streaming processor

Related terms:

Soft Error, Graphics, Kernel, Memory Access, Multiprocessors, Vulnerabilities

View all Topics

Learn more about streaming processor

Adaptive sparse matrix representation for efficient matrix-vector multiplication

P. Zardoshti, ... H. Sarbazi-Azad, in Advances in GPU Research and Practice, 2017

3.1 Hardware Architectures

The GPU architecture consists of several hundred simple cores called streaming processors (SPs), which are grouped in a set of streaming multiprocessors (SMs). Each SM executes instructions in a single-instruction multiple-threads (SIMT) mode and supports a multithreading execution mechanism. GPUs are capable of utilizing large amounts of memory bandwidth. To further increase usable bandwidth, modern GPUs also contain a series of on-chip caches. In modern GPU architecture, each thread has its own private register. For memory, a thread has its own memory space, which is called the local memory. A block also has its own 64 KB on-chip memory that can be configured as shared memory and L1 cache. All threads in a block can access the same shared memory and L1 cache, while threads in other blocks cannot. The L2 cache is the primary point of data unification between the SM units, servicing all load, store, and texture requests and providing efficient, high-speed data sharing across the GPU. The entire kernel also has a global memory. All threads in a block can access the global memory space. Also, all threads can access read-only constant memory and texture memory spaces [35].

NVIDIA GeForce GTX 480

NVIDIA GeForce GTX 480

The NVIDIA GPU is based on the GF100 multiprocessor architecture with 480 cores. It works with a clock frequency of 1.4 GHz, and has 1.5 GB DRAM with a 177 GB/s bandwidth. This GPU consists of 15 SMs. Each SM has a memory space that can be configured as 48K of shared memory with 16K of L1 cache, or as 16K of shared memory with 48K of L1 cache [35].

The NVIDIA GPU is based on the GF100 multiprocessor architecture with 480 cores. It works with a clock frequency of 1.4 GHz, and has 1.5 GB DRAM with a 177 GB/s bandwidth. This GPU consists of 15 SMs. Each SM has a memory space that can be configured as 48K of shared memory with 16K of L1 cache, or as 16K of shared memory with 48K of L1 cache [35].

NVIDIA Tesla K20X

There are 2688 cores with a clock frequency of 0.73 GHz in the Kepler GK110 architecture. It has 6 GB DRAM with 250 GB/s bandwidth. Kepler allows additional flexibility in configuring memory space of SMs by splitting a 32K/32K split between allocation of shared memory and L1 cache [36]. Table 2 summarizes the specifications of these architectures.

There are 2688 cores with a clock frequency of 0.73 GHz in the Kepler GK110 architecture. It has 6 GB DRAM with 250 GB/s bandwidth. Kepler allows additional flexibility in configuring memory space of SMs by splitting a 32K/32K split between allocation of shared memory and L1 cache [36]. Table 2 summarizes the specifications of these architectures.

Table 2. Architectural Summary of Evaluated Platforms

Table 2. Architectural Summary of Evaluated Platforms

Architecture	Architecture	GTX 480 GeForce	Tesla K20X	GTX 480 GeForce
Model	Model	GF100	GK100	GF100
Core	Core	480	2688	480
Clock (GHz)	Clock (GHz)	1.4	0.73	1.4
DP Peak (GFlop/s)	DP Peak (GFlop/s)	168	1310	168
SP Peak (GFlop/s)	SP Peak (GFlop/s)	1345	3950	1345
Memory bandwidth (GB/s)	Memory bandwidth (GB/s)	177.4	250	177.4

> Read full chapter

> Read full chapter

GPU Acceleration of Particle-in-Cell Gas Dynamic Simulations

GPU Acceleration of Particle-in-Cell Gas Dynamic Simulations

Aldo Frezzotti, ... Livio Gibelli, in GPU Computing Gems Jade Edition, 2012

Aldo Frezzotti, ... Livio Gibelli, in GPU Computing Gems Jade Edition, 2012

15.4.2 Performance Evaluation

The performance analysis has been performed on commercially available GPU GeForce GTX 260 using CUDA version 2.0. The GTX 260 GPU model consists of 24 streaming multiprocessors with 8 streaming processors (SP) clocked at 1.242 GHz each for a peak theoretical performance of 715.4 GFLOP/s in single precision. The GPU has 896 MB of device memory with a memory bandwidth of 111.9 GB/s. The graphic processing unit has been hosted by a personal computer equipped with 4

The performance analysis has been performed on commercially available GPU GeForce GTX 260 using CUDA version 2.0. The GTX 260 GPU model consists of 24 streaming multiprocessors with 8 streaming processors (SP) clocked at 1.242 GHz each for a peak theoretical performance of 715.4 GFLOP/s in single precision. The GPU has 896 MB of device memory with a memory bandwidth of 111.9 GB/s. The graphic processing unit has been hosted by a personal computer equipped with 4

GB of main memory with a memory bandwidth of 12.8 GB/s and an Intel Core Duo Quad Q9300 CPU, running at 2.5 GHz for a peak theoretical performance of a single core of 20 GFLOP/s. The host code has been compiled using the gcc/g++ compiler with optimization option "-O3" whereas no SSE instructions have been used.

GB of main memory with a memory bandwidth of 12.8 GB/s and an Intel Core Duo Quad Q9300 CPU, running at 2.5 GHz for a peak theoretical performance of a single core of 20 GFLOP/s. The host code has been compiled using the gcc/g++ compiler with optimization option “-O3” whereas no SSE instructions have been used.

The efficiency of the BHS and BGKW numerical codes has been assessed by computing the speed-up factor $S = T_{CPU}/T_{GPU}$, where T_{CPU} and T_{GPU} are the times used by the CPU and GPU, respectively. Times are measured after initial setup and do not include the initial and final execution times required to transfer data between the disjoint CPU and GPU memory spaces. Figure 15.2 shows the time in seconds which is spent on the streaming step and on the collision step by the BHS code and the BGKW code for $N_r = 6144$. For the BHS code, the collision step is more time consuming than the streaming step, which takes almost 31% of overall computing time. This is not the case for the BGKW code where the evaluation of the collision integral is less computationally demanding.

