Performance Analysis and Tuning for General-Purpose Graphics Processing Units (GPGPU)

September 20, 2012

Contents

Chapter 1

GPU Design, Programming, and Trends

This book aims to help readers understand the key performance issues that arise when programming on general-purpose graphics processing unit (GPGPU) hardware. Although there are many excellent resources available with similar aims, this book emphasizes general principles in algorithm design and how these are translated into low-level GPGPU architectures and programming. It includes a brief overview of GPGPU architectures and programming (Chapter 1), high-level algorithmic design theory (Chapter 2), a case study in translating theory into performance engineering practice on GPGPUs (Chapter 3), and a survey of the current state-of-the-art in lower-level performance modeling and analysis for GPGPUs (Chapter 4). Taken together, we hope this material provides a unique end-to-end view of performance understanding for GPGPUs.

1.1 A Brief History of GPU

As their name suggests, graphics processing units (GPUs) were originally attached to video cards designed specifically to accelerate graphics rendering and display. These early GPUs employed fixed graphics pipelines, which then evolved into programmable cores over successive hardware generations.

The first general-purpose GPU (GPGPU) programming interfaces were limited to shader languages such as DirectX [80], OpenGL [74] and Cg [64]. As a result, programs could

only be expressed in terms of graphics pipeline operations. Later, Brook+ [3] and Sh [66] added stream extensions to the C language to abstract away the graphics hardware details. GPU accelerator [88] moved one step further from stream programming by providing data-parallel arrays with aggregate element-wise operations.

While these early research efforts helped to improve general-purpose programmability of GPUs compared to shader languages, the inflection point in GPGPU computing occurred when NVIDIA introduced a new programmable unified graphics and compute processor, known as the G80 family. This architecture included a new combined software and hardware architecture called the Compute Unified Device Architecture (CUDA), which played a significant role in making GPGPU computing popular. Most recently, the more general term *heterogeneous computing* has subsumed GPGPU programming, particularly with the introduction of more general programming models, such as OpenCL, OpenACC, and Microsoft C++ AMP. This book focuses on GPGPU programming rather than either the more specialized notion of graphics programming or the more general notion of heterogeneous computing. Additionally, we focus only on the GPGPU aspects of the GPU architecture, rather than those specific to graphics applications.

1.2 A Brief Overview of a GPU System

In this section, we provide a brief overview of both a baseline GPU architecture, based primarily on NVIDIA's G80/Fermi architecture, and GPGPU programming using the CUDA programming model.

Figure 1.1 shows how a GPU is typically connected with a modern processor. A GPU is an accelerator (or a co-processor) that is connected to a host processor (typically a conventional general-purpose CPU processor). The host processor and GPU communicate to each other via PCI Express (PCIe) that provides 4 Gb/s (Gen 2) or 8 Gb/s (Gen 3) interconnection bandwidth. This communication bandwidth often becomes one of the biggest bottlenecks; thus, it is critical to offload the work to GPUs only if the benefits of using

GPUs outweigh the offload cost. The communication bandwidth is expected to grow as the CPU bus bandwidth of the system memory increases in the future.

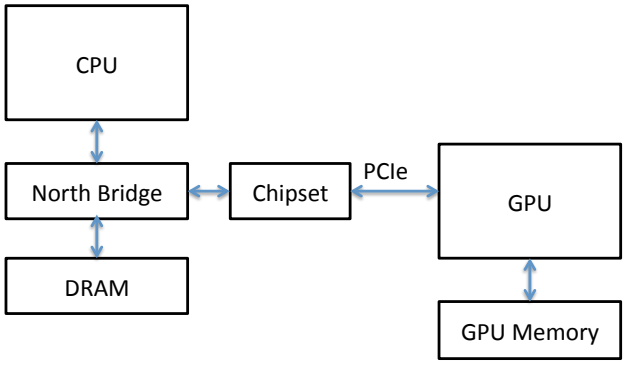


Figure 1.1: A system overview with CPU and a discrete GPU.

1.2.1 An Overview of GPU Architecture

Figure 1.2 illustrates the major components of a general-purpose graphics processor, based on a simplified diagram of G80. At a high-level, the GPU architecture consists of several *streaming multiprocessors* (SMs), which are connected to the GPU's DRAM. (NVIDIA callas an SM and AMD calls a Compute Unit (CU)) Each SM has a number of single-instruction multiple data (SIMD) units, also called stream processors (SPs), and supports a multithreading execution mechanism. GPU architectures employ two important execution paradigms, which we explain below.

SIMD/SIMT GPU processors supply high floating-point (FP) execution bandwidth, which is the driving force for designing graphics applications. To make efficient use of the high number of FP units, GPU architectures employ a *SIMD* or *SIMT* execution paradigm (we explain SIMT and contrast with SIMD in Chapter 1.3.4.) In SIMD, one instruction operates on multiple data (i.e., only one instruction is fetched, decoded, and scheduled but on multiple data operands). Depending on the word width, anywhere from 32, 64, or 128 FP operations may be performed by a single instruction on current systems. This technique

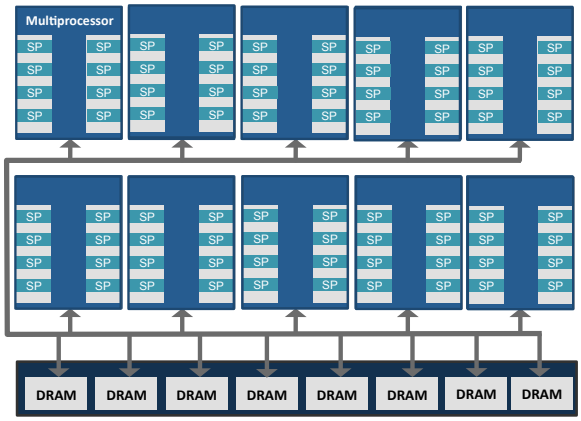


Figure 1.2: Block diagram of NVIDIA's G80 graphics processor (the components required for graphics processing are not shown.)

significantly increases system throughput and also improves its energy-efficiency. This SIMD-style execution is essentially the same technique used in early vector processors. To support SIMD/vector operations, the register file should provide high bandwidth of read and write operations.

Multithreading The other important execution paradigm that GPU architectures employ is *hardware multithreading*. As in more conventional highly multithreaded architectures, such as HEP [86], M-Machine [35], Tera MTA [87], GPU processors use fast hardware-based context switching to tolerate long memory and operation latencies.

The effectiveness of multithreading depends on whether an application can provide a high number of concurrent threads, and most graphics applications have the characteristics since they typically need to process many objects (e.g., pixels, vertices, polygons) simultaneously. Multithreading is the key to understanding the performance behavior of GPGPU applications. In conventional CPU systems, thread context switching is relatively much more expensive: all program states, such as PC (program counter), architectural registers, and stack information, need to be stored by the operating system in memory. However, in

GPUs, the cost of thread switching is much lower due to native hardware support of such process. Although modern CPUs also implement hardware multithreading (e.g., Intel's *hyperthreading*), thus far the *degree* of multithreading (the number of simultaneous hardware thread contexts) is much lower in CPUs than in GPUs (e.g., two hyperthreads vs. hundreds of GPU threads).

To support multithreading in hardware, a processor must maintain a large number of registers, PC registers, and memory operation buffers. Having a large register file is especially critical. For example, the G80 architecture has a 16 KB first-level software managed cache (shared memory) while having a 32 KB register file. This large register file reduces the cost of context switch between threads. This fact is a key distinction from conventional CPU architectures. We discuss these and other architecture features in more detail later in this chapter.

1.3 A GPGPU Programming Model: CUDA

This section briefly discusses the CUDA programming model. The reader should refer to the latest official CUDA programming Guide for more details [73]. The following explanation is adopted from an article by Hwu and Kirk [31].

1.3.1 Kernels

NVIDIA's CUDA (Compute Unified Device Architecture) programming interface is a C/C++ API for GPGPU programming. CUDA programs contain code that runs on a host (typically, the primary CPU processor) and kernels that run on devices (the GPU co-processor).

CUDA source code has a mixture of both *host code* that runs on the CPUs and *device code* that runs on the GPUs. The device code is also typically called as a kernel function. The host code is compiled using the standard compiler; the device code is first converted to an intermediate device language called PTX, which enables the first round of code optimizations. Later this PTX representation is translated into a device-specific binary code, which

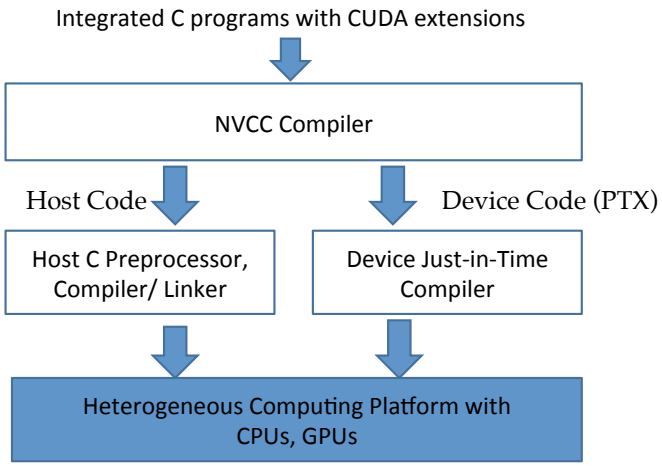


Figure 1.3: Overview of the compilation process of a CUDA program: courtesy of Kirk&Hwu's book [31]

encapsulates optimizations targeting a specific GPU. Figure 1.3 shows an overview of the compilation process of a CUDA program.

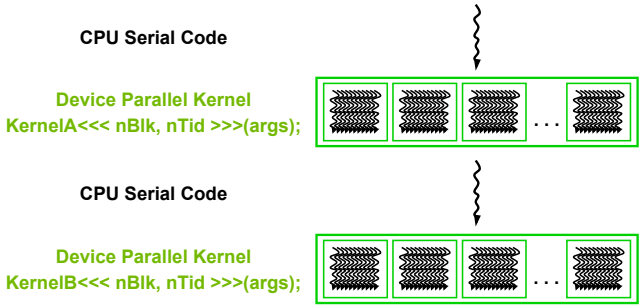


Figure 1.4: Execution of a CUDA program:courtesy of [72]

Figure 1.4 illustrates the execution of a CUDA program, which typically starts with host (CPU) execution. When a kernel function is called or *launched*, it is executed by a large number of threads on a device. A kernel function is called in the host code with the extended function call syntax. e.g. `funcKernel << dimGrid, dimBlock >>(args) funcKernel` will be executed on GPUs. All the threads that are generated by a kernel launch are collectively called a *grid*. Figure 1.4 shows the execution of two grids of threads. After all threads in a kernel are completed, the corresponding grid terminates. The host resumes the CPU code. When the host code encounters another kernel, it invokes the GPU com-

putation again. This example shows a simplified model where the CPU execution and the GPU execution do not overlap. Many heterogeneous computer applications, including newer CUDA programming models, can overlap computations between CPUs and GPUs and even overlap GPU kernel executions.

1.3.2 Thread Hierarchy in CUDA

The CUDA programming model uses a three-level execution hierarchy consisting of *threads*, *blocks* (or *thread blocks*), and *grids*. The smallest work unit is a thread. Threads may be grouped into blocks (thread blocks). A block is also called as cooperative thread array (CTA). When a kernel executes, it does so on a grid, which is one or more thread blocks. This execution hierarchy is strongly coupled with (1) memory space, (2) synchronization, and (3) dispatch and retirement granularity. In real hardware, threads are executed as a lock step forming another execution unit, which is called *warp*.(Warps in NVIDIA terminology and wavefront in AMD terminology.) The concept of warp is discussed in Chapter 1.4.1.

When the kernel, `funcKernel << dimGrid, dimBlock >>(args) funcKernel` is invoked, there is a total `dimGrid` number of blocks and each block has a `dimBlock` number of threads. Hence, in total a `dimGrid` $\times$ `dimBlock` number of threads executes the same kernel.

Execution of threads/blocks

All threads in a block run concurrently. Here, concurrent execution does not mean all threads are executed every single cycle. Rather, it means that all threads in a block have an active state[1] and are executed in a time-multiplexed fashion.

Hence, a thread block is the minimum dispatch and retirement unit. All threads in a block are dispatched together and when all threads in the block are completed, the block can retire. Currently, once a block is assigned a streaming multiprocessor(SM), it cannot be mi-

[1]There are some exceptional cases in which some threads might have terminated earlier than others. But mostly all threads are alive during the execution time of a block.

