces to implement a thermal control and a status information exchange.

For details see the related Datasheet [122].

3.12. The Xeon Phi 5110P coprocessor [124]

9.18. ábra -
3.13. The Xeon Phi coprocessor board (backside) [122]

9.19. ábra -

3.14. Exploded view of an Xeon Phi 3100 with active cooling [122]

9.20. ábra -
3.15. Principle of Intel’s common software development platform for multicores, many-cores and clusters [125]

9.21. ábra -

3.16. The power efficiency of Intel’s Knight Corner vs. competition [120]

9.22. ábra -
3.17. Peak performance of the Xeon Phi 5110P and SE10P/X vs. a 2-socket Intel Xeon server [126]

9.23. ábra -

The reference system is a 2-socket Xeon server with two Intel Xeon E5-2670 processors (8 cores, 20 MB L3 cache, 2.6 GHz clock frequency, 8.0 GT/s QPI speed, DDR3 with 1600 MT/s).

3.18. Performance of Xeon Phi 5110P vs. a 2-socket Sandy Bridge based Xeon server [126]

9.24. ábra -
3.19. Assessing the achieved speed-up of the Xeon Phi 5110P vs a 2-socket Xeon server

As indicated in the above Figure, the typical speed-up of about 2 – 4 achieved for HPC is not to much convincing.

Remark

The reference system is a 2-socket Xeon server with two Intel Xeon E5-2670 processors (8 cores, 20 MB L3 cache, 2.6 GHz clock frequency, 8.0 GT/s QPI speed, DDR3 with 1600 MT/s).

3.20. Intel’s roadmap for the Xeon Phi line [127]
3.21. Assessing the peak FP64 performance of the Knights Corner coprocessor with that of Nvidia’s and AMD’s recent devices

It is worth comparing Knights Corner’s peak FP64 performance with the peak FP64 performance of Nvidia’s and AMD’s devices, shown in the next slides (taken from the Sections 5.6 and 7.2, respectively.

3.22. d) FP64 performance increase in Nvidia’s Tesla and GPGPUs

Performance is bounded by the number of available DP FP execution units.

9.26. ábra -
3.23. Number of execution units (EUs) in Nvidia’s and AMD’s GPUs

As a comparison shows all three up to date flagship GPGPU devices of Nvidia, AMD and Intel provide approximately the same peak FP64 performance of about 1 TFLOPs, so at the time being there is no clear performance winner among them.
10. fejezet - References


[38] NVIDIA G80: Architecture and GPU Analysis, Beyond3D, Nov. 8 2006, http://www.beyond3d.com/content/reviews/1/11
References


References


[88] Heterogeneous System Architecture and the HSA Foundation, AMD Fusion Developer, Summit, June 2012


References


[117] AMD Product and Technology Roadmaps, 2013


II. rész - GPGPUs and their programming
GPGPUs and their programming

1.1. New project wizard ................................................................. 270
1.2. New project wizard ................................................................. 270
1.3. Custom build rules ................................................................. 271
1.4. CUDA related project properties ............................................. 272

2. Compute capabilities .............................................................. 273
   2.1. Compute capability (1) ....................................................... 273
       2.1.1. Details for hardware versions .................................... 273
   2.2. Compute capability (2) ....................................................... 273
   2.3. Compute capability (3) ....................................................... 274
   2.4. Compute capability (4) ....................................................... 275
   2.5. Compute capability (5) ....................................................... 275
   2.6. Device parameters (1) ....................................................... 275
   2.7. Device parameters (2) ....................................................... 276

3. CUDA language extensions ...................................................... 276
   3.1. CUDA language extensions ................................................ 276
       3.2. Common component – new variable types ......................... 277
           3.2.1. Built-in vector types .............................................. 277
           3.2.2. dim3 type ............................................................ 277
       3.3. Common component – available functions ........................... 277
           3.3.1. Mathematical functions .......................................... 277
           3.3.2. Time functions ...................................................... 277
       3.4. Device component - built-in variables ............................... 278
           3.4.1. gridDim .............................................................. 278
           3.4.2. blockIdx ............................................................ 278
           3.4.3. blockDim ............................................................ 278
           3.4.4. threadIdx ............................................................ 278
           3.4.5. warpSize .............................................................. 278
       3.5. Device component - functions .......................................... 278
           3.5.1. Fast mathematical functions .................................... 278
           3.5.2. Synchronization within a block ................................ 278
       3.6. Device component – atomic functions ............................... 279
           3.6.1. Warp vote functions ............................................... 279
       3.7. Host component - functions ............................................ 279
           3.7.1. Error handling ...................................................... 279

4. Asynchronous Concurrent Execution ......................................... 280
   4.1. Streams .......................................................... 280
   4.2. Creating/destroying streams .............................................. 280
   4.3. Using streams .......................................................... 281
   4.4. Using streams example .................................................... 281
   4.5. Stream synchronization .................................................... 282
   4.6. Operations implicitly followed a synchronization .................. 282
   4.7. Stream scheduling [12] .................................................... 283
   4.8. Concurrency support ...................................................... 283
   4.9. Blocked Queue example ................................................... 283
   4.10. Blocked Kernel example .................................................. 284

5. CUDA events .......................................................... 284
   5.1. Create and destroy a new event ......................................... 284
   5.2. Record an event .......................................................... 285
   5.3. Synchronize an event ...................................................... 286
   5.4. Check an event .......................................................... 286
   5.5. Synchronization with events ............................................. 287
   5.6. Synchronization with events (example) ............................... 287
   5.7. Calculate elapsed time between two events ........................... 288
   5.8. Calculate elapsed time (example) ...................................... 288

6. Unified Virtual Address Space ........................................... 289
   6.1. CUDA Unified Virtual Address Management ........................... 289
   6.2. Unified Virtual Address Space .......................................... 289
   6.3. Unified Virtual Address Space – check availability ............... 290
   6.4. Peer to peer communication between devices ........................ 290
   6.5. Using Unified Addressing and P2P transfer .......................... 291
6.6. Peer-to-peer memory transfer between GPUs ........................................ 291
6.7. Peer-to-peer memory access between GPUs ....................................... 292
6.8. Peer-to-peer memory access kernel ..................................................... 293
6.9. CUDA Unified Virtual Address summary ............................................ 293
6.9.1. More information about UVA ......................................................... 293

4. Optimization techniques ................................................................. 295
1. Using shared memory ........................................................................... 295
   1.1. Optimization strategies ................................................................. 295
   1.2. Matrix multiplication .................................................................. 295
       1.2.1. Exam 4.1.1 ......................................................................... 295
   1.3. Multi-dimensional matrix in global memory .................................... 296
       1.3.1. Access elements of a multi-dimensional array ......................... 296
   1.4. Multi-dimensional matrix in the GPU memory .............................. 297
   1.5. Multi-dimensional matrix in the GPU memory .............................. 297
   1.6. Aligned arrays ............................................................................ 298
   1.7. Access elements in case of aligned storage .................................. 299
   1.8. Aligned memory management ..................................................... 299
   1.9. Copy aligned memory .................................................................. 299
   1.10. Matrix multiplication with aligned arrays .................................... 300
       1.10.1. Exam 4.1.2 ........................................................................ 300
   1.11. Kernel with aligned arrays ............................................................ 300
   1.12. Invoke kernel with aligned arrays ............................................... 301
   1.13. Using shared memory .................................................................. 301
       1.13.1. Available solutions ............................................................... 302
   1.14. Tiled matrix multiplication ........................................................... 302
   1.15. Matrix multiplication .................................................................. 302
       1.15.1. Exam 4.1.3 ........................................................................ 302
   1.16. Optimized matrix multiplication .................................................. 303
   1.17. Optimized matrix multiplication (2) ............................................. 303
   1.18. Optimized matrix multiplication (3) ............................................. 304
   1.19. Optimized matrix multiplication (4) ............................................. 305
   1.20. Optimized matrix multiplication (5) ............................................. 306
   1.21. Optimized matrix multiplication source code .............................. 307
   1.22. Comparing runtime of original and tiled algorithms ...................... 308

2. Using atomic instructions ..................................................................... 308
   2.1. Atomic operations ......................................................................... 308
       2.1.1. Possible solutions ................................................................. 308
   2.2. CUDA atomic instructions .............................................................. 308
       2.2.1. Performance notes ................................................................. 309
   2.3. CUDA atomic instructions .............................................................. 309
   2.4. CUDA atomic instructions (2) ....................................................... 309
   2.5. CUDA atomic bitwise instructions .................................................. 310
       2.5.1. Exam 4.2.1 ........................................................................ 310
   2.6. Find the minimal value of a vector – using global memory .......... 310
       2.6.1. Exam 4.2.2 ........................................................................ 310
   2.7. Find the minimal value of a vector – using shared memory .......... 310
       2.7.1. Comparing runtime of global and shared memory usage .......... 311
   2.8. Parallelization inside the block ....................................................... 312
       2.8.1. Exam 4.2.2 ........................................................................ 312
   2.9. Parallel minimum search - loading ............................................... 312
   2.10. Parallel minimum search – find minimum of block ...................... 313
   2.11. Parallel minimum search - kernel .................................................. 314
   2.12. Parallel minimum search – kernel (2) .......................................... 314
   2.13. Comparing runtime of atomic and parallel version ...................... 314
   2.14. Comparing of CPU and GPU implementation ............................. 315

3. Occupancy considerations ................................................................. 315
   3.1. Execution overview ....................................................................... 315
   3.2. Occupancy .................................................................................... 316
   3.3. Occupancy and registers ............................................................... 316
   3.4. Occupancy and shared memory ..................................................... 317
1. Introduction (1)

1.1. Computational power of GPUs

- GPUs have enormous computational power (mainly in the field of single precision arithmetic)

1.1. ábra - [11]

2. Introduction (2)

2.1. Real-world applications

- GPU computing applications developed on the CUDA architecture by programmers, scientists, and researchers around the world.

- See more at CUDA Community Showcase

1.2. ábra -
2.2. Graphical Processing Units

• A Graphical Processing Unit (GPU) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the building of images in a frame buffer intended for output to a display [1].

• Modern GPUs are very efficient at manipulating computer graphics, especially in 3D rendering. These functions are usually available through some standard APIs, like:
  • OpenGL (www.opengl.org)
  • Direct3D (www.microsoft.com)

2.2.1. Shaders

• Shader is a computer program or a hardware unit that is used to do shading (the production of appropriate levels of light and darkness within an image) [2]

• Older graphics cards utilize separate processing units for the main tasks:
  • Vertex shader – the purpose is to transform each vertex’s 3D position in the virtual space to a 2D position on the screen
  • Pixel shader – the purpose is to compute color and lightness information for all pixels in the screen (based on textures, lighting etc.)
  • Geometry shader – the purpose is to generate new primitives and modify the existing ones

2.3. Unified Shader Model

• Older graphics cards utilize separate processing units for each shader type

• It’s hard to optimize the number of the different shaders because different tasks need different shaders:
  • Task 1.:
    geometry is quite simple
    complex light conditions
  • Task 2.:
geometry is complex

texturing is simple

1.3. ábra - [3]

2.3.1. Unified Shader

• Later shader models reduced the differences between the physical processing units (see SM 2.x and SM 3.0)

• Nowadays graphics cards are usually contains only one kind of processing units which is capable for every tasks. These are flexibly schedulable to a variety of tasks

• The Unified Shader Model uses a consistent instruction set across all shade types. All shaders have almost the same capabilities – they can read textures, data buffers and perform the same set of arithmetic instructions [4]

2.4. What is GPGPU

• The Unified Shader Model means that the GPU use the same processor core to implement all functions. These are simple, processing units with a small set of instructions.

• Therefore graphics card manufacturers can increase the number of execution units. Nowadays a GPU usually have ~1000 units.

• Consequently GPUs have massive computing power. It’s worth to utilize this computing power not only in the area of computer graphics:

  **GPGPU**: General-Purpose Computing on Graphics Processor Units

2.4.1. Programmable graphics cards

• In the first time it was a hard job to develop software components for graphics cards. The developer had to use the direct language of the shaders.

• Nowadays the graphics card manufacturers support the software developers with convenient development frameworks:

  • Nvidia CUDA
  
  • ATI Stream
  
  • OpenCL
2.5. GPGPU advantages

- Outstanding peak computing capacity
- Favorable price/performance ratio
- Scalable with the ability of multi-GPU development
- Dynamic development (partly due to the gaming industry)

2.5.1. GPU disadvantages

- Running sequential algorithms on GPUs is not efficient
  → we have to implement a parallel version but it is not a trivial task (and not always worth it: calculating factorial, etc.)