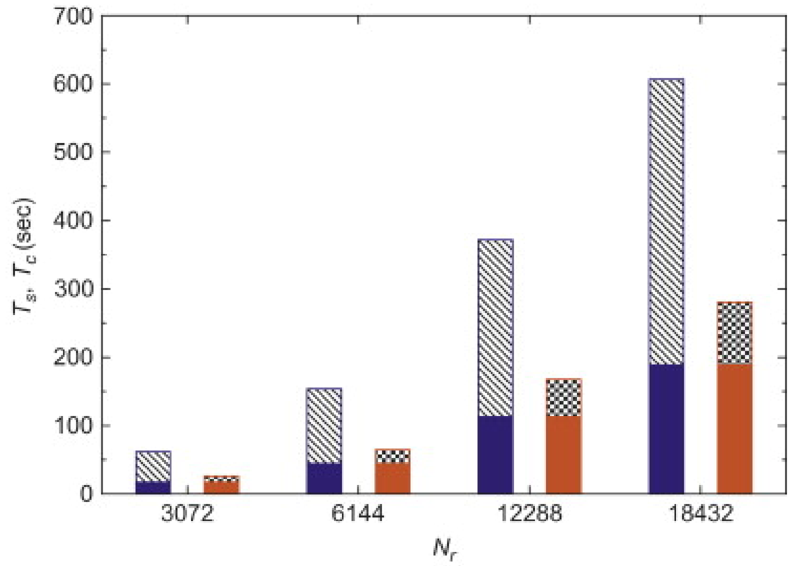
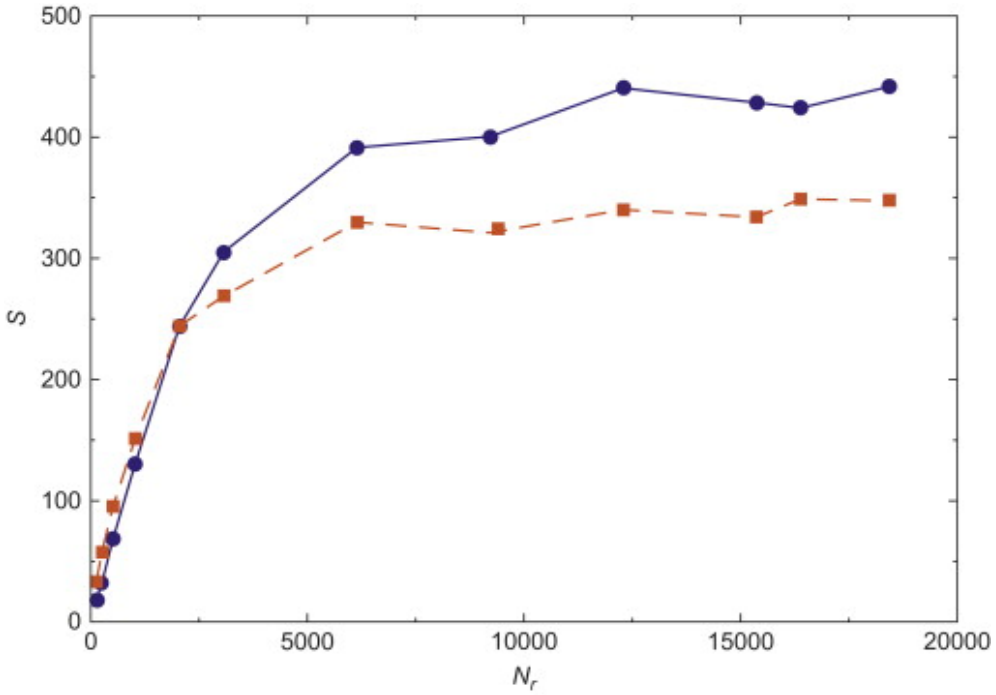


Figure 15.2. Time in seconds spent on the streaming step (dark bar) and on collision step for the BHS code with $N_r = 6144$ (striped bar) and for the BGKW code (checked bar). $N_\nu = 8000$.

Figure 15.3 reports the obtained speed-up factors, S, as a function of the number of spatial grid points, N_r. They grow rapidly with the number of cells, till N_r approximately exceeds 10^4. Afterwards, the speed-up factors level up at about 440 and 350 for the BHS and the BGKW equations, respectively. As shown by the speed-up curves, the GPU power is not fully exploited till the number of concurrent threads reaches a threshold. Beyond, the speed-up saturates and the computing time approximately behaves as a linear function of N_r.


500
400
300
200
100
0
S
0
5000
10000
15000
20000
N_r

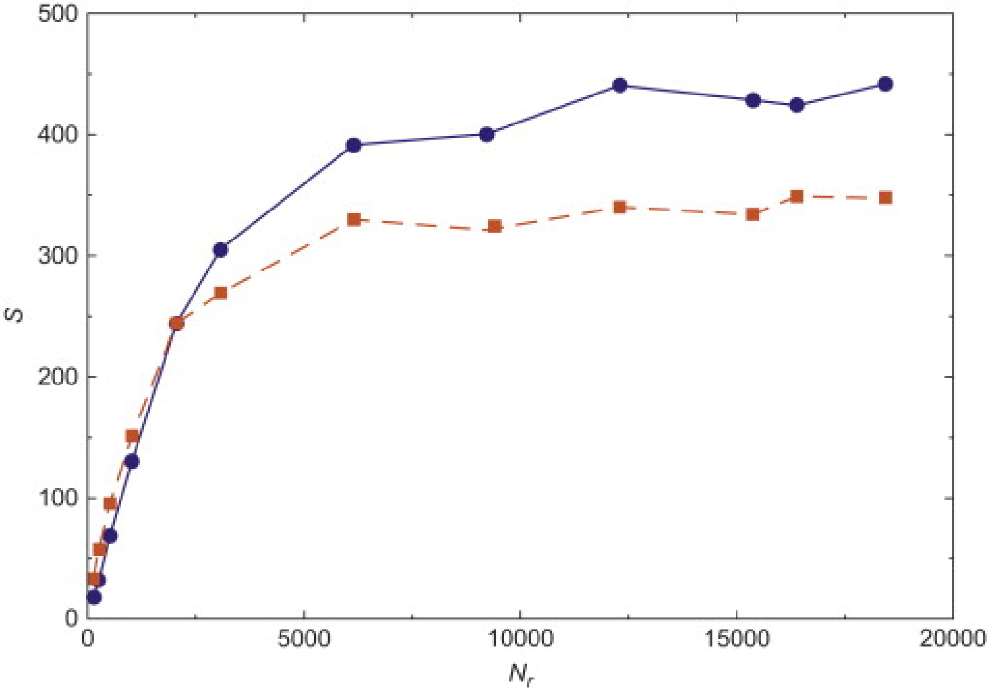


Figure 15.3. Speed-up, S, versus the number of cells in the physical space, N_r. Circles: BHS code with $N_t = 1544$. Squares: BGKW code. $N_\nu = 8000$.

The GPU-based codes can be further analyzed by a simplified evaluation of their ideal performance. A single application of the upwind scheme requires the execution of 7 floating point operations and 2.3 accesses to the global memory. The GPU delivers 715.4 GFLOP/s but the transfer rate to/from the main memory is limited to 111.9 GB/s. Since the ratio of number of floating point operations to the number of bytes accessed is also (7 : 9.2), it is reasonable to obtain the number of floating point operation per second from the transfer rate alone. Hence, the ideal number of GFLOP/s can be obtained by assuming that 7 floating point operations will be executed in the time required to transfer 9.2 bytes from the main memory. Accordingly, this simple argument yields an ideal performance of 85.1 GFLOP/s. Using similar arguments leads to an estimate of the ideal performance of the collision step of 153.8 GFLOP/s and 270.4 GFLOP/s for the BHS and BGKW codes, respectively. Timing the execution of the separate kernels and counting the number of associated floating point operations provides the real kernel's GFLOP/s. As shown by Figure 15.4, it results that the performance of the streaming kernel grows with N_r and quickly levels at about 23 GFLOP/s, approximately one fourth of the estimated ideal performance. The difference can be justified by observing that the real CUDA implementation of the finite difference scheme is not free from thread divergence and ancillary tasks whose effects can be evaluated with difficulty [8]. For both the BHS and BGKW collision integrals, the collision kernel's performance closely patterns the speed-up behavior. They rapidly grow in the range $N_r < 10^4$ and then level up at about 130 GFLOP/s and 190 GFLOP/s, respectively. The main reason why the collision kernel performs better than the streaming kernel is its higher FLOPs to memory operation ratio which allows a more efficient use of GPU computing

power. Altogether, the codes which solve the BHS and BGKW equations show a performance of 100 and 75 GFLOP/s, respectively.

power. Altogether, the codes which solve the BHS and BGKW equations show a performance of 100 and 75 GFLOP/s, respectively.