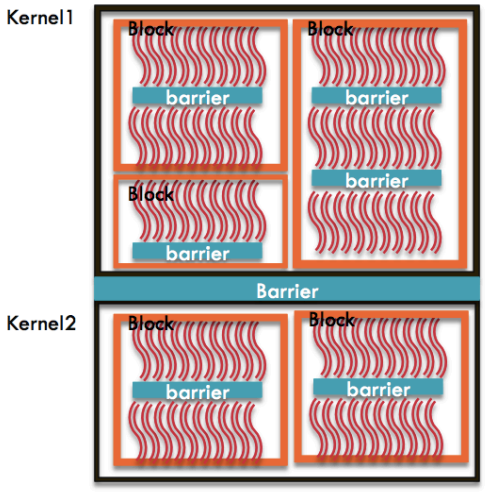


Figure 1.5: CUDA: Bulk synchronization programming model:courtesy of [72].

grated to another SM. All threads in a thread block execute asynchronously until they reach a barrier. Hence, the CUDA programming model is an instance of the bulk-synchronous parallel programming model [33, 91]. In this model, as shown in Figure 1.5, programmers use barriers to synchronize all the threads. However, there is no mechanism to synchronize across all the blocks, since not all blocks are executed in the hardware at the same time. A block can begin execution only when there are enough resources, such as registers, for executing a block. In a given time, the number of running blocks is dependent on the resource requirement from a program and the available hardware resource. When all blocks are finished, a kernel can finish. Hence, there is an implicit barrier between kernels.

The CUDA programming model does not specify the execution order among blocks. In a given time, any number of blocks may be executing on the hardware. This flexibility enables scalable implementation of the hardware as shown in Figure 1.6. Depending on the machine's resources, different number of blocks are executed at the same time. In a low-cost system as in Figure 1.6 (left), only two blocks are executed together; on a current high-end system, up to four blocks may be executing. Currently blocks are scheduled by drivers. There is no specific mapping between blocks and cores.

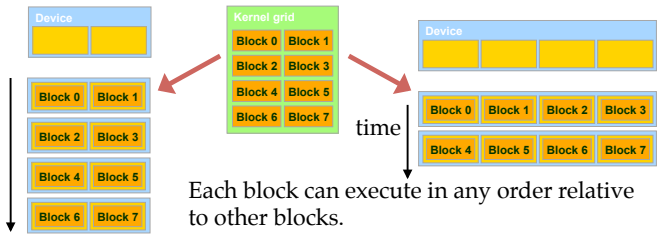


Figure 1.6: CUDA Block execution patterns:courtesy of [72].

1.3.3 Memory Hierarchy

The execution hierarchy is also associated with a memory hierarchy. First, each thread has its own private registers. For memory, a thread has its own memory space, which is called the local memory. A block also has its own memory space, which is called the shared memory. All threads in a block can access the same shared memory, while threads in other blocks cannot. The entire kernel also has its own memory space, which is called the global memory. This memory is so-named because any thread in any block may access the global memory space. Table 1.1 summarizes the various memory spaces and their relation to the execution hierarchy.

There are also constant memory and texture memory spaces, which are similar to global memory in the sense that all threads and blocks may access them. However, these memories are specialized for graphics applications. The texture memory is used to store texture data (2D, 3D data) and the constant memory is used to store a very small number of constant variables. Both texture memory and constant memory are read only from the device side. Only the host processor can write data in these two memory spaces.

Table 1.4 in Section 1.4 shows typical access times to the various memory spaces in GPU architectures.

Table 1.1 also compares traditional parallel programming paradigms. Accessing the shared memory space is very similar to that in the distributed memory (MPI) programming model. Between shared memory regions owned by different blocks, the CUDA program explicitly copies data through the global memory space. However, in the global memory space, all

Table 1.1: Memory Space Comparisons.

	space	$\simeq$ CPU	Programming models in CP
Local memory	within threads	stack	private memory space
Shared memory	within blocks	distributed memory space	distributed memory programı
Global Memory	all	centralized storage	OpenMP programming
Constant Memory	all	centralized read-only storage (very small)	
Texture Memory	all	centralized read-only storage (medium size, 2D access)	

threads can access the space so, the CUDA program is in this way similar to the OpenMP programming model.

1.3.4 SIMT Execution

At a high level, the GPU programming model is based on the Single Program Multiple Data (SPMD) model; a kernel function defines the program that is executed by each of the thousands of fine-grained threads that compose a GPU application. Figure 1.11 shows that all threads execute the same kernel but all access different memory locations.

Since the programming model is SPMD, most of the threads perform the same work. Hence, a group of threads are executed in a lock-step fashion, executing the same instruction (on different data). This microarchitectural grouping of threads, which can affect both control flow and memory access efficiency, introduces the concept of *warp* – a group of threads that are executed together in lock step.

The execution model of G80 is called *SIMT* (single instruction multiple threads). SIMT is very similar to SIMD with slight differences. When programmers write code, they can treat each thread separately. The program model allows each individual thread to perform different work. In contrast, in SIMD, the vector width is determined by the ISA level and one single instruction must perform the fixed vector width data at the same time. When within a warp, if some threads take different paths than the rest of the threads, those branches are called *divergent branches*. An example code is as follows.

```
if (threadId.x>0)  {
    // do work
}
```

(a) if statement

```
if (threadId.x%2) { //divergent branch
   //do work 1
   }
else {
   // do work 2
}
```

(b) if-else statement

Figure 1.7: Examples of divergent branches

1.3.5 CUDA language extensions

CUDA extends the C function declaration syntax. The use of these keywords is summarized in Table 1.2. Using one of `__global__`, `__device__`, or `__host__`, a CUDA programmer can instruct the compiler to generate a kernel function, a device function or a host function. A host function is simply a traditional C function that executes on host and can only be called from another host function. By default, all functions in a CUDA program are host functions if they do not have any of the CUDA keywords in their declaration.

Table 1.2: CUDA C keywords for function declaration.

	Executed on the	Only callable from the
__device__ float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ float HostFunc()	host	host

Another important extension to C/C++ is the use of particular built-in variables for identifying threads: `threadIdx`, `blockIdx`, and `blockDim`. These variables are associated with pre-defined hardware registers that provide the identifying thread "coordinates." In particular, different threads will see different values of these variables; moreover, the values of these variables may be used to construct a globally unique integer thread ID for a given thread. For instance, this facility lets a programmer prescribe a particular mapping of threads to data, as we illustrate below.

```
// compute vector sum C=A+B
void vecAdd(float *A, float *B, float *C, int n)
{
  for (int ii = 0; ii < n; ii++) C[ii] = A[ii] +B[ii];
}

int main()
{
    // memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N element each
    ...
    vecAdd(h_A, h_B, h_C, N);
}
```

Figure 1.8: A simplified traditional vector addition C code example.

```
# include <cuda.h>
...
void vecAdd(float *A, float *B, float *C, int n)
{
   // memory allocation for h_A, h_B, and h_C
   // I/O to read h_A and h_B, N element each

   ...
   // 1. Allocate device memory for A, B, and C
   // Copy A and B to device memory

   // 2. Kernel launch code
   // the device performs the actual vector addition

   // 3. Copy C from the device memory
   // Free device memory
}
```

Figure 1.9: vecAdd function on the CPU side that calls a GPU kernel.

1.3.6 Vector Addition Example

We show an example of a kernel invocation using a vector addition. First, we show a traditional C program for a vector addition ($\vec{C} = \vec{A} + \vec{B}$) in Figure 1.8. The actual vector addition operation is performed by a `for` loop. Now, the same code is changed to perform vector addition in the device (GPU). Figure 1.9 shows only the *host* side of the code that will launch the GPU code. Figure 1.10 shows the *device* (GPU) side of the code.

```
// Each thread performs one pair-wise addition
__global__
void vecAddKernel (float *A, float *B, float *C, int n)
{
  int idx = threadIdx.x + blockDim.x*blockIdx.x;
  c[idx] = A[idx] + B[idx];
}
```

Figure 1.10: A vector addition kernel function on the GPU side.

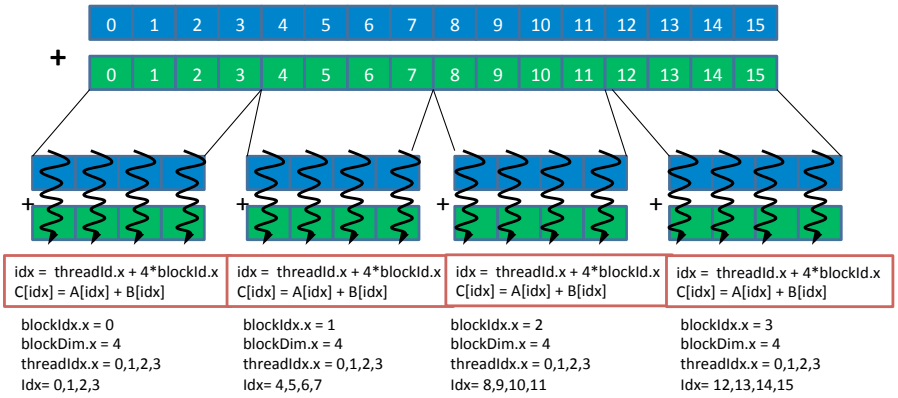


Figure 1.11: All threads in a grid execute the same kernel code but access different memory locations. (Here, `blockDim.x` has the value of 4.)

Device and Global Memory

In CUDA, the host and device have separate memory spaces.[2] The actual hardware also has separate physical memories as shown in Figure 1.1: CPUs have their own DRAM and GPUs also have their own DRAM. The GPU's DRAM memory space is called *global memory* or *device memory*. In order to execute a kernel on a device, the programmer needs to allocate global memory on the device and transfer pertinent data from the host memory to the allocated device memory. This corresponds to Part 1 in Figure 1.9. Similarly, after a kernel execution, the programmer needs to transfer result data from the device memory back to the host memory and free up the device memory that is no longer needed (Part 3 in Figure 1.9). The CUDA runtime system provides API functions for managing data in the device memory. For example, Part 1 and Part 3 of the vecAdd() function in Figure 1.9 need to use these API functions to allocate device memory, transfer data from device/host

[2] In CUDA 4.0, a unified memory space is available but the implementation of this facility is not yet complete.

to host/device, and free the device memory.

Memory Data Indexing

In the SPMD programming model, a single program handles multiple data (SIMD or SIMT). Each thread operates different data. It either accesses its own register files or different memory locations. When accessing memory, a programmer or algorithm designer typically expects a particular assignment of threads to data. To specify such mappings, CUDA provides built-in variables for identifying threads, as discussed above. Figure 1.11 illustrates this concept for vector addition. Here, suppose we wish to add two vectors of length 16, and we wish to assign 1 thread to perform each addition. Further, suppose we create a pool of 16 threads, organized into 4 thread blocks with 4 threads each. Each thread can discover its unique thread block ID using the built-in variable `blockIdx.x`, and its unique thread ID within its block using the variable `threadIdx.x`, as shown in Figure 1.11. From these, we can easily translate the loop iteration variable in the sequential code into an index for use in the threaded code. Also note the `.x` field notation; as it happens, there are also `.y` and `.z` fields, which may be used when the most natural way to organize threads is two- or three-dimensional, as commonly occurs in imaging and graphics applications.

1.3.7 PTX

PTX is the virtual ISA used by NVIDIA GPU architectures. A compiler converts PTX code into the native ISA for a given GPU architecture. Register allocation and specific architecture-based optimizations are performed during the code generation generation from PTX to the native binaries.

1.3.8 Consistency Model and Special Memory Operations

The CUDA programming model does not specify a consistency model. Since the execution model is bulk synchronous, the memory operations across threads can be reordered. One

may regard the CUDA memory model as being one of *weak consistency*. To enforce a memory ordering, recent versions of the CUDA programming model provide built-in functions for fences, e.g., _threadfence() and mem_fence(). These functions guarantee that all the previous memory requests prior to these functions are visible to all threads. In other words, _threadfence() is used to halt the current thread until all previous writes to shared and global memory are visible by other threads.[3] The CUDA programming model also provides several atomic operations that are useful for implementing lock operations. Examples of atomic functions are (atomic add/sub/exchange/inc/dec/min/max).

Often atomic operations and fence() functions are used together. For example,

1. store data

2. _threadfence()

3. atomically mark a flag

These steps guarantee that if other block sees the flag, it will also see the data.[4]

1.3.9 IEEE floating-point support

Earlier GPU designs did not follow the IEEE floating-point standard, as these were not deemed as being necessary for graphics applications. However, with the rise of GPGPU programming, the floating-point implementations have increaesd in compliance with the IEEE-754 standard, including single-precision support in G80 architectures as well as double-precision support in the Fermi architecture.