- GPU execution units are less independents than CPU cores
  → the peak performance is available only in some special (especially data parallel) tasks

- Graphics cards have a separated memory region and GPUs can not access the system memory. Therefore we usually need some memory transfers before the real processing
  → we have to optimize the number of these memory transfers. In some cases these transfers make unusable the whole GPU solution

- GPGPU programming is a new area, therefore the devices are less mature, the development time and cost is significantly higher

2.6. CPU-GPGPU comparison

- It is visible in Figure 1.2, that in the case of CPUs, most of the die area is used by the cache. In case of GPUs, the amount of cache memory is minimal, most of the die area is used by the execution units

- To improve the execution efficiency, GPUs employ a very useful feature: latency hiding. A load from device memory takes hundreds of cycles to complete (without cache). During this interval, instructions dependent on fetched values will block the thread. Utilizing the fast context-switching feature, the execution units can start working in other threads
  → to utilize this feature, the number of threads must be greater than the number of execution units

1.4. ábra - [5]

2.7. Memory architecture
• In case of CPUs, we usually don’t care about the memory architecture, we use only the global system memory and registers

• In practice there are some other memory levels (different kind of cache memories), but the CPU automatically handle these

• In case of GPUs the developer must know the whole memory architecture

→ sometimes it’s worth to load the often requested variables to some faster memory areas (manually handling the cache mechanism)

1.5. ábra - [3]
2.8. SIMT execution

- Sources of parallelism (SIMD < SIMT < SMT) [25]
  - In SIMD, elements of short vectors are processed in parallel
  - In SMT, instructions of several threads are run in parallel
  - SIMT is somewhere in between - an interesting hybrid between vector processing and hardware threading
  - In case of the well known SIMD commands, the developer must ensure that all the operands will be in the right place and format. In case of SIMT execution, the execution units can reach different addresses in the global memory
  - It is possible to use conditions with SIMT execution. The branches of the condition will be executed sequentially:

→ Try to avoid conditions and cycles in GPU codes

1.6. ábra - [7]
2. fejezet - Programming model

1. Basics of CUDA environment

1.1. CUDA environment

- CUDA (Compute Unified Device Architecture) is the compute engine in Nvidia graphics processing units or GPUs, that is accessible to software developers through industry standard programming languages
- Free development framework, downloadable for all developers
- Similar to C / C++ programming languages

1.1.1. Releases

- 2007. June. – CUDA 1.0
- 2008. Aug. – CUDA 2.0
- 2010. March. – CUDA 3.0
- 2011. May – CUDA 4.0
- 2012. Oct. – CUDA 5.0

1.1.2. Supported GPUs

- Nvidia GeForce series
- Nvidia GeForce mobile series
- Nvidia Quadro series
- Nvidia Quadro mobile series
- Nvidia Tesla series

1.2. Requested components

- Appropriate CUDA compatible Nvidia graphics driver
- CUDA compiler
  To compile .cu programs
- CUDA debugger
  To debug GPU code
- CUDA profiler
  To profiling GPU code
- CUDA SDK
  Sample applications, documentation

2.1. ábra -
1.2.1. Download CUDA

- CUDA components are available from:

1.3. CUDA platform overview

- The CUDA language is based on the C/C++ languages (host and device code), but there are other alternatives (Fortran etc.)
- The CUDA environment contains some function libraries that simplify programming (FFT, BLAS)
- Hardware abstraction mechanism hides the details of the GPU architecture
  - It simplifies the high-level programming model
  - It makes easy to change the GPU architecture in the future

1.3.1. Separate host and device code

- Programmers can mix GPU code with general-purpose code for the host CPU
- Common C/C++ source code with different compiler forks for CPUs and GPUs
- The developer can choose the compiler of the host code

2.2. ábra - [5]
1.4. Parts of the CUDA programming interface

1.4.1. C language extensions

- A minimal set of extensions to the C language, that allow the programmer to target portions of the source code for execution on the device
  - function type qualifiers to specify whether a functions executes on the host or on the device and whether it is callable from the host or from the device
  - variable type qualifiers to specify the memory location on the device of a variable
  - a new directive to specify how a kernel is executed on the device from the host
  - built-in variables that specify the grid and block dimensions and the block and thread indices

1.4.2. Runtime library

- The runtime library split into:
  - a host component, that runs on the host and provides functions to control and access the compute devices
  - a device component, that runs on the device and provides device-specific functions
  - a common component, that provides built-in types, and a subset of the C library that are supported in both host and device code

1.5. CUDA software stack

- The CUDA software stack is composed of several layers as illustrated in Figure 4.2.1:
  - device driver
  - application programming interface (API) and it’s runtime
• additional libraries (two higher-level mathematical libraries of common usage)

• Programmers can reach all the three levels depending on simplicity/efficiency requirements

• It’s not recommend to use only one of these levels in one component

• In these lessons we will always use the “CUDA Runtime” level. In this level we can utilize the features of the GPU (writing/executing kernels etc.) and the programming is quite simple.

2.3. ábra - [5]

1.6. Main steps of the CUDA development

• Analysis of the task

• Implement the C/C++ code

• Compile/link the source code

1.6.1. Analyzing of the task

• Unlike traditional programs in addition to selecting the right solution we have to find the well parallelizable parts of the algorithm

• The ratio of parallelizable/nonparallelizable parts can be a good indicator that it is worth to create a parallel version or not
• Sometimes we have to optimize the original solution (decrease the number of memory transfers/kernel executions) or create an entirely new one

1.6.2. Implementing the C/C++ code

• In practice we have only one source file, but it contains both the CPU and the GPU source code:
  • Sequential parts for the CPU
  • Data Parallel parts for the GPU

1.6.3. Compiling and linking

• The CUDA framework contains several utilities, therefore the compiling and linking means only the execution of the ncc compiler

2. Compiling and linking

2.1. CUDA compilation process details

2.1.1. Input

• One source file contains the CPU and GPU codes (in our practice in C/C++ language)

2.4. ábra - [2]
2.1.2. Compilation

- The EDG preprocessor parses the source code and creates different files for the two architectures

- For the host CPU, EDG creates standard .cpp source files, ready for compilation with either the Microsoft or GNU C/C++ compiler

- For Nvidia’s graphics processors, EDG creates a different set of .cpp files (using Open64)

2.1.3. Output
• The output can be an object file, a linked executable file, .ptx code etc..

2.2. Main parameters of the nvcc compiler (1)

2.2.1. Usage of the compiler

• Default path (in case of x64 Windows installation):
  \c:\CUDA\bin64\nvcc.exe

• Usage:
  \nvcc [options] <inputfile>

2.2.2. Specifying the compilation phase:

• --compile(-c)
  Compile each .c/.cc/.cpp/.cxx/.cu input file into an object file

• --link(-link)
  This option specifies the default behavior: compile and link all inputs

• --lib(-lib)
  Compile all inputs into object files (if necessary) and add the results to the specified output library file

• --run(-run)
  This option compiles and links all inputs into an executable, and executes it

• --ptx(-ptx)
  Compile all .cu/gpu input files to device-only .ptx files. This step discards the host code for each of these input file

2.3. Main parameters of the nvcc compiler (2)

2.3.1. Setting directory information

• --output-directory <directory>(-odir)
  Specify the directory of the output file

• --output-file <file>(-o)
  Specify name and location of the output file. Only a single input file is allowed when this option is present in nvcc non-linking/archiving mode

• --compiler-bindir <directory> (-ccbin)
  Specify the directory in which the compiler executable (Microsoft Visual Studio cl, or a gcc derivative) resides. By default, this executable is expected in the current executable search path

• --include-path <include-path>(-I)
  Specify include search paths

• --library <library>(-l)
Specify libraries to be used in the linking stage. The libraries are searched for on the library search paths that have been specified using option '-L'.

- --library-path <library-path>(-L)

Specify library search paths

2.4. Main parameters of the nvcc compiler (3)

2.4.1. Options for steering GPU code generations>

- --gpu-name <gpu architecture name> (-arch)

Specify the name of the NVIDIA GPU to compile for. This can either be a 'real' GPU, or a 'virtual' ptx architecture. The architecture specified with this option is the architecture that is assumed by the compilation chain up to the ptx stage.

Currently supported compilation architectures are: virtual architectures compute_10, compute_11, compute_12, compute_13, compute_20, compute_30, compute_35; and GPU architectures sm_10, sm_11, sm_12, sm_13, sm_20, sm_21, sm_30, sm_35

- --gpu-code <gpu architecture name> (-code)

Specify the name of NVIDIA GPU to generate code for. Architectures specified for options -arch and -code may be virtual as well as real, but the 'code' architectures must be compatible with the 'arch' architecture. This option defaults to the value of option '-arch'.

Currently supported GPU architectures: sm_10, sm_11, sm_12, sm_13, sm_20, sm_21, sm_30, and sm_35

- --device-emulation(-deviceemu)

Generate code for the GPGPU emulation library

2.5. Main parameters of the nvcc compiler (4)

2.5.1. Miscellaneous options for guiding the compiler driver:

- --profile (-pg)

Instrument generated code/executable for use by gprof (Linux only)

- --debug (-g)

Generate debug information for host code.

- --optimize<level>(-O)

Specify optimization level for host code

- --verbose(-v)

List the compilation commands generated by this compiler driver, but donot suppress their execution

- --keep (-keep)

Keep all intermediate files that are generated during internal compilationsteps

- --host-compilation <language>

Specify C vs. C++ language for host code in CUDA source files.

Allowed values for this option: 'C','C++','c','c++'.
Default value: 'C++'

2.6. Compiling example

2.6. ábra -

```
C:\CUDA\bin64\nvcc.exe
  -ccbin "C:\Program Files (x86)
  -I"C:\CUDA\include"
  -I"C:\Program Files (x86)\Mic
  -I"C:\Program Files\NVIDIA Co
  -L"C:\Program Files (x86)\Mic
  --host-compilation C++
  --link
  --save-temps
  "d:\hallgato\CUDA\sample.cu"
```

2.7. Overview of compilation

2.6. ábra -

3. Platform model

3.1. CUDA platform model

- As visible in Figure 3.1.1 the CUDA environment assumes that all the threads are executed in a separate device
Therefore we have to separate the host machine (responsible for memory allocations, thread handling) and the device (responsible for the execution of the threads)

3.1.1. Asynch execution

- With multiple devices one host can control more than one CUDA device
- In case of Fermi and later cards, one device can run parallel more than one thread groups
- In case of Kepler and later cards, any kernel can start other kernels

2.7. ábra - [5]
C Program
Sequential Execution

Serial

Parallel
Kernel0

Serial

Parallel
Kernel1
3.2. Inside one CUDA device

- Figure 3.1.2 illustrates the CUDA hardware model for a device
- Every device contains one or more multiprocessors, and these multiprocessors contain one or (more frequently) more SIMT execution units

3.2.1. Inside one multiprocessor

- SIMT execution units
- Registers
- Shared memory (available for all threads)
- Read-only constant and texture cache

2.8. ábra - [5]
3.3. Device management

3.3.1. Number of CUDA compatible devices

- The result of the `cudaGetDeviceCount` function is the number of CUDA-available devices

```
1  int deviceCount;
2  cudaGetDeviceCount(&
```
• The function will store the number of CUDA compatible devices into the passed deviceCount variable

1. int deviceCount;

2. cudaGetDeviceCount(&deviceCount);

3.3.2. Select the active CUDA compatible device

• This function is used to select the device associated to the host thread. A device must be selected before any __global__ function or any function from the runtime API is called

• The parameter of this function is the number of the selected device (numbering starts with 0)

2.10. ábra -

```cpp
text
int deviceNumber = 0;
cudaSetDevice(deviceN)
```

• Missing the function call, the framework will automatically select the first available CUDA device

• The result of the function will affect the entire host thread

3.4. Detailed information about devices

• The CUDA framework contains a class structure named `cudaDeviceProp`, to store the detailed information of the devices. The main fields of this structure are:

2.11. ábra -

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>name</code></td>
<td>Name of the device</td>
</tr>
<tr>
<td><code>totalGlobalMem</code></td>
<td>Size of the total global memory</td>
</tr>
<tr>
<td><code>sharedMemPerBlock</code></td>
<td>Size of shared memory per block</td>
</tr>
<tr>
<td><code>regsPerBlock</code></td>
<td>Number of registers per block</td>
</tr>
<tr>
<td><code>totalConstMem</code></td>
<td>Size of the total constant memory</td>
</tr>
<tr>
<td><code>warpSize</code></td>
<td>Size of the warp size</td>
</tr>
<tr>
<td><code>maxThreadsPerBlock</code></td>
<td>Maximum number of threads per block</td>
</tr>
<tr>
<td><code>maxThreadsDim</code></td>
<td>Maximum number of threads per dimension</td>
</tr>
<tr>
<td><code>maxGridSize</code></td>
<td>Maximum number of grid size</td>
</tr>
<tr>
<td><code>clockRate</code></td>
<td>Clock frequency</td>
</tr>
<tr>
<td><code>minor, major</code></td>
<td>Version</td>
</tr>
<tr>
<td><code>multiprocessorCount</code></td>
<td>Number of multiprocessors</td>
</tr>
<tr>
<td><code>deviceOverlap</code></td>
<td>Is the device overlapped</td>
</tr>
</tbody>
</table>

3.5. Acquire the detailed information about devices

• The result of the `cudaGetDeviceProperties` is the previously introduced `cudaDeviceProp` structure.

• The first parameter of the function is a pointer to an empty `cudaDeviceProp` structure. The second parameter is the identifier of the device (numbering starts with 0)
3.5.1. Exam 2.3.1

Write out the number of available devices.

List the number of these devices.

List the detailed data of an user selected device.