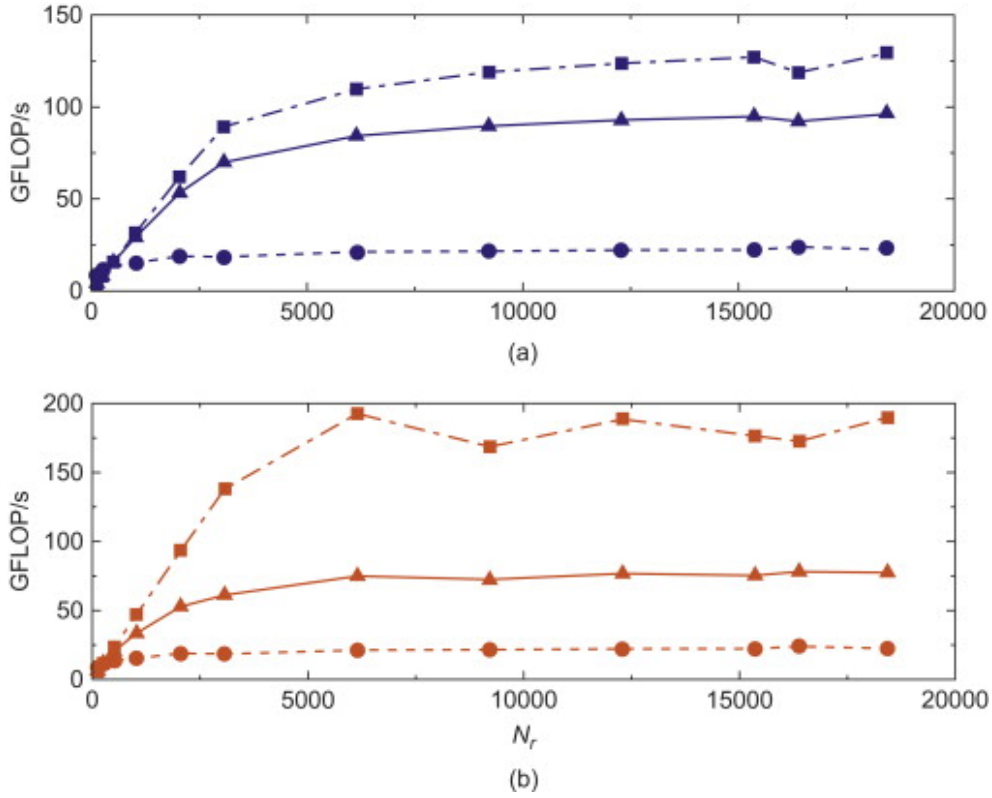


Figure 15.4. Real GFLOP/s versus the number of cells in the physical space, N_r. (a) BHS code with $N_t = 64$, (b) BGKW code with $N_v = 8000$. Dashed lines with circles: streaming step. Dot-dashed lines with squares: collision step. Solid lines with triangles: overall code.

A similar analysis can be performed for the CPU-based codes. The time required by the CPU to execute the floating point operations is negligible with respect to the time related to the transfer of data to/from the memory and, hence, it has to be taken into account in the estimate of the GFLOP/s. The number of accesses to the global memory is supposed to be the same as above whereas the CPU maximum bandwidth and performance per single core are about 12.8 GB/s and 20 GFLOP/s, respectively. Therefore the sequential codes which solve the BHS and BGKW equations may ideally deliver 9.9 GFLOP/s and 11.1 GFLOP/s, respectively, with the effective performance which are about 0.28 GFLOP/s and 0.34 GFLOP/s. This suggests that the high speed-up factors shown in Figure 15.3 are mainly due to the fact that the sequential codes are not fully optimized. However, it is worth noticing that the ideal analysis discussed above is less accurate for the CPU than for the GPU. For instance, as indicated by benchmark tests, the CPU effective bandwidth is approximately one-half of the maximum bandwidth whereas this is not the case for the GPU. Accordingly, the above ideal GFLOP/s of the CPU-based codes should be halved. Moreover, the performance of the sequential codes we developed are comparable

to the performance of similar codes described in literature. For instance, in Ref. [9] the solution of the cavity flow problem is obtained by integrating the linearized BGKW equation in about 3 hours. Using the execution time of this sequential code to compute the speed-up factor yields a result as high as 265. It is worth noticing that in Ref. [9] the number of velocity variables has been reduced by a standard projection method [1]. The value of the speed-up factor given above has been obtained by modifying the parallel code to take advantage of the same reduction technique. To the best of the authors' knowledge, no direct solutions of the cavity flow problem based on the BHS equation have been provided until now. However, a similar reasoning can be applied to the one-dimensional unsteady Couette flow. In Ref. [11] its solution is achieved in approximately 9 hours, which gives a speed-up factor of the BHS code of about 54. This results is still lower than the speed-up factor reported in Figure 15.3, but this is due to the fact that the discretizations used to provide solutions as accurate as the ones reported in Ref. [11] does not allow to fully exploit the computational power of the GPU.

to the performance of similar codes described in literature. For instance, in Ref. [9] the solution of the cavity flow problem is obtained by integrating the linearized BGKW equation in about 3 hours. Using the execution time of this sequential code to compute the speed-up factor yields a result as high as 265. It is worth noticing that in Ref. [9] the number of velocity variables has been reduced by a standard projection method [1]. The value of the speed-up factor given above has been obtained by modifying the parallel code to take advantage of the same reduction technique. To the best of the authors' knowledge, no direct solutions of the cavity flow problem based on the BHS equation have been provided until now. However, a similar reasoning can be applied to the one-dimensional unsteady Couette flow. In Ref. [11] its solution is achieved in approximately 9 hours, which gives a speed-up factor of the BHS code of about 54. This results is still lower than the speed-up factor reported in Figure 15.3, but this is due to the fact that the discretizations used to provide solutions as accurate as the ones reported in Ref. [11] does not allow to fully exploit the computational power of the GPU.

We conclude by observing that the test problem examined here has shown that the size of physical memory is the main obstacle toward the application to complex two- or three-dimensional flows, not the number crunching capability.

> Read full chapter

Advances in COMPUTERS

Jack Dongarra, ... Mustafa Tikir, in Advances in Computers, 2008

3.3.3.1 Molecular Dynamics.

As with the Cell, implementation for the GPU focused on the part of the algorithm that calculates new accelerations from only the locations of the atoms and several constants. For our streaming processor, then, the obvious choice is to have one input array comprising the positions and one output array comprising the new accelerations. The constants were compiled into the shader program source using the provided JIT compiler at program initialization.