1.3.10 Execution Model of OpenCL

Although our discussion thus far has focused on CUDA, a consortium has assembled an alternative open-standard called OpenCL (Open Computing Language). OpenCL adopts CUDA-like constructions but promises portability across a range of platforms, including

[3] (http://stackoverflow.com/questions/11570789/cuda-threadfence)
[4] (http://stackoverflow.com/questions/5232689/cuda-threadfence/5233737#5233737)

NVIDIA GPUs, AMD GPUs, and multicore CPUs from several vendors. Applications written in OpenCL are compiled for the specific target architecture at runtime. The goal of the OpenCL consortium is to develop an industry-wide standard parallel programming environment and enable running a single application code on different types of devices and/or creating kernels from a single application and dispatching them to the available OpenCL-compatible computing devices at runtime. While CUDA is mainly built based on a fine-grained SPMD (Single Program Multiple Data) execution model with limited inter-thread communication OpenCL also supports the task-parallel programming model. The OpenCL API provides data structures and routines to synchronize execution and share data among kernels running on different devices.

As with CUDA, an application or a *program* partly consists of a number of functions or *kernels*, which are executed on OpenCL *devices*. The host code assigns kernels to the available computing devices through a *command queue*. To do so, the programmer needs to set up an OpenCL *context* on the host side to handle memory allocation, data transfer between memory objects and command queue creation for devices.

Table 1.3: CUDA vs. OpenCL.

	OpenCL	CUDA
Execution Model	Work-groups/work-items	Block/Thread
Memory Model	Global/constant/local/private	Global/constant/shared/local + Texture
Memory consistency	weak consistency	weak consistency
synchronization	using a work-group barrier (between work-items)	calling `__sync_threads()` between threads

1.4 GPU Architecture

This section sketches the design of current GPU architectures. Much of this discussion pertains to NVIDIA's Tesla/Fermi architectures. However, our descriptions are not intended to correspond directly to any existing industrial product.

1.4.1 GPU Pipeline

Figure 1.12 shows an overview of a GPU architecture pipeline. It shows one streaming multiprocessor, which is based on an in-order scheduler. Similar to traditional architectures, it has fetch, decode, scheduler, register read, execution and write-back stages.

Fetch and Decode Stage

The front-end is very similar to traditional multithread architectures. Multiple PC registers exist to support multiple warps. The scheduler selects a warp to fetch based on scheduling algorithms, such as round-robin or greedy-fetch. The round-robin policy selects a warp from the list of ready warps, which gives an equal priority to each warp. In the greedy-fetch policy, the streaming multiprocessor fetches instructions from one warp until a certain event occurs such as I-cache miss or fetching a branch instruction or an instruction buffer full. Since the front-end has multiple warps to fetch, when it encounters such events, it simply switches to fetch another warp. For the same reason, branch predictors play a diminished role and are not typically implemented. Newer GPUs execute multiple warps at one cycle, so the front-end could fetch instructions from different warps at the same cycle instead of one warp at one cycle.

After an instruction is fetched, it is decoded in the decode stage. The streaming multiprocessor can have an instruction buffer for each warp or share a buffer for all warps.

Scheduler and Score Boarding

The GPU processor has an in-order scheduler. In the G80 architecture, it executes only one warp at a time; later architectures like Fermi schedule multiple warps. The scheduler uses a scoreboard to find a ready warp. So far, no GPU architectures have employed out-of-order schedulers. However, the scheduler can select any warps that are ready. Hence, from a programmer's view point, a program might look like an out-of-order execution. For example the scenario in Figure 1.13 is possible.

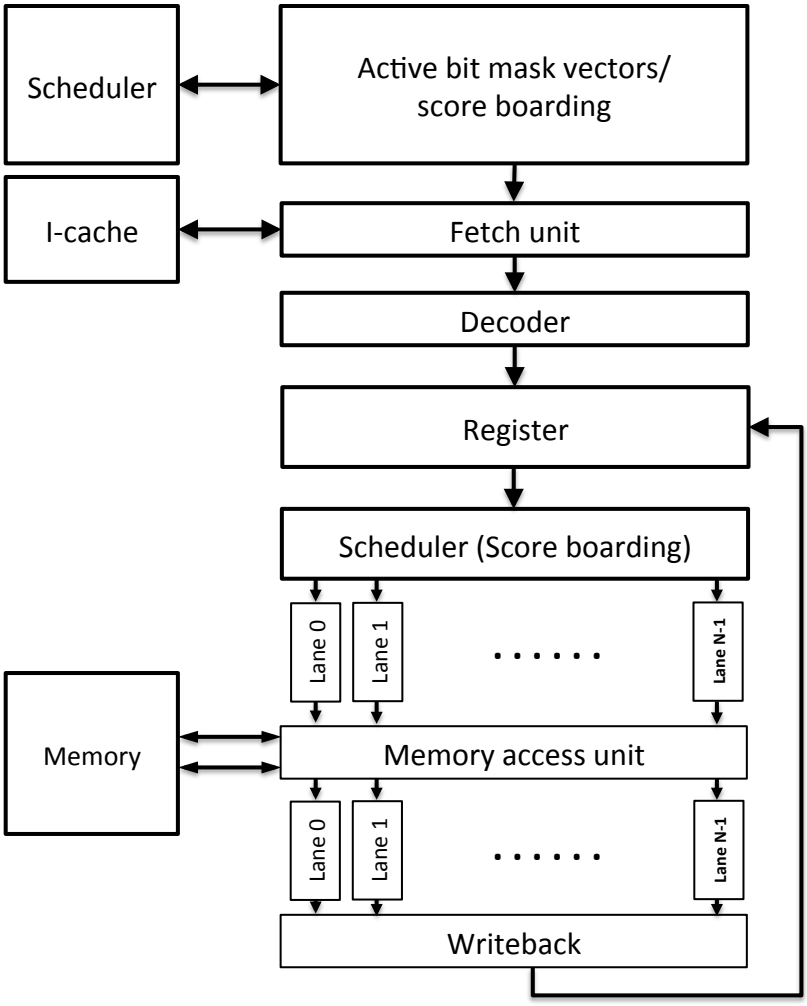


Figure 1.12: An overview of GPU streaming multiprocessor pipeline

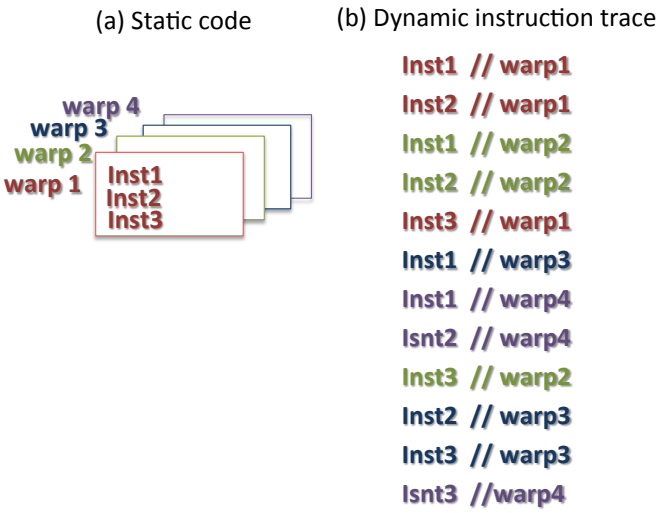


Figure 1.13: An example of static and dynamic instruction traces

Scoreboarding Scoreboarding is one method to implement dynamic scheduling. It was first introduced in CDC6600 [89]. It checks read-after-write and write-after-write data dependencies. Scoreboarding does not provide the register renaming mechanism, but instructions can execute out of order (i.e., instructions should be scheduled in-order but can be finished (complete functional units/memory accesses) out of order) when there are no conflicts and the hardware is available. In GPUs, scoreboarding is used to check any RAW or WAW dependency, so instructions from the same warp can be executed even if earlier instructions have not finished yet. This approach increases instruction/memory level parallelism.

Register read/write

To accommodate a relatively large number of active threads, the GPU processor maintains a large number of register files. For example, if a processor supports 128 threads and each thread uses 64 registers, in total 128×64 registers are needed. As a result, the G80 has a 64 KB register file and the Fermi supports 128 KB of register file storage per streaming multiprocessors (in total, a 2MB register file). The register file should have a high capacity and also high bandwidth. If a GPU has 1 Tflop/s peak performance and each FP operation needs at least two register reads and one register write, 2 T*32 B/s=64 TB/s register

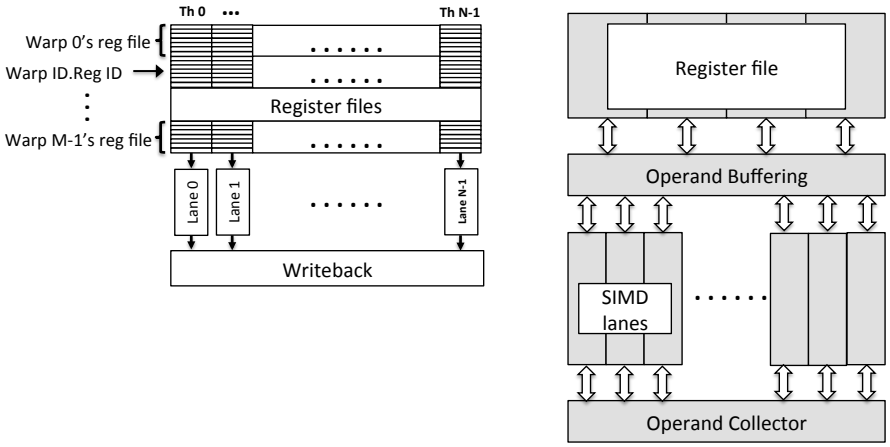


Figure 1.14: GPU Register File Accesses. Left: Using multiple banks (figure courtesy of [70]), Right: Using operand buffering (figure courtesy of [41])

read bandwidth is required. Providing such high bandwidth is particularly challenging, so several techniques have been used, including multiple banks and operand buffer/collectors.

Multiple banks The streaming multiprocessor provides high bandwidth by subdividing the register file into multiple banks. Figure 1.14 shows the register file structure. All threads in the warp read the register value in parallel from the register file indexed by both warp ID and register ID [70]. Then, these register values are directly fed into the SIMD backend of the pipeline.(Please remember that, each SIMD unit/lane is used by only one thread.)

Operand Buffer Gebhart et al. [41] show another example of the register file structure in Figure 1.14. In that design, a buffer exists between the register file and the execution units. Instead of reading all the necessary register values right before the values are needed, which gives a very high pressure to the register file read/write ports, the processor can buffer the register values.

The buffer can store register values that are read through multiple cycles, thereby reducing the register read bandwidth requirement. In their design, four SIMT lanes form a cluster. Each entry in the streaming multiprocessor's main register file is 128 bits wide, with 32

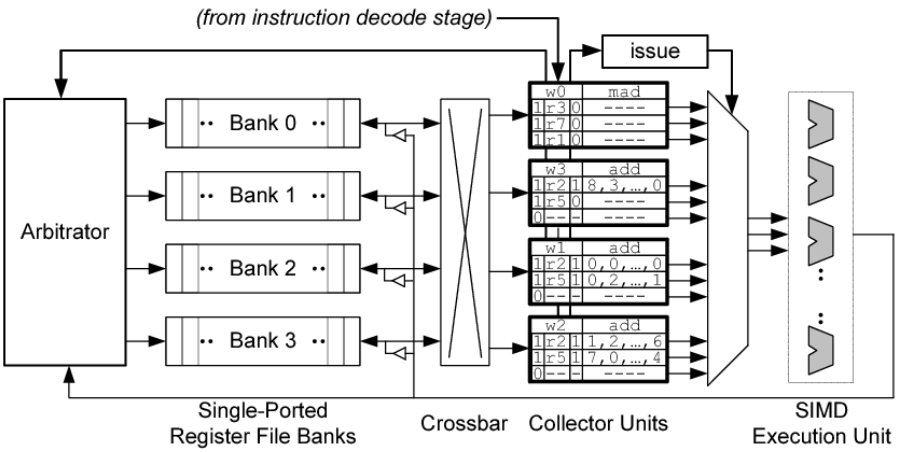


Figure 1.15: Detailed diagram of Operand Collector [13]

bits allocated to the same-named register for threads in each of the four SIMT lanes in the cluster. Several documents [13, 57, 63] indicate that they employ operand collector buffers. These operand collectors are also used to aggregate result values. The operand collector works as a result queue, which buffers the output from functional units before being written back into the register file. Although the main benefit of the result queues is to increase the effective write through-puts, it also provides further optimization opportunities. When the outcomes of instructions are used only in the instructions dynamically scheduled close enough, the output values can be forwarded to the input of the next operation. This behaves just like a CPU's forwarding network [57].