4. Memory model

4.1. The memory concept

- Thread level
  - Private registers (R/W)
  - Local memory (R/W)

- Block level
  - Shared memory (R/W)
  - Constant memory (R)

- Grid level
  - Global memory (R/W)
  - Texture memory (R)

4.1.1. Device-host communication

- The global, constant and texture memory spaces can be read from or written to by the CPU and are persistent across kernel launches by the same application

2.12. ábra - []

```cpp
int deviceNumber = 1;
cudaDeviceProperty(dev, cudaGetDeviceProperties);
```
4.2. CUDA memory model – global memory

- Has the lifetime of the application
- Accessible for all blocks/threads
- Accessible for the host
- Readable/writeable
- Large
- Quite slow

4.2.1. Declaration

- Use the `__device__` keyword

Example:

2.14. ábra -

```
1 | __d
2 | __d
```

2.15. ábra - [5]
4.3. CUDA memory model – constant memory

- Has the lifetime of the application
- Accessible for all blocks/threads
- Accessible for the host
- Readable/writeable for the host
- Readable for the device
- Cached

4.3.1. Declaration

- Use the \texttt{\_constant\_} keyword

- Example:

2.16. ábra -

2.17. ábra - [5]
4.4. CUDA memory model – texture memory

- Has the lifetime of the application
- Accessible for all blocks/threads
- Accessible for the host
- Readable/writeable for the host
- Readable for the device
- Available for image manipulating functions (texturing etc.). Not a common byte based array.

4.4.1. Declaration

- We do not discuss

2.18. ábra - [5]
4.5. CUDA memory model – shared memory

- Has the lifetime of the block
- Accessible for all threads in this block
- Not accessible for the host
- Readable/writeable for threads
- Quite fast
- Size is strongly limited (see kernel start)

4.5.1. Declaration

- Use the `__shared__` keyword
- Example:

2.19. ábra -

```
1  __S
2  __S
```

- Dynamic allocation example
4.6. CUDA memory model - registers

- Has the lifetime of the thread
- Accessible for only the owner thread
- Not accessible for the host/other threads
- Readable/writeable
• Quite fast
• Limited number of registers
• Not dedicated registers, the GPU have a fixed size register set

4.6.1. Declaration

• Default storing area for device variables
• Example

2.22. ábra -

4.7. CUDA memory model – local memory

• Has the lifetime of the thread
• Accessible for only the owner thread
• Not accessible for the host/other threads
• Readable/writeable
• Quite slow

4.7.1. Declaration

• Looks like a normal register, but these variables are stored in the „global” memory
• If there aren’t any space for registers, the compiler will automatically create the variables in local memory
• Example

2.24. ábra -

2.25. ábra - [5]

4.8. Physical implementation of the CUDA memory model

4.8.1. Dedicated hardware memory

• The compiler will map here the
  • registers,
• shared memory
• ~1 cycle

4.8.2. Device memory without cache

• The compiler will map here the
  • local variables,
  • global memory
• ~100 cycle

2.26. ábra - [5]
4.8.3. Device memory with cache

- The compiler will map here the
  - constant memory,
  - texture memory,
  - instruction cache
- ~1-10-100 cycle

4.9. Memory handling
4.9.1. Static allocation

- Variables declared as usual in C languages
- The declaration contains one of the previously introduced keywords (__device__, __constant__ etc.)
- The variable is accessible as usual in C languages, we can use them as operands and function parameters etc.

4.9.2. Dynamic allocation

- The CUDA class library have several memory handling functions. With these functions we can
  - allocate memory
  - copy memory
  - free memory

- The memory is accessible via pointers

- Pointer usage is the same as common in C languages but it is important to note that the device have a separated address space (device and host memory pointers are exchangeable)

4.10. CUDA memory regions

4.10.1. Grouped by visibility

2.27. ábra -

4.10.2. Grouped by accessibility

2.28. ábra -

<table>
<thead>
<tr>
<th></th>
<th>Global</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>Dynamic allocation</td>
<td>Dynamic allocation</td>
</tr>
<tr>
<td></td>
<td>R/W</td>
<td>R/W</td>
</tr>
<tr>
<td>Eszköz</td>
<td>-</td>
<td>Static allocation</td>
</tr>
<tr>
<td></td>
<td>R/W</td>
<td>R</td>
</tr>
</tbody>
</table>

4.11. Dynamic allocation – allocate memory

- Programmer can allocate and deallocate linear memory with the appropriate functions in the host code
- The cudaMalloc function allocates device memory, parameters:
• address of a pointer to the allocated object
• size of allocated object (byte)
• For example, to allocate a float vector with size 256:

2.29. ábra -

```c
float *devPtr,
cudaMalloc(void)
```

4.11.1. Free device memory
• Programmer can free allocated device memory regions with the `cudaFreeArray` function
• The only parameter of the function is a pointer to the object

2.30. ábra -

```c
float *dev,
cudaFree()
```

4.12. Transfer in device memory
• Programmer can copy data between the host and the devices with the `cudaMemCopy` function
• Required parameters:
  • destination pointer
  • source pointer
  • number of bytes to copy
  • direction of memory transfer
• Valid values for direction
  • host → host
    
    `(cudaMemcpyHostToHost)`
  • host → device
    
    `(cudaMemcpyHostToDevice)`
  • device → host
    
    `(cudaMemcpyDeviceToHost)`
  • device → device
    
    `(cudaMemcpyDeviceToDevice)`

2.31. ábra -
2.32. ábra - [5]

4.13. Pinned memory

- In the host side we can allocate pinned memory. This memory object is always stored in the physical memory, therefore the GPU can fetch it without the help of the CPU.

- Non-pinned memory can stored in swap (in practice in the hard drive) therefore it can cause page faults on access. So the driver needs to check every access.

- To use asynchronous memory transfers the memory must be allocated by the special CUDA functions:
  - cudaHostAlloc()
  - cudaFreeHost()

- It has several benefits:
  - Copies between pinned memory and device memory can be performed concurrently with kernel execution for some devices
  - Pinned memory can be mapped to the address space of the device on some GPUs

```c
float *hostPtr = ...;
float *devPtr = ...;
cudaMemcpy(devPtr, hostPtr, 2);
```
• On systems with a front-side bus, bandwidth of memory transfer is higher in case of using pinned memory in the host

• Obviously the OS can not allocate as many page-locked memory as pageable. And the using of too much page-locked memory can decrease the overall system performance

### 4.14. Zero-copy memory

• A special version of the pinned memory is the zero-copy memory. In this case we don’t need to transfer data from host to the device, the kernel can directly access the host memory

• Also called mapped memory because in this case the this memory region is mapped into the CUDA address space

• Useful when
  • the GPU has no memory and uses the system RAM
  • the host side wants to access to data while kernel is still running
  • the data does not fit into GPU memory
  • we want to execute enough calculation to hide the memory transfer latency

• Mapped memory is shared between host and device therefore the application must synchronize memory access using streams or events

• The CUDA device properties structures has information the capabilities of the GPU: canMapHostMemory = 1 if the mapping feature is available

### 4.14.1. Portable pinned memory

• Pinned memory allowed to move between host threads (in case of multi-GPU environments)

### 5. Execution model

#### 5.1. CUDA execution model - threads

• Each thread has a unique ID. So each thread can decide what data to work on

• It can be
  • 1 dimensional
  • 2 dimensional (Fig. 3.3.1)
  • 3 dimensional

• Thread ID is available in the kernel via threadIdx variable

• In case of multidimensional index space, the threadIdx is a structure with the following fields:
  • `threadIdx.x`
  • `threadIdx.y`
  • `threadIdx.z`

2.33. ábra -
5.2. CUDA thread blocks

- CUDA devices has a limitation for the maximal number of parallel executable threads. The index space of a complex task can be greater than this limit (for example maximum 512 thread ↔ 100x100 matrix = 10000 threads)

- In these cases the device will split the entire index space to smaller thread blocks. The scheduling mechanism will process all of these blocks and it will decide the processing order (one-by-one or in case of more than one multiprocessors in a parallel way)

- The hierarchy of blocks is the grid

5.2.1. Block splitting method

- In CUDA, the framework will create, initialize and start all of the threads. The creation, initialization of the blocks is the framework’s task too.

- The programmer can influence this operation via the following parameters (kernel start parameters):
  - Number of threads within a single block (1, 2 or 3 dimension)
  - Number of blocks in the grid (1 or 2 dimension)

5.3. CUDA thread block indexes

- Thread block also have a unique ID. So a thread can reach the owner block data

- It can be
  - 1 dimensional
  - 2 dimensional (Fig. 2.5.2)
• 3 dimensional (Fermi and after)

• Block

ID is available in the kernel via blockIdx variable

• In case of multidimensional index space, the threadIdx is a structure with the following fields:

  • blockIdx
    Idx.x
  • blockIdx
    Idx.y
  • blockIdx
    Idx.z

2.34. ábra - [5]
5.4. Global and local indices

5.4.1. Local identifier

- Every thread have a local identifier, it is stored in the previously introduced threadIdx variable
- This number shows the thread’s place within the block
- The identifier of the „first” thread is (based on the block dimensions):
  0 or [0,0] or [0,0,0]

5.4.2. Global identifier
• In case of more than one block, the local identifier is not unique anymore

• We know the identifier of the block (the owner of the thread), the previously introduced blockIdx variable and the size of the blocks (blockDim variable), we can calculate the global identifier of the thread:

• Pl. Global_x component = blockIdx.x * blockDim.x + threadIdx.x

• The programmer can not send unique parameters to the threads (for example, which matrix element to process). Therefore the thread must use it’s unique global identifier to get it’s actual parameters

5.5. Some useful formulas

• Size of the index space:

\[ G_x, G_y \]

(derived from the problem space)

• Block size:

\[ S_x, S_y \]

(derived from the current hardware)

• Number of threads:

\[ G_x \times G_y \]

(number of all threads)

• Global identifiers: \((0..G_x - 1, 0..G_y - 1)\)

( unique identifier for all threads)

• Number of blocks:

\[ (W_x, W_y) = ((G_x - 1)/ S_x + 1, (G_y - 1)/ S_y + 1) \]

(number of block for the given block size)

• Global identifier: \((g_x, g_y) = (w_x \times S_y + s_y, w_y \times S_x + s_x)\)

• Local identifier: \((w_x, w_y) = ((g_x - s_x)/ S_x, (g_y - s_y)/ S_y)\)

5.6. Create a kernel

• A CUDA kernel looks like a simple C function, but there are some significat differences:

  • there are some special keywords
  • there are some special available variables in the function’s body (the previously mentioned threadIdx etc.)
  • directly not callable from the host code, there is a special kernel invocation syntax

5.6.1. CUDA keywords to sign functions

• __device__

  • Executed in: device
  • Callable from: device

  • __global__
• Executed in: device
• Callable from: host

• __host__
• Executed in: host
• Callable from: host

5.7. Start a kernel

• Any host function can call a kernel using the following syntax:

**Kernel name**

```
<<<Dg, Db, Ns, S>>>(parameters)
```

• where:
  • *Dg* – grid size
  • A *dim3* structure, that contains the size of the grid \( Dg.x \times Dg.y \) = number of blocks
  • *Db* – block size
  • A *dim3* structure, that contains the size of the blocks \( Dg.x \times Dg.y \times Dg.z \) = number of thread within a single block
  • *Ns* – size of the shared memory (optional parameter)
  • A *size_t* variable, that contains the size of the allocated shared memory for each blocks
  • *S* – stream (optional parameter)
  • A *cudaStream_t* variable, that contains the stream associated to the command

5.8. Built-in types

5.8.1. *dim3* type

• In case of kernel start the size of the grid and the size of the blocks are stored in a *dim3* variable. In case of the grid this is a 1 or 2 dimensional, in case of blocks this is a 1, 2 or 3 dimensional vector

• Example for usage of *dim3* variables:

```
2.35. ábra -

```dim3 meret;
meret = 10;
meret = dim3(10, 20);
meret = dim3(10, 20, 3)
```

5.8.2. *size_t* type

• Unsigned integer. Used to store memory sizes

5.8.3. *cudaStream_t* típus
• Identifies a stream. In practice an unsigned integer value

5.9. Kernel implementation

• The following example shows a simple kernel implementation (multiply all values in the vector by 2):

2.36. ábra -

```c
__global__ void vector
{
    int i = threadIdx.x;
}
```

• The `__global__` keyword signs that the device will execute the function
• In case of device functions, there must be not any result values
• The name of the kernel is `vectorMul`
• The function has one parameter: the address of the vector
• As it is clearly visible, the kernel don’t have any information about the execution parameters (how many threads, how many blocks etc.)
• As discussed before, the kernel can use the threadIdx variable to determine which vector element to multiply

5.10. Kernel invocation

• If the size of the vector is not greater than the number of maximum threads, on block is enough to process the entire data space
• We use 1x1 grid size (first parameter)
• We use 200x1 block size (second parameter)

2.37. ábra -

```c
float* A = ...
... Transfer data ...
vectorMul<<<1, 200>: 
... Transfer results ...
```

• With these execution parameters the device will create one block and 200 threads
• The local identifiers of the threads will be one dimensional numbers from 0 to 199
• The identifier of the block will be 0
• The block size will be 200

5.11. Using multiple-block kernel

• If we want to process 2000 items which is more than the number of maximum threads in a single block, we have to create more than one blocks in the device:
2.38. ábra -