We set up the GPU to execute our shader program exactly once for each location in the output array. That is, each shader program calculates the acceleration for one atom by checking for interaction with all other atoms and accumulating forces into a single acceleration value for the target atom. Instruction length limits prevent us from searching more than a few thousand atoms in a single pass, and so with 4096 atoms or more, the algorithm switches to use multiple passes through the input array. After the GPU is completed, the resulting accelerations are read back

into main memory, where the host CPU proceeds with the current time step. At the next time step, the updated positions are re-sent to the GPU and new accelerations computed again.

into main memory, where the host CPU proceeds with the current time step. At the next time step, the updated positions are re-sent to the GPU and new accelerations computed again.

Figure 13 shows performance using an NVIDIA GeForce 7900GTX GPU. This figure includes results from the GPU's host CPU (a 2 GHz Opteron) as a reference for scaling comparison. The readily apparent change in slope for the GPU below 1024 atoms shows the point at which the overheads associated with offloading this acceleration computation to the GPU become a more significant fraction of the total runtime than the $O(N^2)$ calculation itself. These constant and $O(N)$ costs for each time step include sending the positions and reading the acceleration array across the PCIe bus every time step, and these results show that there is a lower bound on problem size where a GPU will not be faster than the CPU. However, the massive parallelism of the GPU helps it maintain a consistent speedup above 1000 atoms.

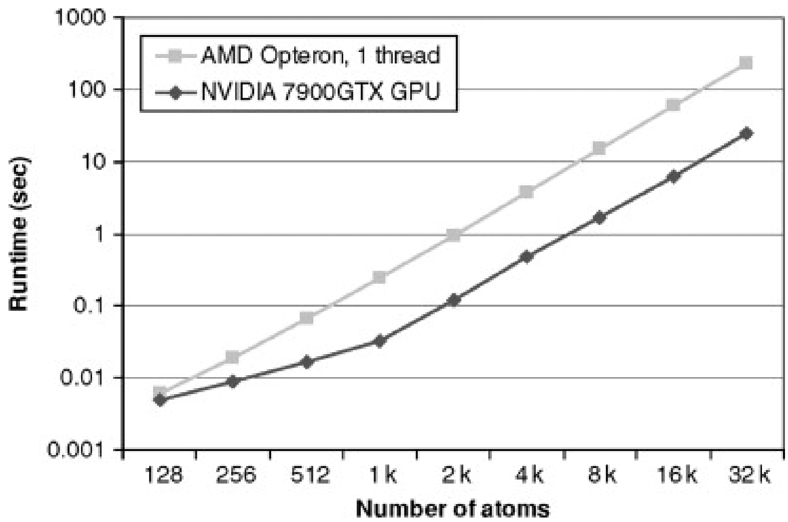


Fig. 13. Performance results on GPU with CPU scaling for comparison.

> Read full chapter

GPU alignment of two and three sequences

J. Li, ... S. Sahni, in Advances in GPU Research and Practice, 2017

2 GPU architecture

The GPU algorithms described in this chapter target the NVIDIA Tesla C2050 GPU, which is also known as Fermi. The C2050 has 14 streaming multiprocessors (SMs) and each SM has 32 streaming processors (SPs) giving the C2050 a total of

448 SPs or cores. Fig. 1 shows the architecture of a C2050 SM. Although each SP of a C2050 has its own integer, single- and double-precision units, the 32 SPs of an SM share 4 single-precision transcendental function units. An SM has 64 KB of on-chip memory that can be "configured as 48 KB of shared memory with 16 KB of L1 cache (default setting) or as 16 KB of shared memory with 48 KB of L1 cache" [25]. Additionally, there are 32K 32-bit registers per SM and 3 GB of off-chip device/global memory that is shared by all 14 SMs. The peak performance of a C2050 is 1288 GFlops (or 1.288 TFlops) of single-precision operations, and 515 GFlops of double-precision operations and the power consumption is 238 W [26]. Once again, the peak of 1288 GFlops requires that MADDs and SF instructions be dual-issued. When there are MADDs alone, the peak single-precision rate is 1.03 GFlops.

448 SPs or cores. Fig. 1 shows the architecture of a C2050 SM. Although each SP of a C2050 has its own integer, single- and double-precision units, the 32 SPs of an SM share 4 single-precision transcendental function units. An SM has 64 KB of on-chip memory that can be "configured as 48 KB of shared memory with 16 KB of L1 cache (default setting) or as 16 KB of shared memory with 48 KB of L1 cache" [25]. Additionally, there are 32K 32-bit registers per SM and 3 GB of off-chip device/global memory that is shared by all 14 SMs. The peak performance of a C2050 is 1288 GFlops (or 1.288 TFlops) of single-precision operations, and 515 GFlops of double-precision operations and the power consumption is 238 W [26]. Once again, the peak of 1288 GFlops requires that MADDs and SF instructions be dual-issued. When there are MADDs alone, the peak single-precision rate is 1.03 GFlops.