When an instruction requires multiple accesses to the same bank, the register values are read over multiple cycles. Since the register addresses are determined statically, the compiler can in principle reduce or remove bank conflicts.

Register File Cache To reduce the pressure on the register file, Gebhart et al. propose a register file cache [41]. Although a register file cache was originally proposed to reduce the register file access time [29], in GPUs the register file cache is proposed to reduce register reads and writes. Current GPU architectures do not support precise exceptions, so register files do not have to maintain architectural states. Therefore, many write opera-

tions can be easily merged inside the register file cache, which may reduce register writes significantly [70].

Execution stage

In the execution stage, an instruction accesses either the memory unit or functional units. The memory system is discussed in Section 1.4.3.

The execution stage consists of vector processing units (AMD GPUs have a scalar unit as well). One vector lane executes one thread, which is called a stream processor in NVIDIA's terminology. The simplest design would be to have the number of lanes and the warp size be the same. However, this approach could also require a relatively large amount of static power consumption and a large area. Instead, GPUs execute the same instruction over multiple cycles. For example, the G80 has only eight vector lanes, and therefore the streaming multiprocessor takes four cycles to execute 32 threads (i.e., one warp). The computed results are temporarily stored and then written back to the register file together.

Back-to-back operation The execution width of GPU is wide (32 threads), which makes it much harder to have a data forwarding path. For example, 32 $\times$ 32 B (total 1 KB) or 32 $\times$ 64 B (total 2 KB) widths are quite large. Hence, the results are either written directly to the register file or temporarily stored in a small buffer and then written back to the register file over multiple cycles. In either case, the results are not available to dependent instructions immediately after execution. This scenario is quite different from many modern CPUs, where data forwarding is used widely. When there is a forwarding path, register read/write cycles are not in the critical path and any back-to-back operations are executed immediately following the execution latency. However, GPUs can avoid this penalty by utilizing thread-level parallelism(TLP). The pipeline simply schedules instructions from other warps, so it can hide the latency. When there is enough TLP, the execution and register write latency can be hidden.

Volkov discussed this issue in his GTC'10 talk [95]. When there are four threads, the

streaming multiprocessor can simply switch to other threads so the execution latency can be hidden. However, when there is only one thread, these back-to-back operations take much longer. This performance issue is also discussed in Chapter 4.1.

Special function units To date, NVIDIA architectures have adopted SIMD execution units and AMD architectures have used VLIW architectures. In addition, special function units provide VLIW-like effects in NVIDIA GPU architectures. Graphics applications often require transcendental functions for algorithms such as geometric rotations and scaling. It is often possible to use implementations of the corresponding transcendental functions that do not have ultra-high accuracy. Many GPU architectures therefore provide fast, approximate implementations in hardware-based special function units (SFUs). Since many programs do not need these SFUs all the time, the processor has only two-four SFUs in addition to its regular FP units. The scheduler can issue SFU instructions and regular FP instructions together if they are independent, e.g., when the instructions come from different warps. In such cases, these SFU units provide additional execution bandwidth. Section 4.1 discusses this benefit in a more detail in real computation.

1.4.2 Handling Branch Instructions

Branches are particularly challenging to implement well in GPUs. Recall that current GPU hardware implementations fetch just one instruction for each warp. When this instruction is a divergent branch, threads within a warp will need to execute different instruction paths. This challenge is common to GPUs and earlier vector processors.

The solution in vector processors is to use vector lane masking or predication [17, 81]. However, the problem in GPUs is slightly different. Not only does part of the SIMD unit need to be predicated, but the processor also needs to fetch instructions from different paths. Indeed, the existence of divergent branch is the major difference between SIMD and SIMT execution. Fung et al. first described this problem in GPGPUs [38].

Figure 1.16 illustrates the problem. Suppose that the warp size is 4 and that two threads take

```
if (thread.Idx < 2) {
  // do work 1
}
else {
   // do work 2
}
```

Figure 1.16: Divergent branch example

the taken path while the remaining two threads go to the fall-through path. The streaming multiprocessor has to fetch both paths one by one. An active mask indicates which threads participate in each path. The active mask bit information is also used in any register read or write.

The most naïve solution is SIMD serialization. When divergence occurs at a given PC, the processor serializes the threads within a warp. This will cause performance degradation.

A stack-based reconvergence solution is proposed to handle divergent branches [38, 39, 62, 100]. In their scheme, divergent threads execute *without* lock-step until the end of the program; the processor detects the point at which the threads can rejoin in a lock-step again, i.e., when they may form a warp again. The control-flow reconvergence point is the point where all threads merge. When a program diverges, the processor inserts the reconvergence point into the stack. When threads in one path reach the reconvergence point, the processor fetches instructions from other remaining paths. Once all the divergent paths are fetched, in other words once all threads reach their reconvergence points, the divergent threads merge.

In the stack-based reconvergence system, the streaming multiprocessor uses a hardware stack structure to record the join location and the next fetch address. When the streaming multiprocessor fetches a divergent branch, it first stores the PC address of the reconvergence point (control-flow merge point) into the stack. Then, the streaming multiprocessor fetches one path first. Since there is only one PC for each warp, the streaming multiprocessor must store the other path PC address in the stack as well. The compiler identifies the control-flow merge point and includes that information. Each stack entry has three fields: the PC address of the reconvergence point, active mask bits, and the next PC address. Figure 1.17 illustrates the process [38, 70]. Figure 1.17(a) shows a control flow graph. Figure 1.17(b)

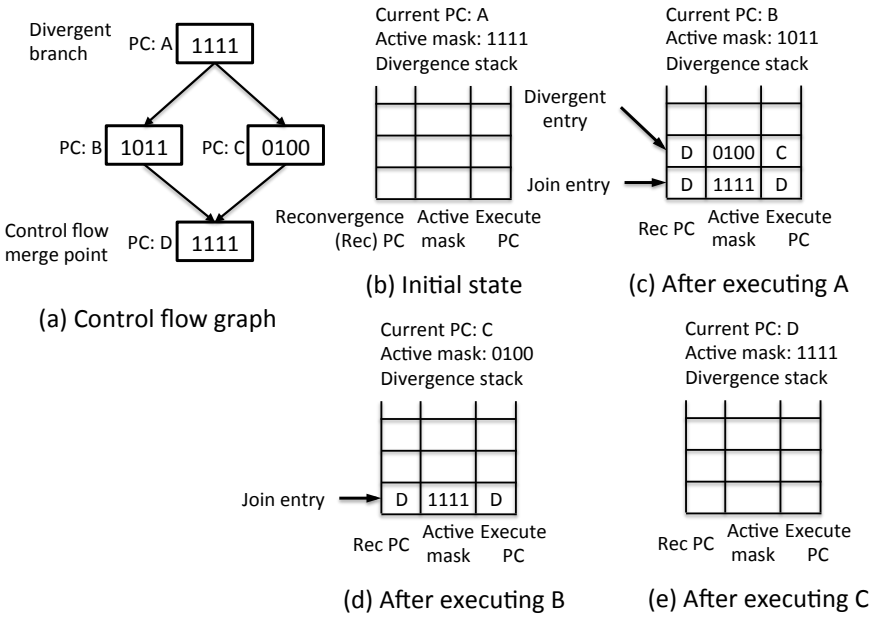


Figure 1.17: A stack-based divergent branch handling mechanism (figure courtesy of [70])

shows an initial state of the stack. When a processor fetches a divergent branch, it pushes a join entry in the stack. Then, the processor chooses one of the path and it pushes the other path information in the stack. The active mask field is set to the current active mask. (Figure 1.17 (c) shows the stack after executing branch A.) The top of the stack shows the join entry and active mask of the path. When a processor fetches an instruction, it always checks whether the next PC address matches with the reconvergence PC. After fetching PC B, the next PC address is D, which is the same value in the top of the stack. Hence, the processor pops the top entry from the stack and starts to fetch from PC C and uses the active masks in the entry. (Figure 1.17 (d) shows the state of the stack after executing PC B.) Since the next PC address is again D, the top entry in the stack is popped. The active mask is all 1s, so now all threads are participating the path. (Figure 1.17 (e) shows the state after executing PC C).

Solutions to reduce the impact of divergent branches in hardware Several solutions are proposed to handle the divergent branch problem. To utilize idle threads in warps, Fung et al. pro-

posed a dynamic warp formation [38, 39]. At run-time, the hardware forms a new warp by combining threads from the same path. Meng et al. proposed dynamic warp subdivision [69]. Dynamic warp subdivision handles not only divergent branches,but it also attacks divergent memory operations. In divergent memory operations, some threads hit the cache but some do not so threads generate different memory latencies within a warp. Several other solutions such as dynamically changing the execution units such as thread compaction [40], large warp formation [70], compaction based on the benefit prediction [79] [79], interweaving threads [20, 32] at dynamic time are also proposed.

Solutions to reduce the impact of divergent branches in software Several compiler-based solutions also exist. Zhang et al. reduce divergent branches by remapping data locations[105] Han and Adelrahman proposed compiler solutions [46]. Diamos et al. rearrange threads based on the frequency at static time [32]. Their solutions can also reduce the divergent branches.

1.4.3 GPU Memory Systems

The GPU memory system has several levels of hierarchy. Because there can be a large number of memory requests on-the-fly, the system requires several large buffers and/or queues.

A streaming multiprocessor has a cache to supply fast data accesses. In earlier GPGPU architecture like the G80, there were only software-managed first-level caches. More recent GPGPU architectures adopt a more aggressive cache hierarchy. In Fermi, the first-level cache can be controlled only by software (program) or in a hybrid software-hardware managed fashion. If a program controls a software managed cache, the instruction explicitly brings data to the cache and evicts it. The shared memory space in CUDA is explicitly controlled by software. Since memory addresses are determined at run-time, unlike register files, there could be bank conflicts in the cache.

Table 1.4 shows different memory spaces and typical memory latencies. So far, computer

Memory Space	Tesla	Fermi	access latency
Local memory	DRAM	DRAM and hardware cache	100s cycles
Shared memory	software cache	software cache	4 32 cycles
Global memory	DRAM	DRAM and hardware cache	100s cycles
Constant memory	DRAM and constant cache	DRAM and constant cache	100s cycles or 4 cycles (cache hit)
Texture memory	DRAM and texture cache	DRAM and texture cache	100s cycles or 4 cycles (cache hit)

Table 1.4: Memory space and hardware access time [82, 98]

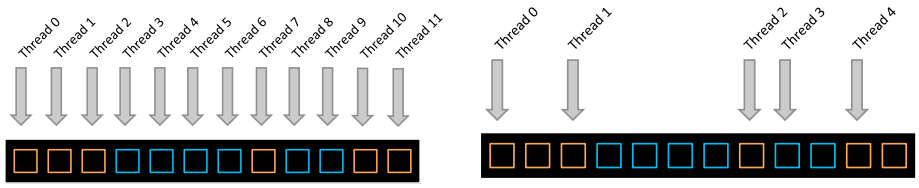


Figure 1.18: Coalesced Uncoalesced memory requests. Left: All threads are accessing sequential memory addresses (coalesced), Right: threads are accessing non-sequential memory addresses (uncoalesced): courtesy of [72]

architects have presented two major problems in the memory system. One is managing the large size of buffers, and the other is dealing with long memory latency.

DRAM system GPU uses GDDR which is specialized for high-bandwidth DRAMs. It increases the bandwidth of the memory system but dram latency is a bit slow. GDDR has a high number of DRAM banks to increase the bandwidth even further. Typically GDDRs provide higher DRAM bandwidth than DDR, because of the wider burst length and DRAMs are directly connected to processors. In CPUs, DRAMs are connected through DIMMS which slows down the communication bandwidth

DRAM scheduling algorithms are tuned to provide high bandwidth. FRFCFS is the best way to increase through-put oriented computing. Since GPGPU applications have high spatial locality, FRFCFS shows much higher performance than FCFS [103].