```c
__global__ void vector
{
    int i = blockIdx.x * blockDim.x;
    if (i < N)
    {
    }
}
```

- In the first line the kernel calculates it’s global identifier. This will be a globally unique number for each threads in each blocks

### 5.12. Invoking a multiple-block kernel

- If we want to process 1000 element and the maximum block size is 512 (with Compute Capability 1.0), we can use the following parameters:
  
  - 4 blocks (identifiers are 0, 1, 2 and 3)
  - 250 threads (local identifiers are 0..249)

2.39. ábra -

```c
float* A = ...
...Transfer data ...
vectorMul<<4, 250>;
...Transfer results ...
```

- If we don’t know the number of elements at compile time, we can calculate the correct block and thread numbers (N – vector size, BM – chosen block size):
  
  - Number of blocks: \( \frac{N-1}{BM} + 1 \)
  - Size of blocks: BM

### 5.13. Create the entire application

#### 5.13.1. Exam 3.3.1

Create a CUDA application to solve the following problems:

- List the name of all CUDA compatible devices
- The user can choose on of them
- Allocate an A vector with size N (A)
- Fill the A vector with random data
- Move these values to the GPU global memory
- Create and start a kernel to calculate \( A = A \times 2 \)
Use N blocks and BlockN size blocks

• Move back the results to A in system memory

• Write out the result to the screen
3. fejezet - Programming interface

1. Using Visual Studio

1.1. Visual Studio capabilities

- Latest CUDA versions support Visual Studio 2008/2010
- After installing CUDA some new functions appear in Visual Studio
  - New project wizard
  - Custom build rules
  - CUDA syntax highlighting
  - Etc.

1.1.1. New project wizard

- Select File/New/Project/Visual C++/CUDA[64]/CUDAWinApp
- Click “Next” on the welcome screen

3.1. ábra -

1.2. New project wizard

- Select application type
• Windows application
• Console application – we will use this option in our examples
• DLL
• Static library
• Select header files for
  • ATL
  • MFC
• Set additional options
  • Empty project
  • Export symbols
  • Precompiled header
• Click “Finish” to generate an empty CUDA project

3.2. ábra -

1.3. Custom build rules
• Right click on project name, and select “Custom build rules”
• There are one or more CUDA custom build rules in the appearing list
• Select the appropriate one based on the followings
  • Want to use runtime API or Driver API
  • CUDA Version

3.3. ábra -

1.4. CUDA related project properties
• Select project and click on “Project properties” and click on “CUDA Build Rule”
• There are several options in multiple tabs (debug symbols, GPU arch., etc.)
• These are the same options as discussed in nvcc compiler options part
• The “Command Line” tab shows the actual compiling parameters

3.4. ábra -
2. Compute capabilities

2.1. Compute capability (1)

- The difference between the newer and older graphics cards are more than the number of execution units and the speed of the processing elements. Often there are really dramatic changes in the whole CUDA architecture. The compute capability is a sort of hardware version number.

- The compute capability of a device is defined by a major revision number and a minor revision number.

- Devices with the same major revision number are of the same core architecture

2.1.1. Details for hardware versions

- Compute capability 1.0
  - The maximum number of threads per block is 512
  - The maximum sizes of the x-, y-, and z-dimension of a thread block are 512, 512, and 64, respectively
  - The maximum size of each dimension of a grid of thread blocks is 65535
  - The warp size is 32 threads
  - The number of registers per multiprocessor is 8192
  - The amount of shared memory available per multiprocessor is 16 KB organized into 16 banks
  - The total amount of constant memory is 64 KB
  - The cache working set for constant memory is 8 KB per multiprocessor

2.2. Compute capability (2)
• Compute capability 1.0 (cont.)
  • The cache working set for constant memory is 8 KB per multiprocessor
  • The cache working set for texture memory varies between 6 and 8 KB per multiprocessor
  • The maximum number of active blocks per multiprocessor is 8
  • The maximum number of active warps per multiprocessor is 24
  • The maximum number of active threads per multiprocessor is 768
  • For a texture reference bound to a one-dimensional CUDA array, the maximum width is 213
  • For a texture reference bound to a two-dimensional CUDA array, the maximum width is 216 and the maximum height is 215
  • For a texture reference bound to linear memory, the maximum width is 227
  • The limit on kernel size is 2 million PTX instructions
  • Each multiprocessor is composed of eight processors, so that a multiprocessor is able to process the 32 threads of a warp in four clock cycles

• Compute capability 1.1
  • Support for atomic functions operating on 32-bit words in global memory

2.3. Compute capability (3)

• Compute capability 1.2
  • Support for atomic functions operating in shared memory and atomic functions operating on 64-bit words in global memory
  • Support for warp vote functions
  • The number of registers per multiprocessor is 16384
  • The maximum number of active warps per multiprocessor is 32
  • The maximum number of active threads per multiprocessor is 1024

• Compute capability 1.3
  • Support for double-precision floating-point numbers

• Compute capability 2.0
  • 3D grid of thread blocks
  • Floating point atomic functions (addition)
  • __ballot() function is available (warp vote)
  • __threadfence_system() function is available
  • __systhreads_count() function is available
  • __systhreads_and() function is available
  • __systhreads_or() function is available
  • Maximum dimension of a block is 1024
• Maximum number of threads per block

2.4. Compute capability (4)

• Compute capability 2.0 (cont)
  • Warp size is 32
  • Maximum threads per multiprocessors is 1536
  • Number of 32 bit registers per multiprocessors is 32K
  • Number of shared memory banks is 32
  • Amount of local memory per thread is 512KB

• Compute capability 3.0
  • Atomic functions operating on 64-bit integer values in shared memory
  • Atomic addition operating on 32-bit floating point values in global and shared memory
  • __ballot()
  • __threadfence_system()
  • __syncthreads_count()
  • __syncthreads_and()
  • __syncthreads_or()
  • Surface functions
  • 3D grid of thread blocks
  • Maximum number of resident blocks per multiprocessor is 16
  • Maximum number of resident warps per multiprocessor is 64
  • Maximum number of resident threads per multiprocessor is 2048

2.5. Compute capability (5)

• Compute capability 3.0 (cont)
  • Number of 32-bit registers per multiprocessor is 64K

• Compute capability 3.5
  • Funnel Shift
  • Maximum number of 32-bit registers per thread is 255

2.6. Device parameters (1)

3.5. ábra -
2.7. Device parameters (2)

3. CUDA language extensions

3.1. CUDA language extensions

- The CUDA source is similar to a standard C or C++ source code and the development steps are the same too. The nvcc compiler do most of the job (separate the CPU and GPU code, compile these sources, linking the executable), this is invisible for the programmer.

- There are some special operations for making kernels, executing kernels etc. These are usually extended keywords and functions, but most of them looks like standard C keywords and functions.

- CUDA source code can be C or C++ based, in practice we will use standard C language in these lessons.

• The runtime library split into:
  • host component, that runs on the host and provides functions to control and access the compute devices
  • device component, that runs on the device and provides device-specific functions
  • common component, that provides built-in types, and a subset of the C library that are supported in both host and device code

3.2. Common component – new variable types

3.2.1. Built-in vector types
• New built-in types for vectors:
  • char1, uchar1, char2, uchar2, char3, uchar3, char4, uchar4
  • short1, ushort1, short2, ushort2, short3, ushort3, short4, ushort4
  • int1, uint1, int2, uint2, int3, uint3, int4, uint4
  • long1, ulong1, long2, ulong2, long3, ulong3, long4, ulong4
  • float1, float2, float3, float4, double2
• For example int4, means a 4 integer size vector
• The components of the vectors are accessible via the x, y, z, w fields (according to the dimension of the vector)
• All of these vectors have constructor function named make_type. For example: int2 make_int2(int x, int y)

3.2.2. dim3 type
• This type is an integer vector type based on uint3 that is used to specify dimensions
• When defining a variable of type dim3, any component left unspecified is initialized to 1

3.3. Common component – available functions

3.3.1. Mathematical functions
• Kernels run in the device therefore most of the common C functions are unavailable (I/O operations, complex functions, recursion etc.)
• CUDA supports most of the C/C++ standard library mathematical functions.
  When executed in host code, a given function uses the C runtime implementation if available
  • basic arithmetic
  • Sin/cos etc.
  • Log, sqrt etc.

3.3.2. Time functions
• The clock() function should measure the runtime of the kernels. The signature of this function:
  
clock_t clock
• The return value is the actual value of a continuously incrementing counter (based on the clock frequency)
• Provides a measure for each thread of the number of clock cycles taken by the device to completely execute the thread, but not of the number of clock cycles the device actually spent executing thread instructions.

3.4. Device component - built-in variables

3.4.1. gridDim
• Type: dim3
• Contains the dimensions of the grid

3.4.2. blockIdx
• Type: uint3
• Contains the block index within the grid

3.4.3. blockDim
• Type: dim3
• Contains the dimensions of the block

3.4.4. threadIdx
• Type: uint3
• Contains the thread index within the block

3.4.5. warpSize
• Type: int
• Contains the warp size in threads

3.5. Device component - functions

3.5.1. Fast mathematical functions
• For some of the functions, a less accurate, but faster version exists in the device runtime component
• It has the same name prefixed with __, like:
  __fdividef__, __sinf, __cosf, __tanf, __sincosf, __logf, __log2f, __log10f, __expf, __exp10f, __powf
• The common C functions are also available, but it is recommended to use the functions above:
  • Faster, based on the hardware units
  • Less accurate

3.5.2. Synchronization within a block
• void __syncthreads()
  • effect: synchronizes all threads in a block. Once all threads have reached this point, execution resumes normally
  • scope: threads in a single block
• __syncthreads is allowed in conditional code but only if the conditional evaluates identically across the entire thread block, otherwise the code execution is likely to hang or produce unintended side effects

### 3.6. Device component – atomic functions

• An atomic function performs a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory:

  \begin{itemize}
  \item atomicAdd, atomicSub, atomicExch, atomicMin, atomicMax, atomicInc, atomicDec, atomicCAS, atomicAnd, atomicOr, atomicXor
  \end{itemize}

• The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads

• Impair the efficiency of parallel algorithms

#### 3.6.1. Warp vote functions

• Compute Capability 1.2 and after

  \begin{itemize}
  \item int __all(int condition)
    
    Evaluates predicate for all threads of the warp and returns non-zero if and only if predicate evaluates to non-zero for all of them
  \item int __any(int condition)
    
    Evaluates predicate for all threads of the warp and returns non-zero if and only if predicate evaluates to non-zero for any of them
  \end{itemize}

### 3.7. Host component - functions

• Device handling functions

  \begin{itemize}
  \item See next chapter
  \end{itemize}

• Context handling functions

  \begin{itemize}
  \item See next chapter
  \end{itemize}

• Memory handling functions

  \begin{itemize}
  \item See next chapter
  \end{itemize}

• Program module handling functions

  \begin{itemize}
  \item See next chapter
  \end{itemize}

• Kernel handling functions

  \begin{itemize}
  \item See next chapter
  \end{itemize}

#### 3.7.1. Error handling

• cudaError_t cudaGetLastError()

  Result is the error code of the last command

• Const char* cudaGetErrorString(cudaError_t error)

  Result is the detailed description of an error code
4. Asynchronous Concurrent Execution

4.1. Streams

• Applications manage concurrency through streams

• A stream is a sequence of commands (possibly issued by different host threads) that execute in order. Different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently; this behavior is not guaranteed and should therefore not be relied upon for correctness [11]

• Streams support concurrent execution

  • Operations in different streams may run concurrently

  • Operations in different streams may be interleaved

3.7. ábra - [12]

4.2. Creating/destroying streams

• Stream is represented by a cudaStream_t type

• Create a stream with cudaStreamCreate function

  • Parameters: pStream – pointer to a new stream identifier

3.8. ábra -

1 2
cudaStream_t stream;
cudaStreamCreate(&stream)

• Destroy stream with cudaStreamDestroy function

3.9. ábra -

1
cudaStreamDestroy(stream)

• Common pattern to create/destroy an array of streams
4.3. Using streams

- Some CUDA functions have an additional stream parameter
  - cudaError_t cudaMemcpyAsync(
    void *dst,
    const void *src,
    size_t count,
    enum cudaMemcpyKind kind,
    cudaStream_t stream = 0)

- Kernel launch:
  Func<<< grid_size, block_size, shared_mem, stream >>>

- Concurrent execution may need some other requirements
  - Async memory copy to different directions
  - Page locked memory
  - Enough device resources

- In case of missing stream parameter the CUDA runtime use the default stream (identified by 0)
  - Used when no stream is specified
  - Completely synchronous host to device calls
  - Exception: GPU kernels are asynchronous with host by default if stream parameter is missing

4.4. Using streams example
All stream1 and stream2 operations will run concurrently

Data used by concurrent operations should be independent

### 4.5. Stream synchronization

- Synchronize everything with cudaDeviceSynchronize() blocks host until all CUDA calls are complete

#### 3.12. ábra -

```c
  cudaDeviceSynchronize()
```

- Synchronize to a specific stream with cudaStreamSynchronize
  - Parameters: stream – stream to synchronize

#### 3.13. ábra -