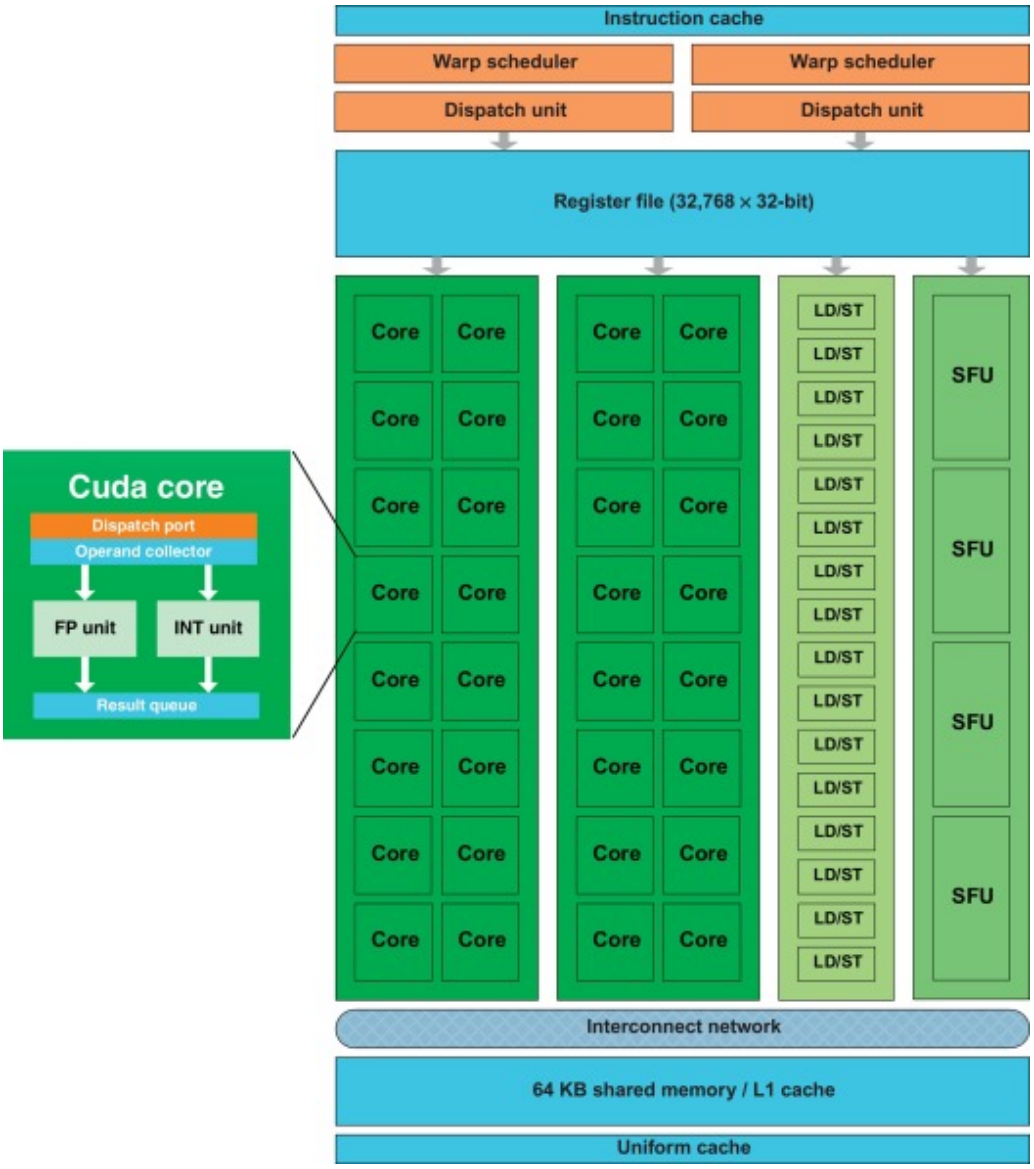


Fig. 1. Architecture of a NVIDIA Fermi [27].

Fig. 1. Architecture of a NVIDIA Fermi [27].

In NVIDIA parlance, the C2050 is a compute capability 2.0 device. The key challenge in deriving high performance on GPUs is to effectively minimize the memory traffic between the SMs and the GPU memory (the global memory of the GPU). Data that

is used repeatedly should go to registers or shared memory, while data that is used less frequently but of larger size should go to device memory.

is used repeatedly should go to registers or shared memory, while data that is used less frequently but of larger size should go to device memory.

> Read full chapter

> Read full chapter

GPGPU: General-Purpose Computing on the GPU

GPGPU: General-Purpose Computing on the GPU

Ying Tan, in GPU-based Parallel Implementation of Swarm Intelligence Algorithms, 2016

Ying Tan, in GPU-based Parallel Implementation of Swarm Intelligence Algorithms, 2016

2.3.4 Single-Instruction, Multiple-Thread (SIMT)

2.3.4 Single-Instruction, Multiple-Thread (SIMT)

A CUDA-enabled GPU has a scalable array of multithreaded streaming multiprocessors (SMs), which is roughly equivalent to CPU cores. Each SM can have certain number of scalar processors (i.e., streaming processors, SPs) with respect to the specific architecture.

A CUDA-enabled GPU has a scalable array of multithreaded streaming multiprocessors (SMs), which is roughly equivalent to CPU cores. Each SM can have certain number of scalar processors (i.e., streaming processors, SPs) with respect to the specific architecture.

When a CUDA program on the host CPU invokes a kernel grid, all blocks are distributed equally to the SMs with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor, and multiple thread blocks can execute concurrently on one multiprocessor. As running blocks finish, new blocks are launched on the vacated SMs, as illustrated in Fig. 2.9.

When a CUDA program on the host CPU invokes a kernel grid, all blocks are distributed equally to the SMs with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor, and multiple thread blocks can execute concurrently on one multiprocessor. As running blocks finish, new blocks are launched on the vacated SMs, as illustrated in Fig. 2.9.

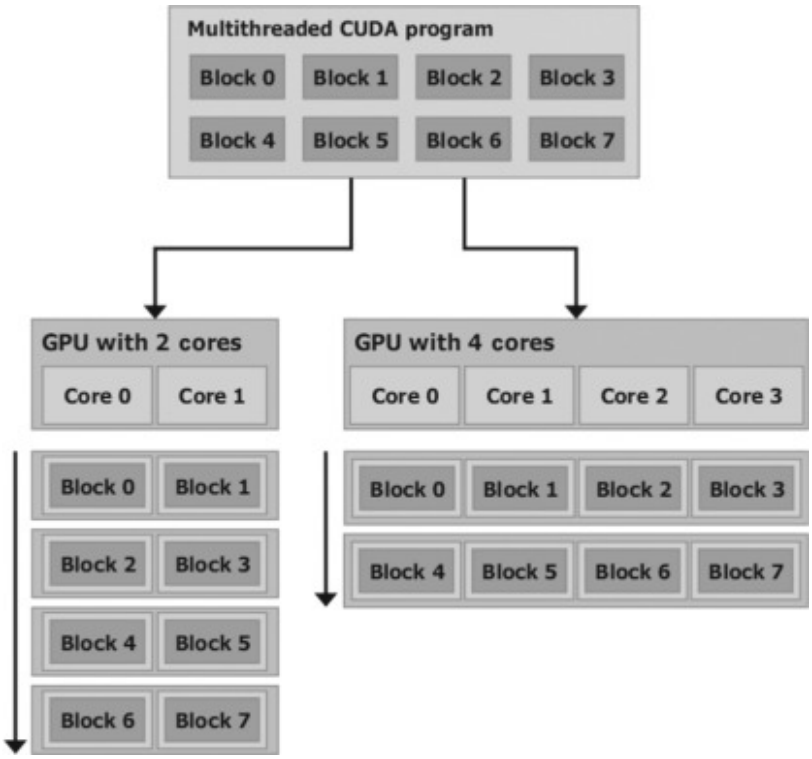


Fig. 2.9. Automatic Scalability

Fig. 2.9. Automatic Scalability

To manage such a large amount of threads, it employs a unique architecture called single-instruction, multiple-thread (SIMT).

The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and therefore free to branch and execute independently.

When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps and each warp gets scheduled by a warp scheduler for execution. The way a block is partitioned into warps is always the same; each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0.

A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths.