Multiple memory transactions The basis of SIMD execution unit is executing multiple data at once. If the data is in the register, all the register files can be accessed together. The challenges result when a warp executes memory operations. The memory addresses can be anywhere. In earlier GPUs, this was one of the most important performance optimizations [82, 83, 96]. When all threads within a warp accesses sequential memory addresses,

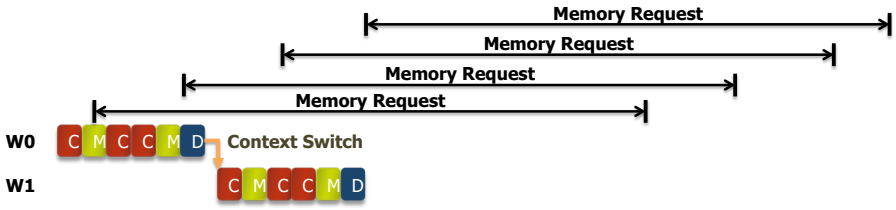


Figure 1.19: Multiple in-flight memory requests (Even though 2 warps are running, 4 memory requests are concurrently serviced)

these memory addresses can be easily represented by a few transactions. However, if memory addresses are all scattered, these memory operations will generate multiple transactions. In earlier GPUs, especially earlier than CUDA computing version 1.2, these were called coalesced and uncoalesced memory addresses as shown in Figure 1.18. Coalesced memory addresses generate one to four memory transactions depending on the transaction sizes and uncoalesced memory addresses generates up to 32 transactions. After some hardware optimizations, the hardware combines as many memory addresses as possible and generate fewer transactions. Coalescing is used to combine as many requests as possible. Still reducing the number of memory transactions is one of the essential performance optimizations. The hardware typically has a special hardware unit to handle this scatter/gather operation.

Multiple in-flight memory requests Naturally, GPU processors can have high MLP which is the same as TLP. To increase the MLP even more, the GPU processors employ stall at dependent instructions. Even though GPUs have an in-order processor, cache misses do not prevent from execution of an instruction from the same thread. The processor can still execute instructions from the same warp until instructions that are dependent on the cache misses stall the warp. Figure 1.19 illustrates this behavior. This gives a small window of exploiting MLP within a warp. In this way, this can increase the number of in-flight memory requests. When a program has high TLP, it does not affect too much. However, when there are fewer threads, exploiting MLP effects can increase the total MLP significantly.

1.5 Other GPU Architectures

1.5.1 The Fermi Architecture

Major successors to the G80 design are based on the Fermi architecture, which was first released in 2010. The Fermi architecture has taken a significant leap forward in GPU architecture design by providing a two-level cache hierarchy to better support the applications that are not able to use the GPU's shared memory efficiently. Among other major improvements are the improved double precision performance, faster atomic operations to reduce the cost of inter-thread-block communication, and the error correction code (ECC) support. A more detailed description of the hardware and software features of the Fermi architecture can be found in [51].

1.5.2 The AMD Architecture

ATI stream technology which is supported by all modern AMD GPUs such as ATI Radeon and FireStream has also provided massively parallel computation. ATI originally had supported Brook$^+$[3] to provide GPGPU features, but instead ATI now supports OpenCL. So far the major difference between NVIDIA's GPU architectures and AMD's GPU architectures is that AMD has a SIMD-VLIW architecture. Four ALUs and one branch or five ALUs and one branch unit are packed together, all of which execute the same instruction. AMD can easily support scalar instructions as one of the VLIW units but NVIDIA so far only has vector units.

Lately, the greatest advantage of AMD architectures has been the fusion [6] architecture, which combines both CPUs and GPUs in the same die. Both CPUs and GPUs are integrated and share the same memory system. AMD's fusion architecture is the first OpenCL computing platform that actually supports a heterogeneous computing system. Sharing the same memory system dramatically reduces the communication cost between CPUs and GPUs. However, so far, only low-end GPUs are integrated in the fusion architecture so the AMD fusion architecture has not yet been widely used for GPGPU computing. Nonethe-

less, we predict that soon AMD's architecture will be widely used as GPGPU computing platforms.

1.5.3 Many Integrated Core Architecture

Intel's Many Integrated Core (MIC) architecture also targets high-throughput computing processors [26]. Unlike AMD and NVIDIA, Intel's MIC architecture executes native x86 ISAs, which is the greatest strength of Intel's platforms. MIC also has a wide SIMD unit (512bits wide) to produce high-throughput computing, but it also has most of the features that are traditionally available in CPU architectures, such as virtual memory, fully coherent cache, and a branch predictor. However, the number of concurrently running threads is much smaller (4) than that of NVIDIA or AMD. Hence, MIC is not only specialized for through-put oriented computing, but also targets latency-limited applications.

1.5.4 Combining CPUs and GPUs on the same Die

Intel's Sandybridge and AMD's Llano put both CPUs and GPUs on the same die. So far, the GPU performance is much lower than discrete GPUs. Power consumption and memory bandwidth could be the main reason to combine powerful GPUs and CPUs on the same die. Combining CPUs and GPUs introduce several resource management problems. The research on resource management such as cache partitioning [61], and DRAM scheduling policy [10, 55]

Chapter 2

Performance Principles

Developing algorithms for GPGPUs is fundamentally about applying the same long-studied principles of parallelization and I/O-efficient design relevant to other shared memory parallel platforms. This chapter reviews these principles, focusing on recent results in both the theory and practice of parallel algorithms, and suggests a connection to GPGPU platforms. Ideally, applying these principles with the right cost models leads not only to provably efficient algorithms, but also offers hints to architects about the features and configurations likely to have the most impact on the performance of a given computation. Thus, we believe this discussion will be useful to practitioners in various aspects of parallel computing, not just those interested specifically in GPGPUs.

2.1 Theory: Algorithm design models overview

To first order, the two characteristics of any algorithm likely to determine its performance are (a) how much parallelism is available, and (b) how much data must move through the memory hierarchy. Thus, when designing an algorithm, we would like an abstract machine model that allows us to assess our algorithm along these dimensions. In such a model, we might want to do the same kind of "big-$\mathcal{O}$" analysis to which we are accustomed in the sequential case. Doing so would allow us to get the high-level algorithm design right before moving on to lower-level performance optimization and tuning. Importantly, the

abstractions should not be so cumbersome that we cannot in a reasonable amount of time design and analyze candidate algorithms.

Although the state of algorithm design models is in flux, we have reasonable options. For GPGPUs and other manycore-style processors, two suitable models are the so-called *work-depth* (or *work-span*) model for analyzing parallelism [15, 28, 54], and the *external memory model* for analyzing I/O behavior in the presence of a memory hierarchy [4, 93, 94]. These two models evolved separately, but recent work has shown ways in which to connect them [16, 30]. The remainder of this chapter reviews aspects of this work.

2.2 Characterizing parallelism: the Work-Depth Model

In the work-depth model, we represent a computation by a directed acyclic graph (DAG) of operations, where edges indicate dependencies, as illustrated in Figure 2.1. Given the DAG, we measure its *work*, $W(n)$, which is the total number of unit-cost operations for an input of size n; and its *depth* or *span*, $D(n)$, which is its critical path length measured again in unit-cost operations. Note that $D(n)$ ought to be a lower bound on the minimum execution time; and the ratio $W(n)/D(n)$ effectively measures the average amount of available parallelism as each critical path node executes. In fact, the ratio $D(n)/W(n)$ is similar to the concept of a sequential fraction, as one might use in evaluating Amdahl's Law [7, 47, 99]. Thus, our implicit goal is to maximize $W(n)/D(n)$, or, alternatively, minimize $D(n)/W(n)$. Importantly, this model makes no explicit reference to the number of processors. In this sense, we may regard the model as being machine-independent. Nevertheless, if we want to know how many processors to throw at an algorithm, $W(n)/D(n)$ is a suitable guide.

However, it is easy to maximize $W(n)/D(n)$ and still get a bad algorithm. For instance, we can artificially inflate the total operations $W(n)$. Thus, we should also try to ensure our algorithm is *work-optimal*, a property which says that $W(n)$ is not asymptotically worse than the best sequential algorithm. Indeed, work-optimality is a critical requirement.

Let's see why work-optimality matters. First, note that it is possible to estimate (crudely)

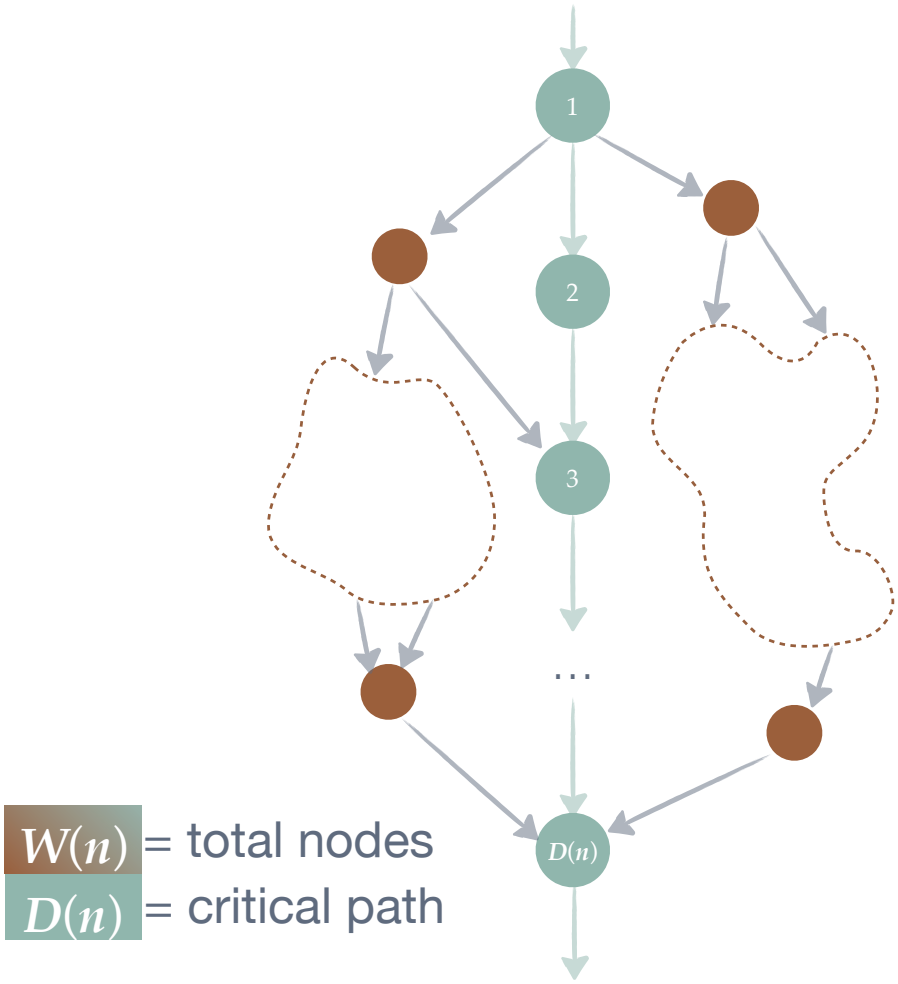
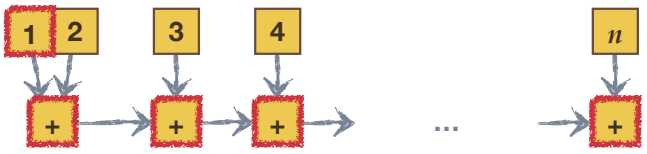


Figure 2.1: A parallel computation in the work-depth (or work-span) model. The computation for an input of size n is a directed acyclic graph with $W(n)$ nodes, each representing a unit-cost operation; edges representing strict dependencies among these operations; and a critical path of length $D(n)$ nodes. Our goal is to design algorithms that achieve *work-optimality* while maximizing the average available parallelism, $W(n)/D(n)$.



$$W(n) = \Theta(n)$$
$$D(n) = \Theta(n)$$

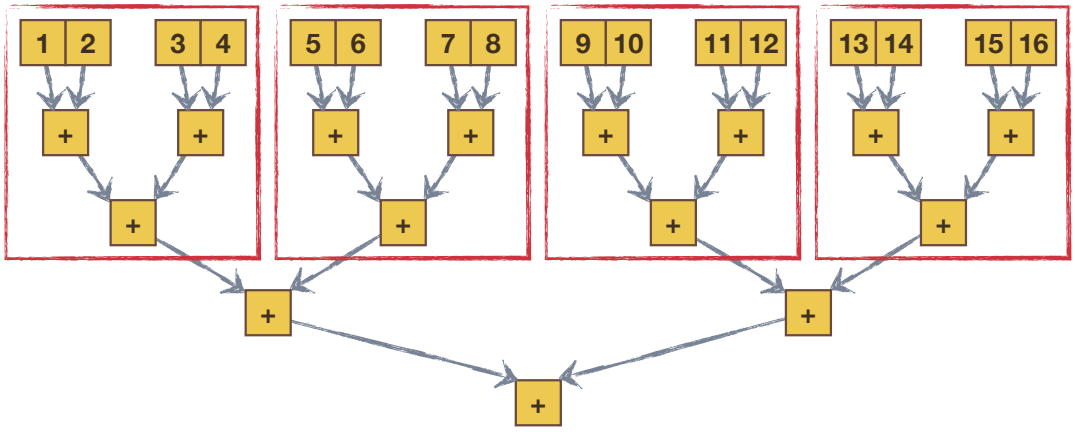
Figure 2.2: Work-depth example: Summing a list of n elements with a sequential algorithm. Both the work and the depth are $\Theta(n)$, meaning the available parallelism $W(n)/D(n)$ is a constant.

the algorithm's running time given p identical processors, using a theorem by Brent [19]. This theorem says that if the nodes of the DAG have unit cost and the machine has p processors, then it is possible to schedule the DAG so that the time to execute (compute) the DAG, $T_{\text{comp}}(n;p)$, is

$$T_{\text{comp}}(n;p) = D(n) + \frac{W(n) - D(n)}{p}. \tag{2.1}$$

Suppose we have designed a highly parallel algorithm, with $W(n) \gg p \cdot D(n)$, and furthermore that our algorithm only exceeds the work $W_0(n)$ of the best sequential algorithm by a factor of $\epsilon(n)$, i.e., $W(n) = W_0(n) \cdot \epsilon(n)$. Then, the speedup of our algorithm on p processors is roughly $W_0(n)/(W(n)/p) = p/\epsilon(n)$. That is, the best possible speedup of p will be reduced by $\epsilon(n)$. Consider the relatively small factor of $\epsilon(n) = \log n$. Even for $n = 1024$, $\log n = 10$, meaning the best possible speedup is an order of magnitude less than we might hope for. Put another way, to even *match* the sequential algorithm, we need $p > \epsilon(n)$. Thus, if we are deciding between an $O(n)$ sequential algorithm and an $O(n^2/p)$ parallel algorithm, we will need $p = n$ just to match the sequential case. These examples underscore the importance of work-optimal algorithms.