```c
  cudaStreamSynchronize()
```

- Programmer can create specific events within streams for synchronization

### 4.6. Operations implicitly followed a synchronization

- Page-locked memory allocation
  - cudaMemcpyHost
  - cudaMemcpyHost

- Device memory allocation
  - cudaMemcpy

- Non-async version of memory operations
  - cudaMemcpy
  - cudaMemcpy
• Change to L1/shared memory configuration
  • cudaDeviceSetCacheConfig

4.7. Stream scheduling [12]

• Fermi hardware has 3 queues
  • 1 Compute Engine queue
  • 2 Copy engine queues
    • Host to device copy engine
    • Device to host copy engine
• CUDA operations are dispatched to devices in the sequence they were issued
  • Placed in the relevant queue
• Stream dependencies between engine queues are maintained but lost within an engine queue
• CUDA operation is dispatched from the engine queue if
  • Preceding calls in the same stream have completed,
  • Preceding calls in the same queue have been dispatched, and
  • Resources are available
• CUDA kernels may be executed concurrently if they are in different streams
  • Thread blocks for a given kernel are scheduled if all thread blocks for preceding kernels have been scheduled and there still are SM resources available
• Note a blocked operation blocks all other operations in the queue, even in other streams

4.8. Concurrency support

• Compute Capability 1.0
  • Support only for GPU/CPU concurrency

• Compute Capability 1.1
  • Supports asynchronous memory copies
    • Check asyncEngineCount device property

• Compute Capability 2.0
  • Supports concurrent GPU kernels
    • Check concurrentKernels device property
  • Supports bidirectional memory copies based on the second copy engine
    • Check asyncEngineCount device property

4.9. Blocked Queue example

• Two streams with the following operations
• Stream1: HDa1, HDb1, K1, DH1

• Stream2: DH2

### 3.14. ábra - [12]

#### 4.10. Blocked Kernel example

- Two streams with the following operations
  - Stream1: Ka1, Kb1
  - Stream2: Ka2, Kb2

### 3.15. ábra - [12]

#### 5. CUDA events

##### 5.1. Create and destroy a new event

- The cudaEventCreate function creates a new CUDA event

  \[
  \text{cudaError_t cudaEventCreate(cudaEvent_t *event)}
  \]

- The first parameter of the function is an event object pointer
- The function will create a new event object the passed pointer will reference to this
• The result of the function is the common CUDA error code

• An example

3.16. ábra -

![CUDA event creation example]

• There is an advanced version of this function, called cudaEventCreateWithFlags (see CUDA documentation)

• The cudaEventDestroy function destroys a CUDA event object

    `cudaError_t cudaEventDestroy(cudaEvent_t event)`

    • The first parameter the already existing event object to destroy

    • An example:

3.17. ábra -

![CUDA event destruction example]

5.2. Record an event

• The cudaEvent Record function records an already existing event in a specified stream

    `cudaError_t cudaEventRecord (`

    `cudaEvent_t event,`

    `cudaStream_t stream = 0`}

    `)`

    • The first parameter is the event to record

    • The second parameter is the stream in which to record the event

    • The event is recorded after all preceding operations in the given stream have been completed (in case of zero stream it is recorded after all preceding operations in the entire CUDA context have been completed)

    • `cudaEventQuery()` and/or `cudaEventSynchronize()` must be called to determine when the event actually been recorded (since this function call is asynchronous)

    • If the event has been recorded, then this will overwrite the existing state

3.18. ábra -

![CUDA event recording example]
5.3. Synchronize an event

- The cudaEventSynchronize function synchronizes and event. It will wait until the completion of all device operations preceding the most recent call to cudaEventRecord() in the given stream

\[
\text{cudaError_t cudaEventSynchronize(cudaEvent_t event)}
\]

- The first parameter is the event to wait for

- If cudaEventRecord has not been called on the specified event the function will return immediately

- Waiting for the event will cause the calling CPU thread to block until the event has been completed by the device

3.19. ábra -

3.4. Check an event

- The cudaEventQuery function returns information about and event

\[
\text{cudaError_t cudaEventQuery(cudaEvent_t event)}
\]

- The first parameter is the event to check for

- Query the status of all device work preceding the most recent call to cudaEventRecord()

- If this work has successfully been completed by the device, or if cudaEventRecord() has not been called on event, then \text{cudaSuccess} is returned

- If this work has not yet been completed by the device then \text{cudaErrorNotReady} is returned

3.20. ábra -
5.5. Synchronization with events

- The `cudaStreamWaitEvent` function will block a stream until an event finishes

```c
cudaEvent_t test_event
...
if (cudaEventQuery(event)
    ... event has been fini.
} else {
    ... event has not been
}
```

- The first parameter is the stream to block
- Second parameter is the event to wait on
- Third parameters are the optional flags (must be 0)
- Makes all future work submitted to stream wait until event reports completion before beginning execution. This synchronization will be performed efficiently on the device
- The event may be from a different context than stream, in which case this function will perform cross-device synchronization
- The stream will wait only for the completion of the most recent host call to `cudaEventRecord()` on event
- If stream is NULL, any future work submitted in any stream will wait for event to complete before beginning execution. This effectively creates a barrier for all future work submitted to the device on this thread

5.6. Synchronization with events (example)

3.21. ábra -
5.7. Calculate elapsed time between two events

- The cudaEventElapsedTime computes the elapsed time between two finished events

\[ \text{cudaError_t cudaEventElapsedTime(float *ms,} \]

- \text{cudaEvent_t start,}
- \text{cudaEvent_t end}

- The first parameter is a float pointer. The result will be stored into this variable
- Start event is the first event
- End event is the second event
- cudaEventRecord() must be called for each events
- Both of the events must be in finished state
- Do not use the \text{cudaEventDisableTiming} flag (advanced event creation)
- If timing is not necessary for performance use:

\[ \text{cudaEventCreateWithFlags(&event, cudaEventDisableTiming)} \]

5.8. Calculate elapsed time (example)

3.22. ábra -
6. Unified Virtual Address Space

6.1. CUDA Unified Virtual Address Management

• Unified virtual addressing (UVA) is a memory address management system enabled by default in CUDA 4.0 and later releases on Fermi and Kepler GPUs running 64-bit processes. The design of UVA memory management provides a basis for the operation of RDMA for GPUDirect [11]

3.23. ábra - [9]

• In the CUDA VA space, addresses can be:
  • GPU – page backed by GPU memory. Not accessible from the host
  • CPU – page backed by CPU memory. Accessible from the host and the GPU
  • Free – reserved for future CUDA allocations

6.2. Unified Virtual Address Space

• UVA means that a single memory address is used for the host and all the devices
• CPU and GPU use the same unified virtual address space
  • The driver can determine from an address where data resides (CPU, GPU, one of the GPUs)
• Allocations still reside on the same device (in case of multi-GPU environments)

• Availability
  • CUDA 4.0 or later
  • Compute Capability 2.0 or later
  • 64bit operation system

• A pointer can reference an address in
  • global memory on the GPU
  • system memory on the host
  • global memory on another GPU

• Applications may query if the unified address space is used for a particular device by checking that the *unifiedAddressing* device property (CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING)

### 6.3. Unified Virtual Address Space – check availability

• Which memory a pointer points to – host memory or any of the device memories – can be determined from the value of the pointer using `cudaPointerGetAttributes()`

#### 3.24. ábra -

```c
void* A;
cudaPointerAttributes a;
cudaPointerGetAttributes(A, &a);
```

• The result of this function is a `cudaPointerAttributes` structure:

```c
cstruct cudaPointerAttributes {
enum cudaMemoryType memoryType;
int device;
void *devicePointer;
void *hostPointer;
}
```

• memoryType identifies the physical location of the memory associated with pointer ptr. It can be `cudaMemoryTypeHost` for host memory or `cudaMemoryTypeDevice` for device memory

• device is the device against which ptr was allocated

• devicePointer is the device pointer alias through which the memory referred to by ptr may be accessed on the current device

• hostPointer is the host pointer alias through which the memory referred to by ptr may be accessed on the host

### 6.4. Peer to peer communication between devices

#### 3.25. ábra - [10]
• UVA memory copy
• P2P memory copy
• P2P memory access

6.5. Using Unified Addressing and P2P transfer

• All host memory allocated using cuMemAllocHost() or cuMemHostAlloc() is directly accessible from all devices that support unified addressing

• The pointer value is the same in the host and in the device side, so it is not necessary to call any functions (cudaHostGetDevicePointer())

• All the pointers are unique, so it is not necessary to specify information about pointers to cudaMemcpy() or any other copy functions. The cudaMemcpy functions needs a parameter about transfer direction, it would be cudaMemcpyDefault. The runtime will know the location of the pointer from its value

3.26. ábra -

- cudaMemcpyHostToH
- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToDevice
- cudaMemcpyDeviceToHost

• Enables libraries to simplify their interfaces

• Note that this will transparently fall back to a normal copy through the host if P2P is not available

6.6. Peer-to-peer memory transfer between GPUs

• Check for P2P access between GPUs [10]:

3.27. ábra -

- cudaMemcpyDeviceCanAccessPeer
- cudaMemcpyDeviceCanAccessPeer
• Enable peer access between GPUs:

3.28. ábra -

```c
1 cudaSetDevice(gpid_
2 cudaDeviceEnablePeer
3 cudaSetDevice(gpid_
4 cudaDeviceEnablePeer
```

• We can use UVA memory copy:

3.29. ábra -

```c
1 cudaMemcpy(gpu0_buf
```

• Stop peer access:

3.30. ábra -

```c
1 cudaSetDevice(gpid_
2 cudaDeviceDisablePeer
3 cudaSetDevice(gpid_
4 cudaDeviceDisablePeer
```

6.7. Peer-to-peer memory access between GPUs

• System requirements are the same as P2P memory transfer

• Same checking steps [10]:

3.31. ábra -

```c
1 cudaDeviceCanAccessPeer()
2 cudaDeviceCanAccessPeer
```

• Same initialization steps:

3.32. ábra -

```c
1 cudaSetDevice(gpid_
2 cudaDeviceEnablePeer
3 cudaSetDevice(gpid_
4 cudaDeviceEnablePeer
```

• Same shutdown steps:

3.33. ábra -
6.8. Peer-to-peer memory access kernel

- Well known kernel copy an array from destination to target:

```
__global__ void CopyKernel
{
    int idx = blockIdx.x * blockDim;
    dst[idx] = src[idx];
}
```

- We can start a kernel with different parameters:

```
CopyKernel<<<blockdim
CopyKernel<<<blockdim
CopyKernel<<<blockdim
CopyKernel<<<blockdim
```

- Due to UVA the kernel knows whether its argument is from another GPU memory/host memory/local memory.

6.9. CUDA Unified Virtual Address summary

- Faster memory transfers between devices
- Device to device memory transfers with less host overhead
- Kernels in a device can access memory of other devices (read and write)
- Memory addressing on different devices (other GPUs, host memory)

- Requirements
  - 64bit OS and application (Windows TCC)
  - CUDA 4.0
  - Fermi GPU
  - Latest drivers
  - GPUs need to be on same IOH

6.9.1. More information about UVA
• CUDA Programming Guide 4.0
  • 3.2.6.4 Peer-to-Peer Memory Access
  • 3.2.6.5 Peer-to-Peer Memory Copy
  • 3.2.7 Unified Virtual Address Space
4. fejezet - Optimization techniques

1. Using shared memory

1.1. Optimization strategies

- Memory usage
  - Use registers
  - Use shared memory
  - Minimize CPU-GPU data transfers
  - Processing data instead of moving it (move code to the GPU)
  - Group data transfers
  - Special memory access patterns (we don’t discuss)

- Maximize parallel execution
  - Maximize GPU parallelism
    - Hide memory latency by running as many threads as possible
  - Use CPU-GPU parallelism
  - Optimize block size
  - Optimize number of blocks
  - Use multiple-GPUs

- Instruction level optimization
  - Use float arithmetic
  - Use low precision
  - Use fast mathematic functions
  - Minimize divergent warps
  - Branch conditions

1.2. Matrix multiplication

1.2.1. Exam 4.1.1

Create CUDA application to solve the following problems: multiplying 2 dimensional (NxN) matrices with the GPU

- N is a constant in source code
- Allocate memory for 3 Nxn matrices (A, B, C)
- Fill the A matrix with numbers (for example: a_{i,j} = i + j)
- Fill the B matrix with numbers (for example: b_{i,j} = i - j)
• Allocate 3 NxN matrices in the global memory of the graphics card (devA, devB, devC)
• Move the input data to the GPU: \(A \rightarrow \text{devA}, \ B \rightarrow \text{devB}\)
• Execute a kernel to calculate \(\text{devC} = \text{devA} \times \text{devB}\)
• Move the results back to the system memory: \(\text{devC} \rightarrow C\)
• List the values in the C vector to the screen

1.3. Multi-dimensional matrix in global memory

• We can use multi-dimensional arrays in C programs, but these are obviously stored in a linear memory area
• For example a 4x4 matrix in the memory:

\[
\begin{align*}
A &= \begin{bmatrix}
\mathbf{a}_{0,0} & \mathbf{i} \\
\mathbf{a}_{1,0} & \mathbf{i} \\
\mathbf{a}_{2,0} & \mathbf{i} \\
\mathbf{a}_{3,0} & \mathbf{i} \\
\end{bmatrix}
\end{align*}
\]

A two dimensional array

1.3.1. Access elements of a multi-dimensional array

• We know the address of the first item in the array and we know the size of each elements. In this case we can use the following formula:

\[
a_{\text{row},\text{col}} = a_{0,0} + (\text{row} \times \text{col\_number} + \text{col}) \times \text{item\_size}
\]

• The CUDA kernel will get only the starting address of the array, we have to use this formula to access the elements
1.4. Multi-dimensional matrix multiplication

- If one thread processes one item in the matrix, we need as many threads as the number of matrix elements. A relatively small 30x30 matrix needs 900 threads in GPU, therefore we have to use multiple blocks.

- Therefore we have to use the block identifier in the kernel. The improved kernel for the devC = devA * devB matrix multiplication:

4.3. ábra -