> Read full chapter

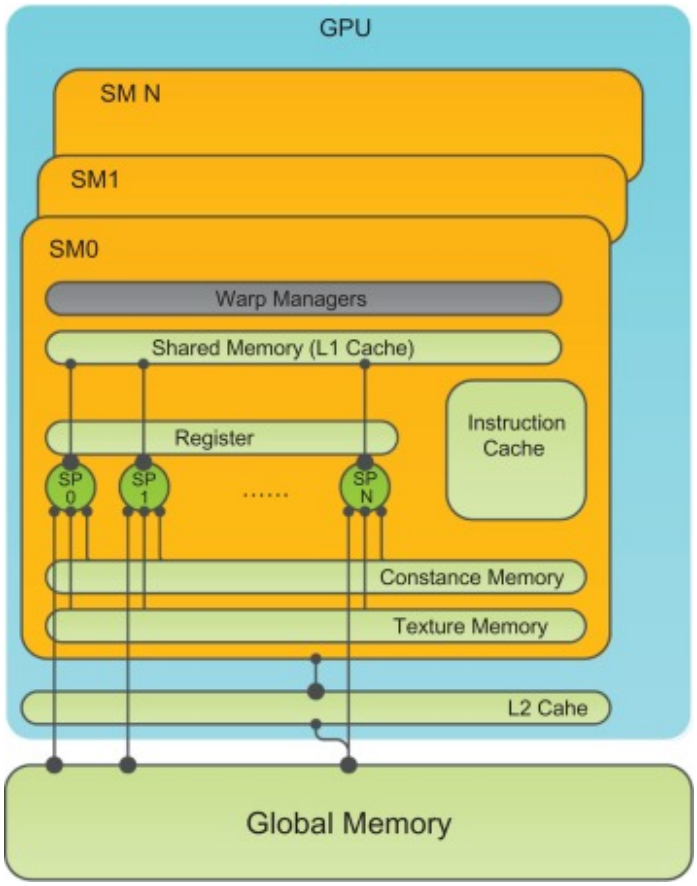
Deep Learning and Its Parallelization

X. Li, ... W. Zheng, in Big Data, 2016

The architecture of GPU

GPU is connected to the host device through the PCI Express (PCIe) and it has its own device memory that is to serve high throughput to modern GPU hardware mainly consists of many streaming multiprocessors (SMs), and streaming processors (SPs).

GPU is an array of SMs that consist of SPs (Fig. 2.10). Each SM can execute in parallel with the others. NVIDIA Kepler K40 has 15 SMs and 2880 cores and each core can execute a sequential thread. SIMT (single instruction, multiple thread). Each SM is associated with memory (L1 cache), read-only texture memory, registers, L2 cache, etc. Each core has a fully pipelined integer ALU, a floating point unit, floating point compare, move, and conversion logic, and a branch unit [17,18].


GPU
SM N
SM1
SM0
Warp Managers
Shared Memory (L1 Cache)
Register
Instruction Cache
SP 0
SP 1
.....
SP N
Constance Memory
Texture Memory
L2 Cahe
Global Memory

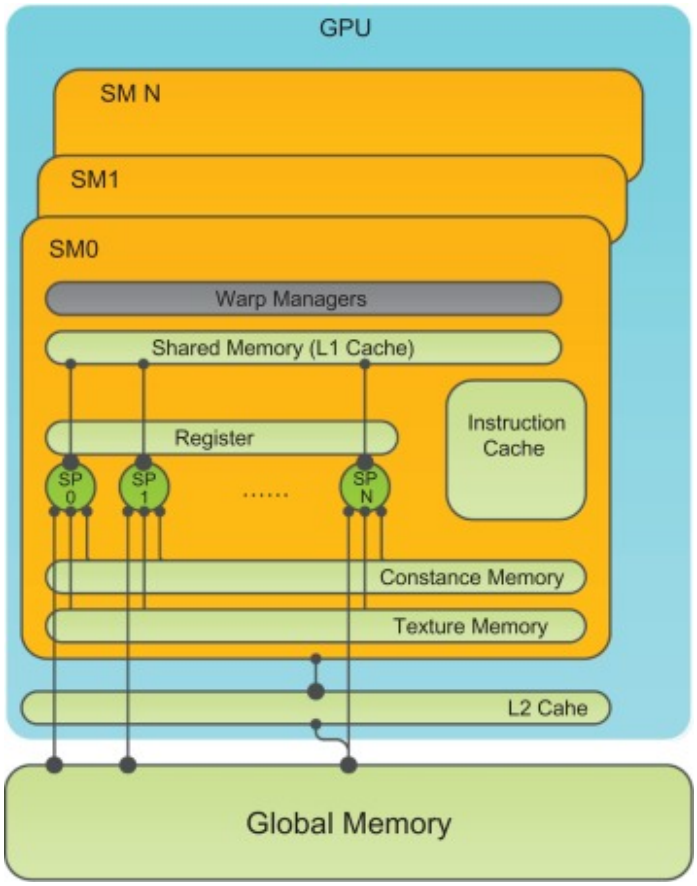


Fig. 10. A simplified architecture of GPU hardware.

Each thread can access its own registers at very high speed in parallel, and we also refer to shared memory as on-chip memory. Registers are private to individual threads, and each thread can only access its own registers. Thirty-two threads are grouped into a warp, which is scheduled by a warp manager, and the threads in a warp are executed in SIMT mode. The computational ability can be enhanced by extending the number of SMs and memory resources.

> Read full chapter

GPU programming

Gerassimos Barlas, Multicore and GPU Programming, 2015

6.3 CUDA's execution model: Streaming multiprocessors and warps

GPU cores are essentially vector processing units, capable of applying the same instruction on a large collection of operands. So, when a kernel is run on a GPU core, the same instruction is synchronously executed by a large collection of processing units called SPs. A group of SPs that execute under the control of

a single control unit is called a streaming multiprocessor, or SM. A GPU can contain multiple SMs, each running each own kernel. Since each thread runs on its own SP, we will refer to SPs as cores (Nvidia documentation calls them CUDA cores), although a more purist approach would be to treat SMs as cores. Nvidia calls this execution model Single-Instruction, Multiple Threads (SIMT).

cution context that is maintained on-chip. This contradicts the arrangement used by multithreading on CPUs, where a very expensive context switch (involving the saving of CPU registers) accompanies thread switching.

cution context that is maintained on-chip. This contradicts the arrangement used by multithreading on CPUs, where a very expensive context switch (involving the saving of CPU registers) accompanies thread switching.

Each SM can have multiple warp schedulers, e.g., in Kepler there are four. This means that up to four independent instruction sequences from four warps can be issued simultaneously. Additionally, each warp scheduler can issue up to two instructions as long as they are independent, i.e., the outcome of one does not depend on the outcome of the other. This is known as *instruction-level parallelism* (ILP). As an example, let's consider the following code:

```
a = a □ b;
d = b + e;
```

These two statements can be executed concurrently, whereas the following example does not present such an opportunity due to the dependency between the two statements:

```
a = a □ b;
d = a + e; // needs the value of a
```

Once an SM completes the execution of all the threads in a block, it switches to a different block in the grid. In reality, each SM may have a large number of *resident blocks* and *resident warps*, i.e., executing concurrently. In compute capability 3.x devices, each SM[10] has 192 CUDA cores. So, how can we maximize the utilization of an SM if only 32 threads are active at any time? Obviously the answer is that we cannot, unless we have multiple warps running on the SM. Each SM of a compute capability 3.x device has four warp schedulers, which means it can direct four different instruction sequences. This would still leave 192 – 4 □ 32 = 64 cores unused. These are utilized by the SM for running more warps in case there is a stall.

In the Maxwell architecture, Nvidia has reduced the number of cores to 128 per SM, but they retain the four warp schedulers. Each scheduler is permanently assigned to a group of 32 cores, simplifying scheduling and improving efficiency, despite the overall reduction in cores.

The number of resident kernels, blocks, and warps depends on the memory requirements of a kernel and the limits imposed by the compute capability of a device. These limits are shown in Table 6.3.