Example: Reduction. Suppose we wish to compute the sum of n values. A sequential algorithm would lead to a DAG like the one shown in Figure 2.2. In this case, we perform $\Theta(n)$



$$W(n) = \mathcal{O}(n)$$
$$D(n) = \mathcal{O}(\log n)$$

Figure 2.3: Work-depth example: Summing a list of $n = 16$ elements with a tree algorithm.

operations but the critical path is also of length $\Theta(n)$. Thus, the available parallelism—or ratio of $W(n)/D(n)$—is a constant: no matter how large the input, there is never more than a fixed amount of concurrency.

An algorithm with more parallelism might instead have the DAG shown in Figure 2.3. This algorithm organizes the additions into a tree, where independent subtrees can be performed in parallel. This tree still performs $W(n) = \Theta(n)$ total operations, and is thus work-optimal. However, the depth of this DAG is just $D(n) = \Theta(\log n)$, the height of the tree. Thus, as n grows, so does the average available parallelism, in the amount of $W(n)/D(n) = \Theta(n/\log n)$. By this measure, this algorithm is a better one than the sequential algorithm, just as we would expect.

In the specific case of Figure 2.3, where $n = 16$, the work $W(16) = 31$ nodes and the depth $D(16) = 5$. (Imagine that the "input" nodes at the top of Figure 2.3 represent load operations to retrieve the values.) Then, $W(16)/D(16) = 31/5 = 6.2$. Thus, there are an average of ≈ 6 nodes available to be executed in parallel during the computation. This implies that we could not gainfully use more than about six "processors."[1]

[1] In this case, we really mean six adders.

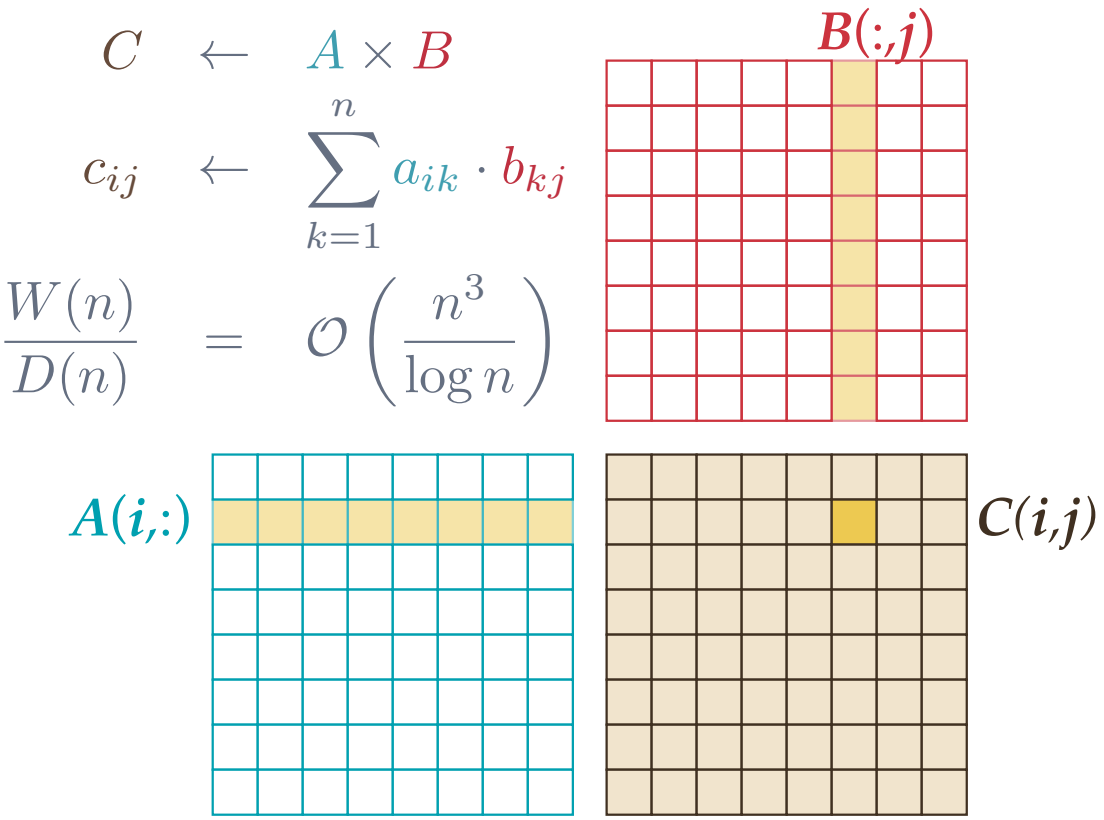


Figure 2.4: Work-depth example: Multiplying two $n \times n$ matrices.

Example: Matrix multiply. Consider the multiplication of two $n \times n$ matrices, $C \leftarrow A \times B$, as illustrated in Figure 2.4. Each output element c_{ij} is the dot product between $A_{i,:}$, which is row i of A, and $B_{:,j}$, which is column j of B. A dot product involves an elementwise multiplication of the vectors, followed by a sum-reduction of those results. This dot product requires $\Theta(n)$ operations. Since there are n^2 elements of C, the work $W(n) = \Theta(n^3)$.[2]

To compute $D(n)$, observe that the n^2 output elements have no dependencies among them; the only dependencies occur during the reduction to compute each output element. Thus, $D(n) = \Theta(\log n)$. The ratio of $W(n)/D(n) = \Theta(n^3/\log n)$ is asymptotically very high, and so our analysis confirms what we would expect, namely, that a matrix multiply has plenty of parallelism for even modestly sized values of n.

2.3 Characterizing I/O behavior: the External Memory Model

Besides parallelism, data movement is the other critical characteristic of an algorithm. To analyze data movement, we consider the classical *external memory model* [4].

Consider first a *sequential* processor with a two-level memory hierarchy consisting of a large but slow memory and a small fast memory of size Z words; work operations may only be performed on data that lives in fast memory. This fast memory may be an automatic cache or a software-controlled scratchpad; our analysis here is agnostic to this choice. We may further consider that transfers between slow and fast memory must occur in discrete transactions (blocks) of size L words. When we design an algorithm in this model, we again measure the work, $W(n)$; however, we also measure $Q(n; Z, L)$, the number of L-sized transfers between slow and fast memory for an input of size n. There are several ways to design either cache-aware or cache-oblivious algorithms and then analyze $Q(n; Z, L)$ [4, 36, 56, 102]. In either case, when we design an algorithm in this model, we again aim for work-optimality while also trying to maximize the algorithm's *computational intensity*, which is $W(n)/(Q(n; Z, L) \cdot L)$. Intensity is the algorithmic ratio of operations performed

[2] We assume a conventional matrix multiply, rather than an asymptotically faster algorithm, such as Strassen's [28].

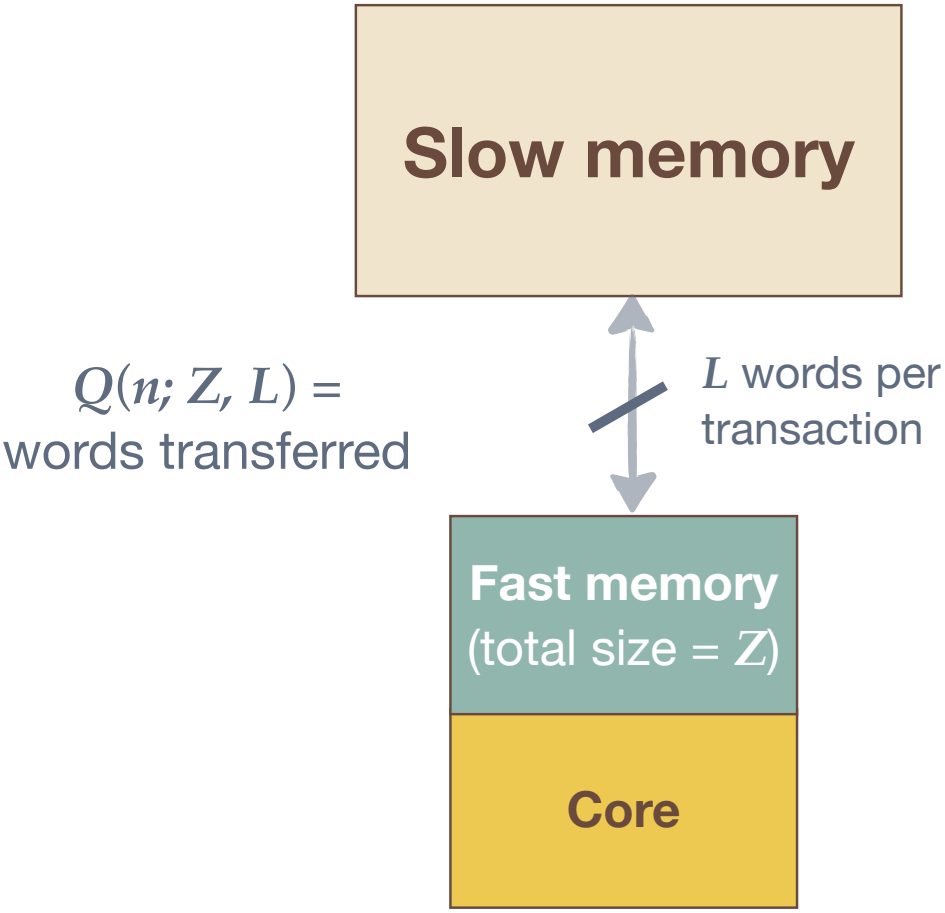


Figure 2.5: An abstract machine with a two-level memory hierarchy. The fast memory (e.g., cache or local-store) can hold Z words. Our goal is to design algorithms that, given an input of size n, are work-optimal but also minimize the number of transfers $Q(n; Z)$ between the slow and fast memories (alternatively, that are work-optimal and *maximize* the *computational intensity*, $W(n)/Q(n; Z)$.

to words transferred and, in practice, is often converted and cited as the algorithm's "flop-to-byte" ratio [97].

Example: Reduction. Let us again consider the sum-reduction problem (Figure 2.3), but this time analyze its I/O behavior. For simplicity, assume n, Z, and L are all powers of 2 with $n > Z > L$. A reduction tree with Z input nodes, where the input nodes are stored contiguously in slow memory, requires exactly Z/L transfers and $Z - 1$ operations (additions) to evaluate. These are just the transfers to load the input nodes, after which no further I/Os are necessary to sum the elements. Any n-element list can be broken up into n/Z partial reduction trees, thereby incurring $Q(n; Z, L) = n/L$ transfers and $W(n) = n - Z$ additions. That is, we need only perform the compulsory loads and the intensity is $1 - Z/n = \Theta(1)$.

In terms of the architecture, this intensity is independent of Z and L. Therefore, provided the architecture can deliver at least the required number of flops per byte, adding more cache or increasing the line size cannot fundamentally improve performance or scaling.