```c
__global__ static void Matrix(int idx, int jdx, float* devC, float* devA, float* devB, int N)
{
    float sum = 0;
    for(int i = 0; i < N; i++)
    {
        sum += devA[idx * N + i] * devB[jdx * N + i];
    }
    devC[idx * N + jdx] = sum;
}
```

1.5. Multi-dimensional matrix in the GPU memory

- Initialization, memory allocation

4.4. ábra -

```c
cudaSetDevice(0);
cudaMalloc((void**) &devA,
cudaMalloc((void**) &devB,
cudaMalloc((void**) &devC,
```

- Move input data

4.5. ábra -

```c
cudaMemcpy(devA, A, sizeof
```

- Invoke the kernel

4.6. ábra -
• Move the results back, free memory

4.7. ábra -

cudaMemcpy(C, devC, sizeC

cudaFree(devA); cudaFree(d)

1.6. Aligned arrays

• In some cases the number of columns in one matrix row differs from the size of the rows in the memory. This can speed up the access of values because technical reasons (for example with the real memory row size, we can use faster multiplications or we can utilize the capacity of the GPU memory controllers)

• A simple 5x5 matrix with 8 item alignment:

A matrix

4.8. ábra -

A array in memory

4.9. ábra -
1.7. Access elements in case of aligned storage

- The formula is similar but we use the aligned row size:

\[ a_{row, col} = a_{0,0} + (row \times aligned\_row\_size + col) \times item\_size \]

1.8. Aligned memory management

- The CUDA class library have several functions to manage aligned memory. The following function allocates aligned memory area:

  \[ \text{cudaMallocPitch}(\text{void**} \ devPtr, \text{size\_t } pitch, \text{size\_t } width, \text{size\_t } height) \]

  - \( devPtr \) – pointer to the aligned memory
  - \( pitch \) – alignment
  - \( width \) – size of one matrix row
  - \( height \) – number of matrix rows

- Similar to the linear memory management the start address of the allocated object will be stored in the \( devPtr \) variable

- The alignment is not an input value, this is one of the outputs of the function. The CUDA library will determine the optimal value (based on the array and device properties)

- Size of the matrix row is given by bytes

1.9. Copy aligned memory

- Because the different alignment the normal linear memory transfer is not usable in case of pitched memory regions

- The following CUDA function transfers data from one region to an other

  \[ \text{cudaMemcpy2D}(\text{void*} \ dst, \text{size\_t } dpitch, \text{const void*} \ src, \text{size\_t } spitch, \text{size\_t } width, \text{size\_t } height, \text{enum cudaMemcpyKing } kind) \]

  - \( dst \) – destination pointer
  - \( dpitch \) – destination pitch value
  - \( src \) – source pointer
• \textit{spitch} – source pitch value
• \textit{width} – size of one row of the 2 dimensional array
• \textit{height} – number of rows of the 2 dimensional array
• \textit{kind} – transfer direction
  • host \to host (\texttt{cudaMemcpyHostToHost})
  • host \to device (\texttt{cudaMemcpyHostToDevice})
  • device \to host (\texttt{cudaMemcpyDeviceToHost})
  • device \to device (\texttt{cudaMemcpyDeviceToDevice})

• In case of simple not aligned arrays, the pitch value is 0

1.10. Matrix multiplication with aligned arrays

1.10.1. Exam 4.1.2

Create CUDA application to solve the following problems: multiplying 2 dimensional (NxN) matrices with the GPU

• N is a constant in source code
• Allocate memory for 3 NxN matrices

(A, B, C)
• Fill the A matrix with numbers (for example: \(a_{ij} = i + j\))
• Fill the B matrix with numbers (for example: \(b_{ij} = i - j\))
• Allocate 3 NxN \texttt{pitched arrays} in the global memory of the graphics card (devA, devB, devC)
• Move the input data to the GPU: A \to devA, B \to devB
• Execute a kernel to calculate \(\text{devC} = \text{devA} \times \text{devB}\)
• Move the results back to the system memory: \(\text{devC} \to C\)
• List the values in the C vector to the screen

1.11. Kernel with aligned arrays

• The multiplier is the pitch value instead of the matrix column number
• The pitch size is given by bytes therefore in case of typed pointers we have to correct it’s actual value by a \texttt{sizeof(item\_type)} division
• \(\text{devC} = \text{devA} \times \text{devB}\) source code:

\textbf{4.10. ábra -}
1.12. Invoke kernel with aligned arrays

- Initialization, allocate arrays

4.11. ábra -

```c
__global__ static void MatrixMul(float

int indx = blockIdx.x * blockDim.x + t
int indy = blockIdx.y * blockDim.y + t

if (indx < N && indy < N) {
  float sum = 0;
  for(int i = 0; i < N; i++) {
    sum += devA[indy * pitch/sizeof
  }
  devC[indy * pitch/sizeof(float) + in
}
```

- Transfer input data (we assume pitch value is the same)

4.12. ábra -

```c
cudaSetDevice(0);
float A[N][N], B[N][N], C[N][N];
cudaMallocPitch((void**) &a
cudaMallocPitch((void**) &b
cudaMallocPitch((void**) &c

ccudaMemcpy2D(devA, pitch, A, sizeof(float) * 1
ccudaMemcpy2D(devB, pitch, B, sizeof(float) * 1
```

- Kernel invocation

4.13. ábra -

```c
dim3 grid((N - 1) / BlockN +
dim3 block(BlockN, BlockN);
MatrixMul<<<grid, block>>>
cudaThreadSynchronize();
```

- Transfer results, free memory

4.14. ábra -

```c
cudaMemcpy2D(C, sizeof(float) * N, devC, pitch
ccudaFree(devA); cudaFree(devB); cudaFree(dev
```

1.13. Using shared memory

- Matrix multiplication uses relatively small amount of arithmetic operations for the amount of memory transfers
Optimization techniques

• We need as many operations as possible to hide the latency caused by memory transfers (the GPU tries to schedule the execution units in case of memory latencies but without a lot of operations this is not possible)

• Our goal is to increase the ratio of arithmetic operations / memory transfers

1.13.1. Available solutions

• Increase parallelism (in this case it is not possible)

• Decrease the number of memory transfers (in practice this means manually programmed caching)
  
  • holding as many variables in registers as possible
  
  • using the shared memory

• Find another solution

1.14. Tiled matrix multiplication

• One input cell is necessary for the calculations of more than one output cell. In the not optimized version of the algorithm, more than one thread will read the same input value from global memory

• It would be practical to harmonize these thread’s work:
  
  • divide the entire output matrix to small regions (tiles)
  
  • allocate shared memory for one region
  
  • in the region, every thread loads the corresponding value from the input matrices to the shared memory
  
  • every thread calculates one partial result based on the values in the shared memory

• The size of the shared memory is limited therefore the steps above are usually executable only in more than one steps. We have to divide the input matrix to more than one tiles, and at the end of the kernel executions we have to summarize the values in these tiles

• The latter case it is necessary to synchronize the threads. Every thread must wait until all of the other threads loads the values from global memory to the shared memory, and after that the threads must wait again until all of them finished calculation before load the next value

1.15. Matrix multiplication

1.15.1. Exam 4.1.3

Create CUDA application to solve the following problems: multiplying 2 dimensional (NxN) matrices with the GPU

• N is a constant in source code

• Allocate memory for 3 NxN matrices(A, B, C)

• Fill the A matrix with numbers (for example: \( a_{i,j} = i + j \))

• Fill the B matrix with numbers (for example: \( b_{i,j} = i - j \))

• Allocate 3 NxN matrices in the global memory of the graphics card (devA, devB, devC)

• Move the input date to the GPU: \( A \rightarrow \text{devA} \), \( B \rightarrow \text{devB} \)

• Execute a kernel to calculate \( \text{devC} = \text{devA} \ast \text{devB} \) with tile technique

• Move the results back to the system memory: \( \text{devC} \rightarrow C \)
• List the values in the C vector to the screen

1.16. Optimized matrix multiplication

1. Division to tiles. In this case 3x3 regions, 3x3 threads

2. Every thread copies one value from the global memory to the shared memory

3. Synchronization

4.15. ábra -

1.17. Optimized matrix multiplication (2)

4. Every thread calculated one cell’s result in the shared memory

5. Synchronization

4.16. ábra -
1.18. **Optimized matrix multiplication (3)**

6. Load next tiles

7. Synchronization

8. Threads do the multiplication again and add the result to the already existing results

9. Synchronization

4.17. ábra -
1.19. Optimized matrix multiplication (4)

6. Load next tiles

7. Synchronization

8. Threads do the multiplication again and add the result to the already existing results

9. Synchronization

4.18. ábra -
1.20. Optimized matrix multiplication (5)

10. Every thread copies the result to the result matrix C

11. When all of the blocks finished, the C matrix contains the final result

Threads do the multiplication again. The result added to the already existing partial result. Synchronization

4.19. ábra -
1.21. Optimized matrix multiplication source code

- Kernel invocation is the same as the not-optimized version

---

4.20. ábra -

```c
__global__ static void MatrixMul(float
__shared__ float A_shared[BlockN]
__shared__ float B_shared[BlockN]
int indx = blockIdx.x * blockDim.x +
int indy = blockIdx.y * blockDim.y +
float c = 0;
for(int k = 0; k < N / BlockN; k++) {
    A_shared[threadIdx.y][threadIdx.x]
    B_shared[threadIdx.y][threadIdx.x]
    __syncthreads();
    for(int i = 0; i < BlockN; i++) {
        c += A_shared[threadIdx.y][i]
    }
    __syncthreads();
}
devC[indx + indy * N] = c;
```
1.22. Comparing runtime of original and tiled algorithms

- Horizontal axis: size of matrix (N)
- Vertical axis: runtime (second)

4.21. ábra -

2. Using atomic instructions

2.1. Atomic operations

- Atomic operations are operations which are performed without interference from any other threads. Atomic operations are often used to prevent race conditions which are common problems in multithreaded applications [8].

- In case of some task we need atomic operations, for example:
  - sum/average of a data structure
  - min/max item of a data structure
  - Count of some items in a data structure
  - etc.

2.1.1. Possible solutions

- We can implement some of these tasks in parallel environment (for example, is there any special item in the data structure?)

- But some of them is hard to parallelize (for example, find the minimum value in the data structure)

2.2. CUDA atomic instructions

- The atomic instructions of the CUDA environment can solve the race conditions mentioned before. When using atomic instructions the hardware will guarantee the serialized execution
• Operand location
  • variable in global memory
  • variable in shared memory

• Operand size
  • 32bit integer (Compute Capability 1.1)
  • 64bit integer (Compute Capability 1.2)

2.2.1. Performance notes
• If two threads perform an atomic operation at the same memory address at the same time, those operations will be serialized. This will slow down the kernel execution

• In case of some special tasks, we can not avoid atomic instructions. But in most cases if it is possible we would try to find another solution. The goal is to use as less atomic instructions as possible.

2.3. CUDA atomic instructions
• The first parameter of atomic instructions is usually a memory address (in global or local memory), the second parameter is an integer

  • int atomicAdd(int* address, int val)

    Reads the 32-bit or 64-bit word old located at the address in global or shared memory, computes (old + val), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old

  • int atomicSub(int* address, int val)

    Reads the 32-bit word old located at the address in global or shared memory, computes (old - val), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old

  • int atomicExch(int* address, int val);

    Reads the 32-bit or 64-bit word old located at the address in global or shared memory and stores val back to memory at the same address. These two operations are performed in one atomic transaction. The function returns old

  • int atomicMin(int* address, int val);

    Reads the 32-bit word old located at the address in global or shared memory, computes the minimum of old and val, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old

2.4. CUDA atomic instructions (2)
• int atomicMax(int* address, int val);

  Reads the 32-bit word old located at the address in global or shared memory, computes the maximum of old and val, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old

• unsigned int atomicInc(unsigned int* address, unsigned int val)

  Reads the 32-bit word old located at the address in global or shared memory, computes ((old >= val) ? 0 : (old+1)), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old
• unsigned int atomicDec(unsigned int* address, unsigned int val)

    Reads the 32-bit word old located at the address in global or shared memory, computes ((old == 0) | (old > val)) ? val : (old-1), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old.

• int atomicCAS(int* address, int compare, int val)

    Compare And Swap: reads the 32-bit or 64-bit word old located at the address in global or shared memory, computes (old == compare ? val : old), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old.

2.5. CUDA atomic bitwise instructions

• int atomicAnd(int* address, int val)

    Reads the 32-bit word old located at the address in global or shared memory, computes (old & val), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old.

• int atomicOr(int* address, int val)

    Reads the 32-bit word old located at the address in global or shared memory, computes (old | val), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old.

• int atomicXor(int* address, int val)

    Reads the 32-bit word old located at the address in global or shared memory, computes (old ^ val), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old.

2.5.1. Exam 4.2.1

Create a CUDA application to solve the following problem. Find the minimal value from a randomly filled vector (length: N). Use the atomic operations!

2.6. Find the minimal value of a vector – using global memory

• The source code is really simple. Every thread calls the atomicMin atomic instruction and pass the parameter from the array based on the thread identifier.

• In this implementation the first item of the array will contain the minimal value of the array.

4.22. ábra -