Table 6.3. Compute Capabilities and Associated Limits on Kernel and Thread Scheduling

Item	Compute Capability					
	1.0, 1.1	**1.2, 1.3**	**2.x**	**3.0**	**3.5**	**5.0**

Concurrent kernels/device

of a conditional operation leads them to different paths? The answer is that all the divergent paths are evaluated (if threads branch into them) in sequence until the paths merge again. The threads that do not follow the path currently being executed are stalled. So, given the kernel in Listing 6.4, the execution of a warp would be as shown in Figure 6.4.

of a conditional operation leads them to different paths? The answer is that all the divergent paths are evaluated (if threads branch into them) in sequence until the paths merge again. The threads that do not follow the path currently being executed are stalled. So, given the kernel in Listing 6.4, the execution of a warp would be as shown in Figure 6.4.

```
        doSmtElse();
    }
    doFinal();
}
```

Listing 6.4. An example of a kernel that would slash the utilization of an SM in half by keeping half the threads in a warp stalled.

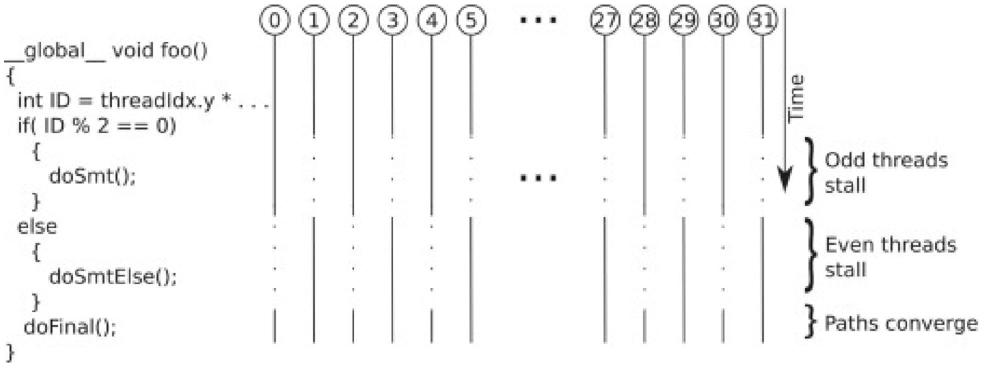


Figure 6.4. An illustration of the execution of the kernel in Listing 6.4 by a warp of threads. The dotted lines indicate a stall.

```
__global__ void foo()
{
    int ID = threadIdx.y * blockDim.x + threadIdx.x;
    if( ID % 2 == 0)
      {
        doSmt();
      }
    else
      {
```

A solution to this problem is discussed in Section 6.7.2.

> Read full chapter

Addressing hardware reliability challenges in general-purpose GPUs

J. Tan, X. Fu, in Advances in GPU Research and Practice, 2017

Dynamic warp formation

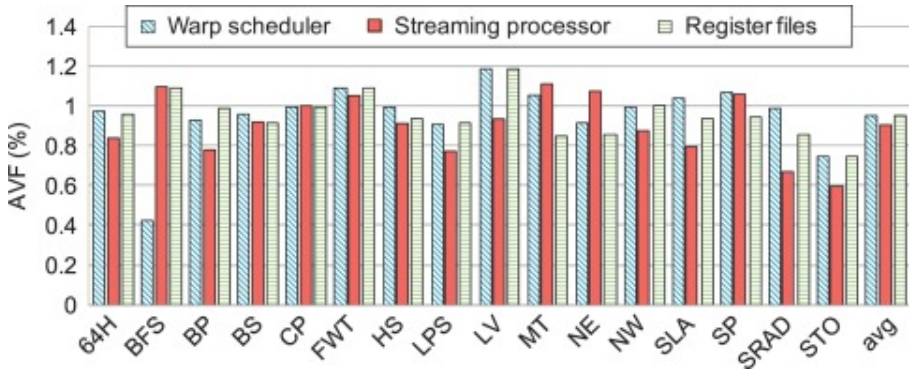
Branch divergence is a major cause for performance degradation in GPGPUs. As we discussed earlier, the immediate postdominator (PDOM) lacks the capability to reconverge threads at the beginning for branch divergence to further improve

the performance. DWF is proposed in Ref. [24] to efficiently handle the threads' divergence. It groups threads from multiple warps but branching to the same target into a new and complete warp, and issues it into the SIMD pipeline. Therefore the parallel streaming processors in the SM are fully utilized, and their performance is enhanced.

the performance. DWF is proposed in Ref. [24] to efficiently handle the threads' divergence. It groups threads from multiple warps but branching to the same target into a new and complete warp, and issues it into the SIMD pipeline. Therefore the parallel streaming processors in the SM are fully utilized, and their performance is enhanced.

In our baseline GPGPUs, a warp splits into multiple warps at the occurrence of the branch divergence, those spawned warps have to compete for the issued bandwidth to leave the warp scheduler. They delay the issue time of other warps and increase the warp residency time. DWF regroups the divergent threads into new warps. It reduces the number of warps competing for the issue bandwidth and shrinks the warp waiting time. Meanwhile, DWF aims to improve the performance and reduce the kernel execution time. Note that reducing the warp residency time and correspondingly the ACE instruction residency time can help to decrease AVF, but shrinking the execution time will increase the AVF as it increases the ACE instruction quantity per cycle [27]. Therefore the impact of DWF on the warps scheduler's vulnerability can be negative or positive, depending on the warp residency time and the kernel execution time. When the reduction of warp residency time dominates that of the kernel execution time, the warp scheduler's AVF decreases, and vice versa. Since the warps enter the SIMD pipeline earlier, registers are written/read faster, which helps to reduce their lifetime. Similar to the warp scheduler, the DWF could increase/decrease the registers' AVF.

Fig. 6 presents the structures' AVF under the impact of DWF, they are normalized to the baseline case with PDOM applied for comparison. Benchmarks *KM*, *LIB*, *MRIF*, *NN*, and *ST3D* cannot execute when DWF is enabled, so we eliminated those benchmarks from this figure. The results of the L1 caches and shared memory are not shown in Fig. 6 and the following figures since they are always robust to the soft-error strikes under different architectural optimizations. On average, DWF optimizes the warp scheduler and the registers' reliability by 5% and 5%, respectively. Obviously, most benchmarks studied here fall into the category in which DWF has positive effect on these two structures' reliability. Note that the capability of DWF to optimize a structure's reliability is significantly affected by the percentage of branch divergence operations in the benchmark. When there are few branch divergences, DWF has limited opportunities to explore the warp formation, whose effect on reliability is trivial. For example, the warp scheduler AVF decreases only 1% in *CP* with 2% branch divergence operations. On the other hand, DWF helps to improve the warps scheduler's reliability by 60% in *BFS* with 40% branch divergence operations.