Example: Matrix multiply. Algorithms for conventional matrix multiply perform $W(n) = \Theta(n^3)$ operations on $\Theta(n^2)$ words of data; thus, a naïve lower bound on $Q(n; Z, L)$ would be $\Omega(n^2/L)$ transfers. The intensity could therefore be as high as $\mathcal{O}(n)$.

Consider two candidate algorithms for matrix multiply. The first is the one shown in Figure 2.4, which is based on performing n^2 reductions of length n each. This algorithm results in $Q(n; Z, L) = \mathcal{O}(n^3/L)$ transfers, assuming the preceding analysis for reductions. The intensity is therefore just $\mathcal{O}(1)$. Compared to our naïve estimate of $\mathcal{O}(n)$, it is likely there is a better algorithm.

The well-known alternative appears in Figure 2.6. This algorithm updates *blocks* of C at a time, rather than a single element of C, and streams through blocks of A and B as shown. That is, the algorithm performs block updates instead of row/column dot products. In this case, if

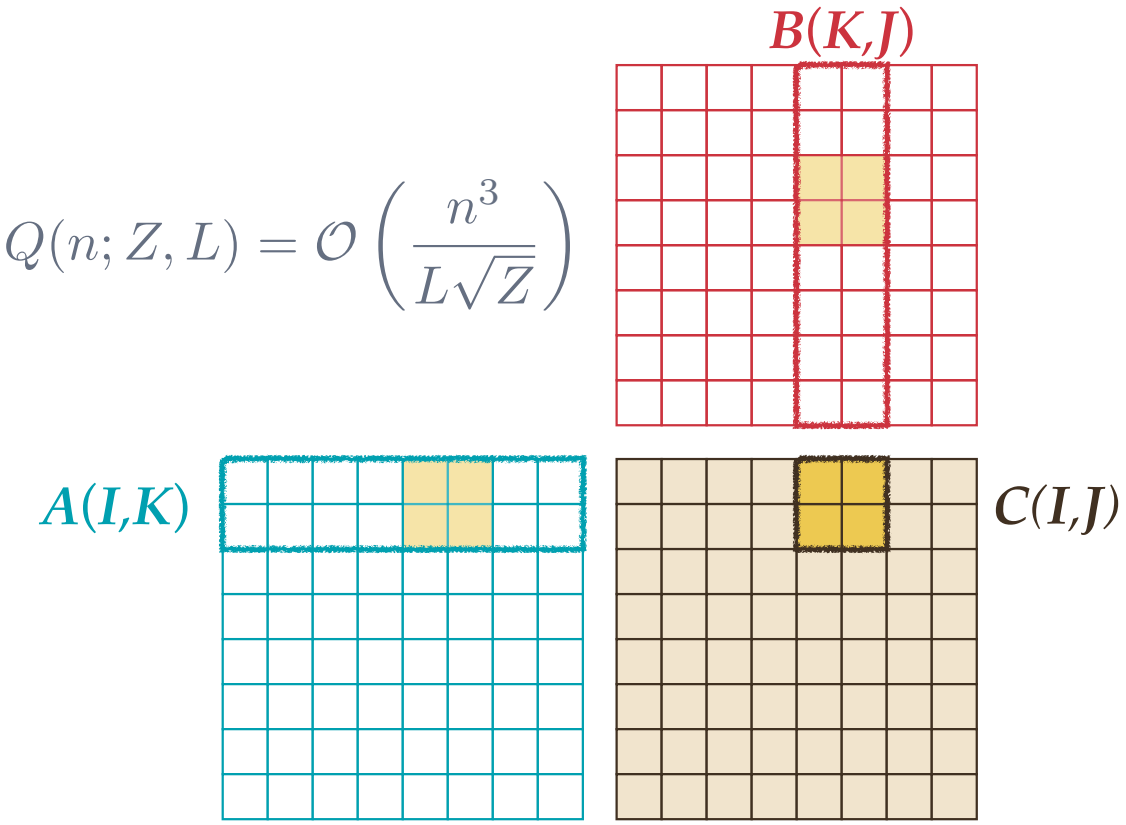


Figure 2.6: A blocked matrix multiply performs $\mathcal{O}\left(n^3/(L\sqrt{Z})\right)$ I/Os in the external memory model, assuming that (i) each of the three blocks of A, B, and C needed to compute an output block of C are stored contiguously; and (ii) that these three blocks just fit into the fast memory.

- the block size is $b \times b$;
- three blocks can be fit in fast memory at once, i.e., $3b^2 \leq Z$; and
- blocks are stored contiguously, so that just b^2/L transfers are required to load a block;

then it can be shown that $Q(n; Z, L) = \mathcal{O}\left(n^3/(L\sqrt{Z})\right)$; therefore, the intensity is $\mathcal{O}\left(\sqrt{Z}\right)$. If $Z = \Omega\left(L^2\right)$, the intensity of this algorithm is much higher than that of the naïve algorithm. In practice, a local-store or cache size Z is typically much larger than the corresponding minimum transfer or line size L, so this assumption is likely to hold.[3]

One might reasonably ask whether this value of $Q(n; Z, L)$ can be improved upon further, given our naïve lower-bound estimate of $\mathcal{O}\left(n^2\right)$. For general matrices and the conventional matrix multiply algorithm, the answer is that no algorithm can move fewer words than the blocked algorithm—the blocked algorithm is asymptotically optimal. For a highly readable analysis of this case, see Irony et al. [53]. So-called *cache-oblivious* approaches also only match this bound [37].

A few summary observations about I/O. The case of I/O analysis is evidently more complex than the analysis of parallelism in the following ways:

- The I/O analysis is not independent of the machine parameters, the way the work-depth analysis was independent of the number of processors, p.
- The two example I/O analyses both included assumptions of *contiguous layouts*. That is, inclusion of the parameter, L, forces the algorithm designer to take data layout into account.
- Recall that for matrix multiply, we were able to state a *lower bound* on the number of I/O transfers for *any* algorithm. This is one area in which theoretical analysis, when further refined to account for additional architectural details, can provide insights into performance that are difficult to extract from code or benchmarks.

[3] An exception to this rule is a TLB, which for typical configurations is like a cache with very large lines and having a capacity of a small number of lines.

In addition, we might ask what the relative asymptotic pay-off from increasing hardware memory resources (e.g., Z and L) is relative to increasing hardware parallelism (e.g., p). For example, for compute-rich matrix multiply, we see that doubling Z yields a $\sqrt{2}$ reduction in I/Os, whereas doubling L cuts I/Os in half. Then, if the cost of doubling Z and doubling L are equal, it would be more cost effective to double L. This analysis confirms an intuition behind GPU design of favoring significant increases in memory level parallelism (larger L) over bigger caches (larger Z).

2.4 Combined analyses of parallelism and I/O-efficiency

The preceding discussion treats parallelism and locality (I/O-efficiency) separately. However, for a GPGPU platform, we might prefer an analysis that considers these two algorithmic attributes simultaneously.

There are several models that combine parallel and I/O analysis within a single framework [8, 14, 16, 25, 92]. These differ primarily in whether

- they consider private [8] or shared caches, or both [14, 16, 25];
- they model only I/O transactions [8, 14, 16, 25] or include other costs, such as latency and bandwidth [92];
- the resulting algorithms are resource-aware (e.g., cache-aware) [8, 14, 92] or resource-oblivious [16, 25];
- and whether scheduling strategies or hints are assumed as part of the execution model [8, 14, 16, 25, 92] or not [92].

From this work, important design principles are emerging that should be directly relevant to current and future GPGPU platforms. Here is one example. Blelloch, Gibbons, and Simhadri suggest that nested-parallel algorithms with low depth and low *sequential* cache complexity[4] are likely to have good cache-efficient parallel behavior as well [16]. By se-

[4] . . . in the cache-oblivious model [36].

quential cache complexity we mean a "natural" sequential ordering of the nested-parallel execution, such as depth-first execution of threads. More precisely, suppose we design a nested-parallel algorithm with work $W(n)$, depth $D(n)$, and sequential cache misses $Q_1(n; Z, L)$ on a one-level memory hierarchy with private caches of size Z each and line-size L, and use work-stealing scheduling. Then, Blelloch et al. show that the total number of cache misses $Q_p(n; Z, L)$ in a parallel execution is bounded as follows:

$$Q_p(n; Z, L) < Q_1(n; Z, L) + \mathcal{O}\left(\frac{p \cdot Z \cdot D(n)}{L}\right) \tag{2.2}$$

with high probability [2]. One might have reasonably guessed an upper-bound of $p \cdot Q_1(n; Z, L)$, i.e., that the algorithm could incur as many as p times more misses in the multithreaded case, but Equation (2.2) shows otherwise. Note also that the algorithmic depth shows up in the bound; the principle, then, is that low depth is good not just for parallelism, but also for locality.

Though this kind of result is interesting, there is not yet a clear consensus on which of the various analysis models is "best," either in terms of the ease in which one can design algorithms in the model (compared to, say, designing sequential RAM or parallel PRAM algorithms) or the quality of the resulting algorithms when implemented for a real machine. This lack of consensus likely stems in part from the current instability in the architectural design space. Nevertheless, we feel it is useful and important to track the theoretical developments, as they could inform future hardware design as well, as the next section suggests.

2.5 Abstract and concrete measures

Work, depth, and the number of I/Os or cache misses are all abstract quantities, whereas the ultimate goal—if it is possible—would be to make stronger statements about execution time and time-scalability. Such a time-based analysis minimally requires an architectural cost model. If we are able to do so, what would the theory tell us?

Consider an example of the generic manycore processor system shown in Figure 2.7. This system has p cores, each of which can deliver a maximum of C_0 operations per unit time;

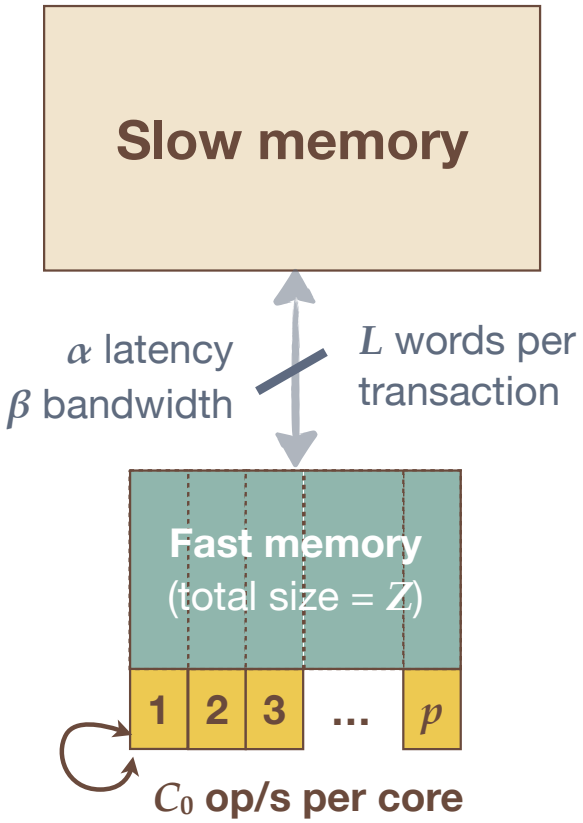


Figure 2.7: An abstract manycore processor with p cores, each with a peak execution rate of C_0 operations per unit time; an aggregate fast memory capacity of Z words partitioned among the cores; and a channel between slow and fast memory such with latency α time units, bandwidth β words per unit time, and minimum transaction size L words.

a fast memory of total size Z words, partitioned among the cores; and a memory system whose cost to transfer $m \cdot L$ words is $\alpha + m \cdot L/\beta$, where α is the latency and β is the bandwidth in units of words per unit time.[5]

Equation (2.1) gives us a way to estimate the time, $T_{\text{comp}}(n; p)$, just to complete the computational work for this system. To estimate the I/O time, $T_{\text{mem}}(n)$, suppose we know $Q_p(n; Z, L)$ and charge the full latency α for each node on the critical path. Let us further assume that all $Q_p(n; Z, L)$ memory transfers will in the best case be aggregated and pipelined by the memory system and thereby be delivered at the peak bandwidth, β. Then,

$$T_{\text{mem}}(n; p, Z, L, \alpha, \beta) = \alpha \cdot D(n) + \frac{Q_p(n; Z, L)(n) \cdot L}{\beta}. \tag{2.3}$$

(This cost model is not definitive, but sufficient to illustrate a few points.)