```c
__global__ static void MinSeq
    int indx = blockIdx.x * blockDim.x;
    atomicMin(devA, devA[indx]);
}
```

• As it is visible this kernel can run in multi-block execution context. The atomic instructions are useable in this environment.

2.6.1. Exam 4.2.2

Try to speed up the existing algorithm. Use the shared memory instead of global memory.

2.7. Find the minimal value of a vector – using shared memory
• First we have to initialize the localMin variable, the first thread in every block will do this

• In the next step, every thread check the value indexed by its thread identifier

• After the next synchronization, the first thread will compare the local minimum to the global minimum (every block have a local minimum)

4.23. ábra -

```java
__global__ static void MinSearch(int
__shared__ int localMin[Block];
int blockSize = BlockN;
int itemc1 = threadIdx.x * 2;
int itemc2 = threadIdx.x * 2 + 1;
for(int k = 0; k <= 1; k++) {
    int blockStart = blockIdx.x * b
    int loadIdx = threadIdx.x + b1
    if (blockStart + itemc2 < N) {
        int value1 = devA[blockStart]
        int value2 = devA[blockStart + localMin[loadIdx] = value1
    } else
        if (blockStart + itemc1 < N)
            localMin[loadIdx] =
        else
            localMin[loadIdx] =
    }
__syncthreads();
```

2.7.1. Comparing runtime of global and shared memory usage

• Horizontal axis : size of vector (N)

• Vertical axis : runtime (second)

4.24. ábra -
2.8. Parallelization inside the block

- We have to try avoid atomic instructions. It would be better to find a parallelizable solution. We have to divide the task of each block into smaller parts.

- First load a part of global memory into the block’s shared memory:
  every thread load one value from the global memory to the shared memory array

- Inside the block every thread compare two values and store the smaller one into the vector cell with smaller index

- In the next iteration we will check only the smaller items

- In the last step we have the minimal value of the block. We have to find only the global minimum (same as before)

2.8.1. Exam 4.2.2

Create an algorithm based on the idea before.

2.9. Parallel minimum search - loading

- One example

\[ N = 24 \]

\[ \text{BlockN} = 4 \text{ (nbr of threads)} \]

- Every block allocate one array in the shared memory (size is BlockN*2)

- Every thread in every blocks load 2 values from the global memory and stores the smaller one

- If we have empty spaces we have to fill them with some values

- Synchronization

4.25. ábra -
2.10. Parallel minimum search – find minimum of block

4.26. ábra -

- Every thread do $\log_2{\text{BlokkN}}$ number of iterations. In every iteration the threads do the following operation:

  
  \[ S_i = \min(S_{2i}, S_{2i+1}) \]

  
- At the end of the last iteration, the first value of the array will be the smallest one

- After that we find the globally minimum

  
  - using atomic instructions

  
  - we store the minimum values of blocks into another vector and redo the minimum search to this vector (this solution is better in case of large block number)
2.11. Parallel minimum search - kernel

4.27. ábra -

```c
__global__ static void MinSearch(float *devA, int N)
    __shared__ int localMin;
    int idx = blockIdx.x * blockDim.x;
    if (threadIdx.x == 0) localMin = devA[idx];
    __syncthreads();
    atomicMin(&localMin, devA[idx], __syncthreads());
    if (threadIdx.x == 0) atomicMin(
```

2.12. Parallel minimum search – kernel (2)

4.28. ábra -

```c
while (blockSize > 0) {
    int locMin = localMin[idx] < localMin; %
    __syncthreads();
    localMin[threadIdx.x] = locMin;
    __syncthreads();
    blockSize = blockSize / 2;
}
    if (threadIdx.x == 0) atomicMin(devA, loci;
```

• A more optimized version is available at

  http://developer.download.nvidia.com/compute/cuda/1.1-
  Beta/x86_website/projects/reduction/doc/reduction.pdf

• Block size must be $2^n$

2.13. Comparing runtime of atomic and parallel version

• Horizontal axis: size of vector (N)

• Vertical axis: runtime (second)

4.29. ábra -
2.14. Comparing of CPU and GPU implementation

- Horizontal axis: size of vector (N)
- Vertical axis: runtime (second)

4.30. ábra -

3. Occupancy considerations

3.1. Execution overview

- Problem space is divided into blocks
  - Grid is composed of independent blocks
• Blocks are composed of threads
• Instructions are executed per warp
  • In case of Fermi, 32 threads form a warp
  • Fermi can have 48 active warps per SM (1536 threads)
  • Warp will stall if any of the operands is not ready
• To avoid latency
  • Switch between contexts while warps stalled
  • Context switching latency is very small
• Registers and shared memory are allocated for a block as long as the block is active
  • Once a block is active it will stay active until all threads completed in that block
  • Registers/shared memory do not need store/reload in case of context switching

### 3.2. Occupancy

• Occupancy is the ratio of active processing units to available processing units

\[
\text{Occupancy} = \frac{\text{Active Warps}}{\text{Maximum Number of Warps}}
\]

Occupancy is limited by:
• Max Warps or Max Blocks per Multiprocessor
• Registers per Multiprocessor
• Shared memory per Multiprocessor

\[
\text{Occupancy} = \min(\text{register occ.}, \text{shared mem occ.}, \text{block size occ.})
\]

### 3.3. Occupancy and registers

• Fermi has 32K registers per SM
• The maximum number of threads is 1536
• For example, if a kernel uses 40 registers per thread:
  • Number of active threads: \(\frac{32K}{40} = 819\)
  • Occupancy: \(\frac{819}{1536} = 0.53\)
• In this case the number of registers limits the occupancy (meanwhile there are some unused resources in the GPU)
• Goal: try to limit the register usage
  • Check register usage: compile with \(--\text{ptxax-options=\text{-v}}\)
  • Limit register usage: compile with \(--\text{maxregcount}\)
• For example, in case of 21 registers:
  • Number of active threads: \(\frac{32K}{21} = 1560\)
• Occupancy: $1560 / 1536 \approx 1$

• This means only that the number of registers will not limit the occupancy (it is highly depends on other resources)

3.4. Occupancy and shared memory

• Size of shared memory is configurable in Fermi

  • 16K shared memory
  
  • 48K shared memory (we use this configuration in the examples)

• For example, if a kernel uses 64 bytes of shared memory

  • Number of active threads: $48K / 64 = 819$
  
  • Occupancy: $819 / 1536 = 0,53$

• In this case the size of shared memory limits the occupancy (meanwhile there are some unused resources in the GPU)

• Goal: try to limit the shared memory usage

  • Check shared memory usage: compile with `--ptxas-options=-v`
  
  • Limit shared memory usage

    • Use lower shared memory in kernels (kernel invocation)
    
    • Use appropriate L1/Shared configuration in case of Fermi

• For example, in case of 32 bytes of shared memory:

  • Number of active threads: $48K / 32 = 1536$
  
  • Occupancy: $1536 / 1536 = 1$

• This means only that the size of shared memory will not limit the occupancy (it is highly depends on other resources)

3.5. Occupancy and block size

• Each SM can have up to 8 active blocks

• There is a hardware based upper limit for block size

  • Compute Capability 1.0 – 512
  
  • Compute Capability 2.0 – 1024

• Lower limit is 1 but small block size will limit the total number of threads

• For example,

  • Block size: 128

  • Active threads in one SM: $128 \times 8 = 1024$
  
  • Occupancy: $1536 / 1024 = 0,66$

• In this case the block size limits the occupancy (meanwhile there are some unused resources in the GPU)
• Goal: try to increase the block size (kernel invocation parameter)

• For example,
  • Block size: 192
  • Active threads in one SM: 192 * 8 = 1536
  • Occupancy: 1536 / 1536 = 1

3.6. CUDA Occupancy calculator

• A CUDA tool to investigate the occupancy

• In practice it is an Excel sheet, located in „NVIDIA GPU Computing SDK\tools\CUDA_Occupancy_Calculator.xls”

• Input data:
  • Hardware configuration
  • Compute Capability
  • Shared Memory Config
  • Resource usage
  • Threads per block
  • Registers per thread
  • Shared memory per block

• Output data:
  • Active threads per MP
  • Active warps per MP
  • Active thread blocks per MP
  • Occupancy of each MP

3.7. CUDA Occupancy calculator - example

4.31. ábra -
3.8. CUDA Occupancy calculator – impact of varying block size

4.32. ábra -

3.9. CUDA Occupancy calculator – impact of varying register count

4.33. ábra -
3.10. CUDA Occupancy calculator – impact of varying shared memory

4.34. ábra -

3.11. Block size considerations [18]

- Choose number of threads per block as a multiple of warp size
- Avoid wasting computation on under-populated warps
Optimization techniques

- Optimize block size
  - More thread block – better memory latency hiding
  - Too much thread block – fewer register per thread, kernel invocation can fail if too many are registers are used

- Heuristics
  - Minimum: 64 threads per block
    - Only if multiple concurrent blocks
  - 192 or 256 threads a better choice
    - Usually still enough registers to compile and invoke successfully
  - This all depends on your computation!
    - Experiment!
  - Try to maximize occupancy
  - Increasing occupancy does not necessarily increase performance
  - But, low-occupancy multiprocessors cannot adequately hide latency on memory-bound kernels

4. ParallelNsight

4.1. ParallelNsight

- Debugger for GPGPU development
- Available only for registered users(?):
  http://www.nvidia.com/object/nsight.html

- Available editions
  - Visual Studio Edition
  - Nsight Eclipse Edition

- Main features
  - Visual Studio/Eclipse support
  - PTX/SASS Assembly Debugging
  - CUDA Debugger (debug kernels directly)
  - Use conditional breakpoints
  - View GPU memory
  - Graphics debugger
  - Profiler functions
  - Hardware requirements
• Analyzer – Single GPU system
• CUDA Debugger – Dual GPU system
• Direct3D Shader Debugger – Two separate GPU systems

4.2. Kernel debugging

• Main steps for local debugging
  • Start Nsight Monitor
    (All Programs > NVIDIA Corporation > Nsight Visual Studio Edition 2.2 > Nsight Monitor)
  • Set breakpoint
    Like setting breakpoint in CPU code

4.35. ábra -

• Start CUDA debugging in Visual Studio
  (Nsight/Start CUDA debugging)
• Debugger will stop at the breakpoint
• All the common debugger commands are available
  • Step over
  • Step into
  • Etc.
• Remote debugging
  • We do not discuss

4.3. Watch GPU memory regions

• Nsight supports the Visual Studio „Memory” window for examining the contents of GPU memory
  • Shared memory
  • Local memory
  • Global memory
• To show a memory region, select Debug/Windows/Memory
  • In case of kernel debugging just enter the name of the variable of the direct address
  • In case of direct addresses use the following keywords: __shared__, __local__, __device__
  • For example: (__shared__ float*)0
• The common visual studio functions also available
  • Watch window to check kernel variables
  • Move the cursor over a variable to see the actual value

• Built-in CUDA variables are also available
  • threadIdx
  • blockIdx
  • blockDim
  • gridDim
  • etc.

4.4. CUDA Debug Focus
• Some variables in CUDA belongs to a context
  • Registers and local memory to threads
  • Shared memory to blocks

• To see the variable actual value the developer must define the owner thread (block index and thread index)
  • Select Nsight/Windows/CUDA Debug Focus

4.36. ábra -

• Set block index
• Set thread index

• Watch window/quick watch etc. will show information about the variables of the corresponding thread

4.5. CUDA Device Summary
• An overview about the state of available devices
  • Select Nsight/Windows/CUDA Device Summary
  • Select a device from the list
- Lots of statics and runtime parameters are displayed in the right

4.37. ábra -

4.6. CUDA Device Summary - grid
- An overview about the state of available devices
  - Select Nsight/Windows/CUDA Device Summary
  - Select a grid from the list

4.38. ábra -

4.7. CUDA Device Summary - warp
- An overview about the state of available devices
  - Select Nsight/Windows/CUDA Device Summary
  - Select a running warp
4.39. ábra - 

- Developer can check the current state of all running warps
- `SourceFile/SourceLine` can be very useful to understand the execution mechanism

4.8. Debugging PTX code

- Check the Tools/Options/Debugging options
  - Select “Enable Address Level Debugging”
  - Select “Show disassembly if source is not available”
- When the CUDA debugger is stopped
  - Select “Go to Disassembly”
  - The PTX code appears (SASS code is also available)
- Debugging is the same as CPU applications

4.40. ábra - 

4.9. Using the memory checker
• The CUDA Memory Checker detects problems in global and shared memory. If the CUDA Debugger detects an MMU fault when running a kernel, it will not be able to specify the exact location of the fault. In this case, enable the CUDA Memory Checker and restart debugging, and the CUDA Memory Checker will pinpoint the exact statements that are triggering the fault [22]

• Select Nsight/Options/CUDA
  • Set “Enable Memory Checker” to true

• Launch the CUDA debugger and run the application
  • During the execution if the kernel tries to write to an invalid memory location (for example in case of arrays) the debugger will stop
  • The debugger will stop before the execution of this instruction

• The CUDA memory checker will write results to the Output window
  • Launch parameters
  • Number of detected problems
  • GPU state in these cases
    • Block index
    • Thread index
    • Sourcecode line number
  • Summary of access violations

4.10. CUDA memory checker result

================================================================================
CUDA Memory Checker detected 2 threads caused an access violation:
Launch Parameters
CUcontext = 003868b8
CUstream = 00000000
CUmodule = 0347e780
CUfunction = 03478980
FunctionName = _Z9addKernelPiPKiS1_
gridDim = {1,1,1}
blockDim = {5,1,1}
sharedSize = 0
Parameters:
Parameters (raw):
0x05200000 0x05200200 0x05200400
GPU State:
Summary of access violations:

Parallel Nsight Debug

Memory Checker detected 2 access violations.

error = access violation on store

blockIdx = \{0,0,0\}

threadIdx = \{3,0,0\}

address = 0x05200018

accessSize = 4

4.11. Possible error codes and meanings

- CUDA memory checker error codes:
5. fejezet - CUDA libraries

1. CUBLAS library

1.1. CUBLAS Library

- BLAS: Basic Linear Algebra Subprograms [14]

Basic Linear Algebra Subprograms (BLAS) is a de facto application programming interface standard for publishing libraries to perform basic linear algebra operations such as vector and matrix multiplication. Heavily used in high-performance computing, highly optimized implementations of the BLAS interface have been developed by hardware vendors such as by Intel and Nvidia.

- CUBLAS: CUDA BLAS library

CUBLAS is an implementation of the BLAS library based on the CUDA driver and framework. It has some easy to use data types and functions. The library is self-contained in the API level, so the CUDA is driver is unnecessary.

- Technical details

  - The interface to the CUBLAS library is the header file cublas.h
  - Applications using CUBLAS need to link against the DSO the DLL cublas.dll (for Windows applications) when building for the device,
  - and against the DSO the DLL cublasemu.dll (for Windows applications) when building for device emulation.

1.2. Developing CUBLAS based applications

- Step 1 - Create CUBLAS data structures

  - CUBLAS provides functions to create and destroy objects in the GPU space
  - There are not any special types (like matrices or vector), the library functions usually needs typed pointers to the data structures

- Step 2 - Fill structures with data

  - There are some functions to handle data transfers between the system memory and the GPU memory

- Step 3 - Call CUBLAS function(s)

  - The developer can call a CUBLAS function, or a sequence of these functions

- Step 4 - Retrieve results to system memory

  - Finally the developer can upload the function results from the GPU memory to system memory.

1.3. CUBLAS function result

- The type cublasStatus is used for function status returns

- CUBLAS helper functions return status directly, while the status of CUBLAS core functions can be retrieved via cublasGetError() function

- Currently, the following values are defined:
1.4. CUBLAS helper functions

- `cublasStatus cublasInit( )`

Initializes the CUBLAS library: it allocates hardware resources for accessing GPU. It must be called before any other CUBLAS functions

Return values:

- CUBLAS_STATUS_ALLOC_FAILED: if resources could not be allocated
- CUBLAS_STATUS_SUCCESS: if CUBLAS library initialized successfully

- `cublasStatus cublasShutdown( )`

Shuts down the CUBLAS library: deallocates any hardware resource in the CPU side

Return values:

- CUBLAS_STATUS_NOT_INITIALIZED: if CUBLAS library was not initialized
- CUBLAS_STATUS_SUCCESS: CUBLAS library shut down successfully

- `cublasStatus cublasGetError( )`

Returns the last error that occurred on invocation of any of the CUBLAS core functions (helper functions return the status directly, the core functions do not)

1.5. CUBLAS memory handling

- `cublasStatus cublasAlloc(int n, int elemSize, void **ptr)`

Creates an object in GPU memory space capable of holding an array of n elements, where each element’s size is `elemSize` byte. The result of the function is the common status code, the `ptr` pointer points to the new allocated memory space

- `cublasStatus cublasFree(const void *ptr)`

Deallocates the object in the GPU memory referenced by the `ptr` pointer
• `cublasStatus cublasSetVector(int n, int elemSize, 
  const void *x, 
  int incx,void *y, 
  int incy)`

  The function copies $n$ elements from a vector in the system memory (pointed by $x$ reference) to the $y$ vector in the GPU memory (pointed by the $y$ reference). Storage spacing between elements is $incx$ in the source vector and $incy$ in the destination vector.

• `cublasStatus cublasGetVector(int n, int elemSize, 
  const void *x, 
  int incx,void *y, 
  int incy)`

  Similar to cublasSetVector function. It copies $n$ elements from a vector in the GPU memory (pointed by $x$ reference) to the $y$ vector in the system memory (pointed by the $y$ reference). Storage spacing between elements is $incx$ in the source vector and $incy$ in the destination vector.

### 1.6. BLAS functions overview

• The BLAS functionality is divided into three levels: 1, 2 and 3

• The CUBLAS framework uses the same division method as the original BLAS library

• BLAS level 1 functions
  • This level contains vector operations of the form as well as scalar dot products and vector norms, among other things
  • Functions are grouped into subgroups by the operand types
    • Single-precision BLAS1 functions
    • Single-precision complex BLAS1 functions
    • Double-precision BLAS1 functions
    • Double-precision complex BLAS1 functions

• BLAS level 2 functions
  • This level contains matrix–vector operations, solving equals, among other things

• BLAS level 3 functions
  • This level contains matrix–matrix operations. This level contains the widely used general matrix multiply operation

### 1.7. Some CUBLAS level 1 functions

• `int cublasIsamax(int n, const float *x, int incx)`

  Finds the smallest index of the maximum element (result is 1-based indexing!)

• Parameters:
  • $n$: number of elements in input vector
• x: single-precision vector with n elements
• incx: storage spacing between elements of x

• Error codes:
  • CUBLAS_STATUS_NOT_INITIALIZED: if CUBLAS library was not initialized
  • CUBLAS_STATUS_ALLOC_FAILED: if function could not allocate reduction buffer
  • CUBLAS_STATUS_EXECUTION_FAILED: if function failed to launch on GPU
• float cublasSasum(int n, const float *x, int incx)
• Computes the sum of the values of the elements in the vector

  ... 

  • See the CUBLAS library documentation for full list of available functions

1.8. Some CUBLAS level 2 functions

• void cublasSsmbv(char uplo,
  int n,
  int k,
  float alpha,
  const float *A,
  int lda,
  const float *x,
  int incx,
  float beta,
  float *y,
  int incy)
• Performs the following matrix-vector operation:
  
y = alpha * A * x + beta * y
where
  • alpha, beta – scalars
  • x, y – vectors
  • A – matrix

• void cublasStrsv(char uplo, char trans, char diag, int n,
  const float *A, int lda, float *x, int incx)
Performs the following matrix-solves a system of equations

  ...
1.9. Some CUBLAS level 3 functions

- void cublasSgemm( char transa,
  char transb,
  int m,
  int n,
  int k,
  float alpha,
  const float *A,
  int lda,
  const float *B,
  int ldb,
  float beta,
  float *C,
  int ldc)

- Performs the following matrix-matrix operation:
  \[ C = \alpha \times \text{op}(A) \times \text{op}(B) + \beta \times C \] (where \( \text{op}(x) = x \) or \( \text{op}(x) = x^T \))

- where
  - \( \alpha, \beta \) – scalars
  - \( \text{lda}, \text{ldb}, \text{ldc} \) – leading dims
  - \( A, B, C \) – matrices
  - if \( \text{transa} = "T" \) then \( \text{op}(A) = A^T \)
  - if \( \text{transb} = "T" \) then \( \text{op}(B) = B^T \)

- See the CUBLAS library documentation for full list of available functions
6. fejezet - CUDA versions

1. CUDA 4 features

1.1. CUDA 4.0 features

1.1.1. Share GPUs accross multiple threads

- Easier porting of multi-threaded applications. CPU threads can share one GPU (OpenMP etc.)
- Launch concurrent threads from different host threads (eliminates context switching overhead)
- New, simple context management APIs. Old context migration APIs still supported

1.1.2. One thread can access all GPUs

- Each host thread can access all GPUs
  (CUDA had a „1 thread – 1 GPU” limitation before)
- Single-threaded application can use multi-GPU features
- Easy to coordinate more than GPUs

1.2. Set current device

- All CUDA operations are issued to the “current” GPU (except asynchronous P2P memory copies)
- To select the current device, use `cudaSetDevice`