Warp scheduler
Streaming processor
Register files
AVF (%)
1.4
1.2
1
0.8
0.6
0.4
0.2
0
64H
BFS
BP
BS
CP
FWT
HS
LPS
LV
MT
NE
NW
SLA
SP
SRAD
STO
avg

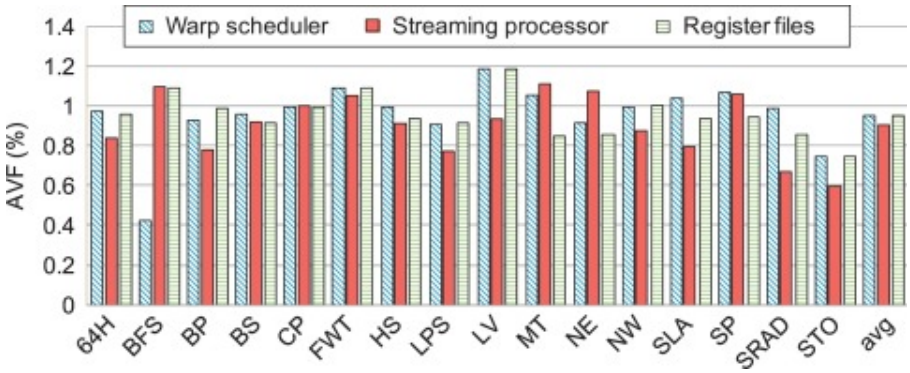


Fig. 6. The normalized AVF of GPGPU streaming multiprocessor microarchitecture structures under the impact of DWF.

The streaming processor's AVF is highly related to the kernel execution time. Since the instruction count per kernel is mainly determined by the instruction type, a shorter kernel execution time implies an increased quantity of ACE instructions per cycle in the streaming processor. And the streaming processors are more susceptible to soft errors when the execution time becomes shorter. Intuitively, DWF would hurt the streaming processor's reliability because it targets on optimizing performance and reduces the kernel execution time. Interestingly, DWF decreases streaming processor's AVF by 10% on average, as shown in Fig. 6. This is because DWF decreases the number of committed instructions (as shown in Fig. 7). When randomly regrouping threads into new warps, DWF may lose opportunities to combine memory accesses originally from the same warp, and introduce extra off-chip memory transactions, which negatively affect the performance. When the benefit of DWF in terms of reducing execution cycles outweighs the increased memory access requests, performance starts to degrade.

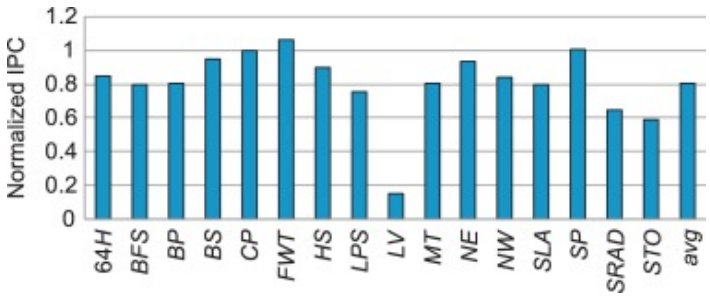


Fig. 7. The normalized IPC under DWF.

> Read full chapter

An Introduction to OpenCLTM

David B. Kirk, Wen-mei W. Hwu, in Programming Massively Parallel Processors (Second Edition), 2013

14.3 Device Architecture

14.3 Device Architecture

Like CUDA, OpenCL models parallel computing systems as a host and one or more OpenCL devices. The host is a traditional CPU that executes the host program. Figure 14.2 shows the conceptual architecture of an OpenCL device. Each device consists of one or more compute units (CUs) that correspond to CUDA streaming multiprocessors (SMs). However, a CU can also correspond to CPU cores or other types of execution units that can support parallel execution such as DSPs and FPGAs.

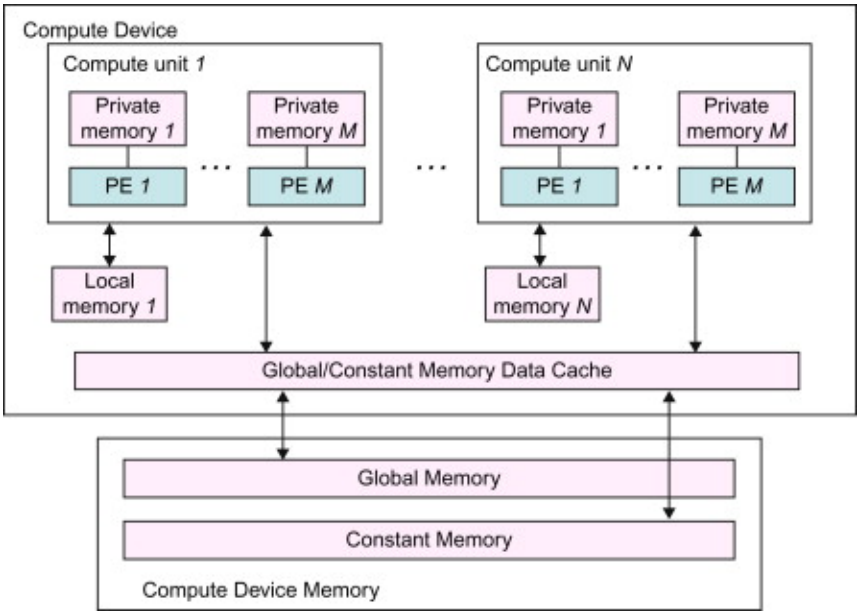


Figure 14.2. Conceptual OpenCL device architecture.

Each compute unit, in turn, consists of one or more processing elements (PEs), which corresponds to the streaming processors (SPs) in CUDA. Computation on a device ultimately happens in individual PEs.

Like CUDA, OpenCL also exposes memory types that can be used by programmers. Figure 14.2 illustrates the 4 memory types: global, constant, local, and private. Table 14.3 summarizes the OpenCL memory types and the mapping of these memory types to CUDA memory types. The OpenCL global memory corresponds to the CUDA global memory. Like CUDA, the global memory can be dynamically allocated by the host program and supports read/write access by both host and devices.

Table 14.3. Mapping of OpenCL Memory Types to CUDA Memory Types

Memory Type	Memory Type	Host Access	Device Access	CUDA Equivalent
Global memory	Global memory	Dynamic allocation; read/write access	No allocation; read/write access by all work items in all work groups, large and slow,	Dynamic allocation; read/write access	Global memory

		but may be cached in some devices		
Constant memory	Constant memory; dynamic allocation; read/write access	Static allocation; read-only access by all work items	Dynamic allocation; read/write access	Constant memory
Local memory	Local memory; dynamic allocation; no access	Static allocation; shared read/write access by all work items in a work group	Dynamic allocation; no access	Shared memory
Private memory	Private memory; no allocation; no access	Static allocation; read/write access by a single work item	No allocation; no access	Registers and local memory

Unlike CUDA, the constant memory can be dynamically allocated by the host. Like CUDA, the constant memory supports read/write access by the host and read-only access by devices. To support multiple platforms, OpenCL provides a device query that returns the constant memory size supported by the device.

The mapping of OpenCL memory types to CUDA memory types is more interesting. The OpenCL local memory corresponds to CUDA shared memory. The OpenCL local memory can be dynamically allocated by the host or statically allocated in the device code. Unlike the CUDA shared memory, the OpenCL local memory cannot be accessed by the host and it supports read/write access by all work items in a work group. The private memory of OpenCL corresponds to the CUDA local memory.

> Read full chapter

> Read full chapter