  ```
  cudaError_t cudaSetDevice(int device)
  ```

- First parameter is the number of the device
- Any device memory subsequently allocated from this host thread using `cudaMalloc()`, `cudaMallocPitch()` or `cudaMallocArray()` will be physically resident on device
- Any host memory allocated from this host thread using `cudaMallocHost()` or `cudaHostAlloc()` or `cudaHostRegister()` will have its lifetime associated with device
- Any streams or events created from this host thread will be associated with device
- Any kernels launched from this host thread using the `<<< >>>` operator or `cudaLaunch()` will be executed on device
- This call may be made from any host thread, to any device, and at any time
- This function will do no synchronization with the previous or new device, and should be considered a very low overhead call

1.3. Current device - streams, events

- Streams and events are per device
  - Streams are created in the current device
  - Events are created in the current device
- NULL stream (or 0 stream)
• Each device has its own default stream

• Default streams of different devices are independents

• Using streams and events

• Streams can contain only events of the same device

• Using current device

• Calls to streams are available only when the appropriate device is current

1.4. Multi-GPU example

• Synchronization between devices

• eventB belongs to streamB and device 1

• At cudaEventSynchronize the current GPU is device 0

6.1. ábra -

```c
1. cudaStream_t streamA, streamB;
2. cudaEvent_t eventA, eventB;
3. cudaSetDevice(0);
4. cudaStreamCreate( &streamA
5. cudaEventCreate( &eventA);
6. cudaSetDevice(1);
7. cudaStreamCreate( &streamB
8. cudaEventCreate( &eventB);
9. kernel<<<..., ..., streamB>>>(.
10. cudaEventRecord(eventB, stre
11. cudaSetDevice(0);
12. cudaEventSynchronize( eventB
13. kernel<<<..., ..., streamA>>>(.
```

1.5. Using multiple CPU threads

• In case of multiple CPU threads of the same process

• GPU handling is the same as single-thread environment

• Every thread can select the current device

• Every thread can communicate to any GPUs

• The process has its own address space, all of the threads can reach this region

• In case of multiple processes

• Processes have their own memory address spaces
• It’s like the processes are on different nodes
• Therefore some CPU side messaging needed (MPI)
• In case of different nodes
  • The CPUs have to solve the communication
  • From the GPUs perspective it is the same as the single-node environment

1.6. Vector multiplication with multiple GPUs - kernel

• Simple kernel to multiply all items in the array by 2

6.2. ábra -

```c
#include "cuda_runtime.h"
#include "device_launch_parameters"
#include <stdio.h>

#define N 100
#define blockN 10
#define MaxDeviceCount 4

__global__ static void VectorMul(float
int i = blockIdx.x * blockDim.x + ti

if (i < NperD) {
}
```

1.7. Vector multiplication with multiple GPUs – memory allocation

• Get information about devices and allocate memory in all devices

6.3. ábra -
1.8. Vector multiplication with multiple GPUs – kernel invocation

- Select one of the devices
- Copy the appropriate part of the input array (asynchronously)
- Start a kernel in the selected device
- Copy back the results to the host memory (asynchronously)
- Do the iteration before for all devices
- After this synchronize all devices

6.4. ábra -

1.9. Vector multiplication with multiple GPUs – kernel invocation
• Free all memory objects in devices

• Print out the results

6.5. ábra -

```c
for(int di = 0; di < deviceCount; di++) {
    cudaFree(devA[di]);
}
for(int i = 0; i < N; i++) {
    printf("A[%d] = \t%f\t%f\n", i, ok
```  

2. CUDA 5 features

2.1. CUDA 5.0 features [26]

2.1.1. Dynamic Parallelism

• GPU threads can dynamically spawn new threads, allowing the GPU to adapt to the data. By minimizing the back and forth with the CPU, dynamic parallelism greatly simplifies parallel programming. And it enables GPU acceleration of a broader set of popular algorithms, such as those used in adaptive mesh refinement and computational fluid dynamics applications.

2.1.2. GPU-Callable Libraries

• A new CUDA BLAS library allows developers to use dynamic parallelism for their own GPU-callable libraries. They can design plug-in APIs that allow other developers to extend the functionality of their kernels, and allow them to implement callbacks on the GPU to customize the functionality of third-party GPU-callable libraries.

• The “object linking” capability provides an efficient and familiar process for developing large GPU applications by enabling developers to compile multiple CUDA source files into separate object files, and link them into larger applications and libraries

2.1.3. GPUDirect Support for RDMA

• Enables direct communication between GPUs and other PCI-E devices, and supports direct memory access between network interface cards and the GPU.

2.2. Dynamic parallelism

2.2.1. Dynamic parallelism example

6.6. ábra -
• Programmer can use kernel launch <<< >>> in any kernel

• Launch is per-thread

• __syncthreads() includes all launches by any thread in the block
7. fejezet - References

1. References


http://courses.ece.uiuc.edu/ece498/al/