A necessary condition for this computation to run at peak speed (i.e., to be compute-limited) is that the compute time exceed the memory time, i.e., $T_{\text{comp}} \geq T_{\text{mem}}$. If this condition holds, then we can *in principle* hide the cost of data movement. We refer to this constraint as a *balance constraint*, which is similar to the classical definition of system balance proposed by Kung, among others, and as used in queuing theory [21, 48, 59, 65]. After imposing this balance constraint and applying a little algebra, we arrive at Equation (2.4):

$$\underbrace{\frac{p \cdot C_0}{\beta}}_{\text{balance}} \left(1 + \underbrace{\frac{\alpha \cdot \beta / L}{Q_p(n; Z, L)/D(n)}}_{\text{Little's Law}} \right) \leq \underbrace{\frac{W(n)}{Q_p(n; Z, L) \cdot L}}_{\text{intensity}} \left(1 + \underbrace{\frac{p}{W(n)/D(n)}}_{\text{Amdahl's Law}} \right). \tag{2.4}$$

Compare Equation (2.4) to Kung's classical balance principle [59], which in our notation would have been:

$$\frac{p \cdot C_0}{\beta} \leq \frac{W(n)}{Q_p(n; Z, L) \cdot L}. \tag{2.5}$$

Equation (2.5) says that the machine's inherent balance point (left-hand side) should be no greater than the algorithm's inherent computational intensity (right-hand side). Indeed,

[5] This abstract machine is intended to very coarsely approximate an NVIDIA Fermi-class architecture, where Z might represent the aggregate register and multiprocessor-private first-level local-store and cache capacity.

Equation (2.5) also appears within Equation (2.4), but with two additional correction terms that, as it happens, embody familiar and intuitive performance engineering principles.

For example, the $\alpha \cdot \beta/L$ factor in Equation (2.4) is latency times bandwidth; if this factor is large, an algorithm designer can compensate by finding an algorithm with a high value of $Q_p(n; Z, L)/D(n)$, which we can interpret as the average degree of available memory-level parallelism. Indeed, this term is just a restatement of Little's Law. Similarly, we can as noted previously interpret the factor of $W(n)/D(n)$ as the inverse of the Amdahl fraction, making the term in which it appears an incorporation of Amdahl's Law. Interestingly, Equation (2.4) unifies and relates these classical principles [30]. Of course, Equation (2.4) is still somewhat notional, as the algorithm characteristics—$W(n)$, $D(n)$, and $Q_p(n; Z, L)$—are typically computed in a big-O sense with undetermined constants. However, one could imagine estimating these characteristics empirically. In any case, it is reassuring that the theoretical models, when applied to a GPGPU-like system, yield familiar classical principles.

2.6 Summary

In summary, the theory of cache-efficient parallel algorithm design, though somewhat in flux, suggests the following best-practices, all of which are relevant to GPGPU platforms.

- In terms of parallelism, design algorithms that are *work-optimal* and have *low depth*, i.e., having a $W(n)$ that asymptotically matches the work of the *best* sequential algorithm, and with $W(n) \gg D(n)$.
- In terms of memory locality, design *high intensity* algorithms, i.e., having $W(n) \gg Q_p(n; Z, L)$, paying attention to both limited memory capacity (Z) and block transfers (L).
- Seek algorithms with abundant *memory-level parallelism*, i.e., $Q_p(n; Z, L) \gg D(n)$.

We have purposefully omitted a discussion of how one might model the impact of the limited-bandwidth I/O channel to the GPGPU co-processor, which today is implemented by PCI Express (PCIe). In a sense, this cost is straightforward to model if one assumes computational schemes in which one transfers all inputs to the GPGPU, runs the GPGPU code, and then copies the results back. However, we fully expect that in the long run, the PCIe interface to the GPGPU co-processor will disappear as GPGPU units move on-die with what are now the host processors.

Chapter 3

From Principles to Practice: Analysis and Tuning

Having designed a high-level algorithm according to the principles of Section 2, the next step is implementation and performance engineering for scalability. Though there are no hard-and-fast rules, there are many broadly applicable heuristics. In this section, we use the theory of the preceding section and ask how it guides and informs practical performance engineering onto GPGPU platforms. For concreteness, we will use a case study based on an algorithm used in n-body particle simulations known as the *fast multipole method*, or FMM [44].[1] Our specific example describes the implementation for GPGPUs developed as part of a blood flow simulation project [22, 23, 60, 78], though n-body problems in general appear in diverse applications in graphics, computational physics, gaming, and statistical machine learning, among numerous others [42].

3.1 The computational problem: Particle interactions

In an n-body problem, we wish to compute the effect that a set of *source* particles has on a set of *target* particles. For instance, the particles may be stars whose trajectories we wish to predict given that every star experiences a gravitational force from all other stars. More

[1] We consider a specific variant of the FMM referred to as a "kernel-independent" method. The algorithm is complex and its details beyond the scope of this Synthesis Lecture; we instead refer the interested reader elsewhere for details [43, 101].

formally, suppose we are given a system of n sources, with positions given by $\{s_1, \ldots, s_n\}$, and n targets, with positions $\{t_1, \ldots, t_n\}$.[2] Then, our goal in solving an n-body problem is to compute the n sums,

$$\phi(t_i) = \sum_{j=1}^{n} \mathcal{K}(s_j, t_i) \cdot \delta(s_j), \quad i = 1, \ldots, n \quad , \tag{3.1}$$

where $\phi(t)$ is called the *potential* at target point t; $\delta(s)$ is the *density* at source point s; and $\mathcal{K}(s,t)$ is an *interaction kernel* that specifies the effect we are modeling. For instance, the so-called single-layer Laplace kernel that models particle interactions due to electrostatic or gravitational forces has $\mathcal{K}(x,y) \propto \frac{1}{||x-y||}$, which expresses an interaction strength that is inversely proportional to the distance between x and y. Since each sum requires $\mathcal{O}(n)$ operations and there are n such sums, calculating the sums in the straightforward way requires $\mathcal{O}(n^2)$ operations.

3.2 An optimal approximation: the fast multipole method

The fast multipole method (FMM) *approximately* computes the sums of Equation (3.1), reducing the total number of operations from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. The algorithm also accepts as input a desired level of accuracy, which affects the complexity constant [43]. This significant asymptotic improvement comes from two key ideas: (i) using a *tree* to organize the points spatially; and (ii) using *fast approximate evaluation*, in which we perform "summary computations" at each node of the tree, using a constant number of overall tree traversals with constant work per tree node. This section provides the background on the FMM needed to understand the GPU tuning case study.

To build the tree, the usual method is as follows, as Figure 3.1 depicts for a two-dimensional example. Begin by creating a root node corresponding to the entire spatial region. Next, subdivide this root node into a fixed number of child nodes corresponding to disjoint spatial subregions of equal size. Lastly, repeat this subdivision process recursively until all leaf

[2] For simplicity, we assume the number of source and target points is the same, though in general they may differ. Moreover, the sets of points may in fact be exactly the same, with the sum excluding the computed target.

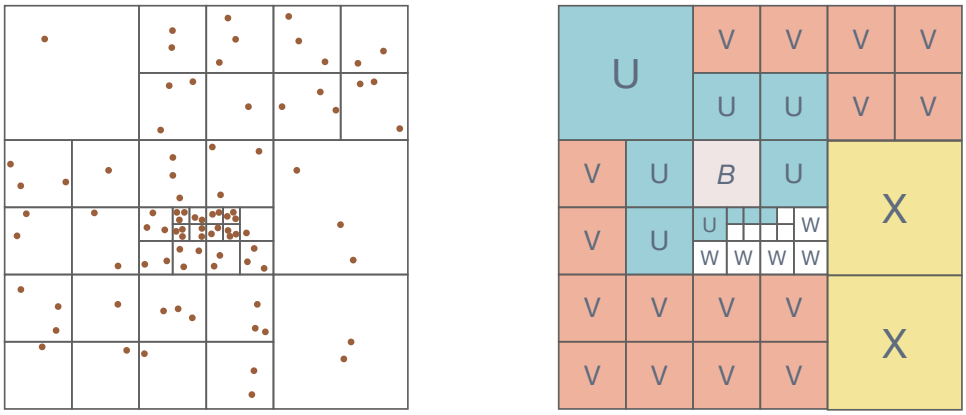


Figure 3.1: (Left) Leaves of an adaptive quad-tree for a spatial domain of points, where each leaf contains no more than $q = 3$ points. (Right) How leaves are distributed among the U, V, W, and X lists, for a given tree node B.

nodes contain at most q points, where q is a tunable parameter that controls the overall cost of the algorithm.[3]

Given the tree, the fast approximate evaluation consists of several phases. Each phase iterates over the tree nodes and performs a calculation involving the tree node and a subset of the other tree nodes. These phases differ in their computational intensity as well as their type and degree of parallelism. For concreteness, we consider two of the most expensive of these phases next.

In the so-called *direct evaluation* or *U-list* phase of the FMM, each leaf node B has an associated U-list, denoted $\mathcal{U}(B)$, which contains B itself and all leaf nodes adjacent to B. (See Figure 3.1.) The U-list computation iterates over all $B' \in \mathcal{U}(B)$, performing the exact summation in Equation (3.1) where the target points in B serve as the targets while the source points in B' serve as the sources. Figure 3.2 shows hypothetical sets of target nodes and source nodes, and illustrates the sparse dependencies between a target node B and its $\mathcal{U}(B)$. The sequential pseudocode for the U-list phase appears in Algorithm 1.

Since we've constructed the tree so that each node contains at most q points, then the cost

[3]Note that *accuracy* is controlled by a separate parameter that is independent of q [43, 101]. However, the setting of q that minimizes execution time does depend on the desired accuracy, among other factors.

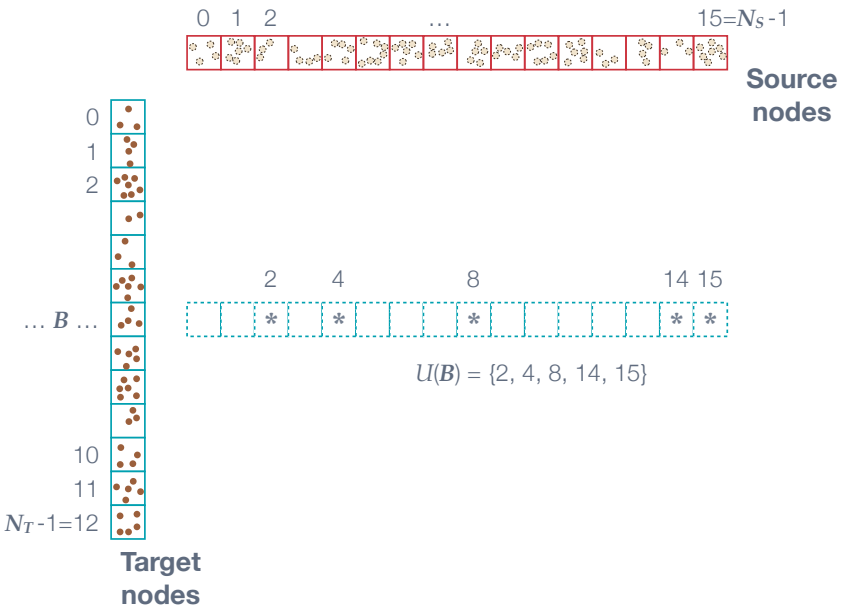


Figure 3.2: Each target node, B, will interact with a subset of the source nodes given by $\mathcal{U}(B)$.

Algorithm 1 FMM$_U$(target points $\mathcal{T}$, potentials Φ, source points $\mathcal{S}$, densities Δ, tree T, U-list $\mathcal{U}$)

```
for each target leaf node, B ∈ T do
    for each target point t ∈ B do
        for each neighboring source node, S ∈ U(B) do
            for each source point s ∈ S do
                Φ_t += K(S_s, T_t) · Δ_s /* Kernel evaluation */
```

of evaluating Equation (3.1) for each (B, B') pair is $\mathcal{O}(q^2)$. When the points are uniformly distributed in d-dimensional space, then $|U(B)| = 3^d$, which we may regard as a constant if d is constant. Moreover, the number of leaf nodes will be $\mathcal{O}(n/q)$, leading to an overall complexity of $\mathcal{O}(nq)$ for the U-list phase.

Another important phase is the so-called *V-list* phase. Again, for each leaf B we associate a bounded list of other leaves, $V(B)$.[4] (See Figure 3.1.) The overall cost of the V-list phase is, as it happens, $\mathcal{O}(n/q)$. Comparing to the cost of the U-list, we see that there is a trade-off between the costs of the U-list and V-list, controlled by the algorithmic tuning knob, q. Figure 3.3 illustrates this trade-off as measured in an actual experiment [23].

[4]The V-list, $V(B)$, for each leaf B is the set difference between all of the children of the neighbors of the parents of B and $U(B)$.

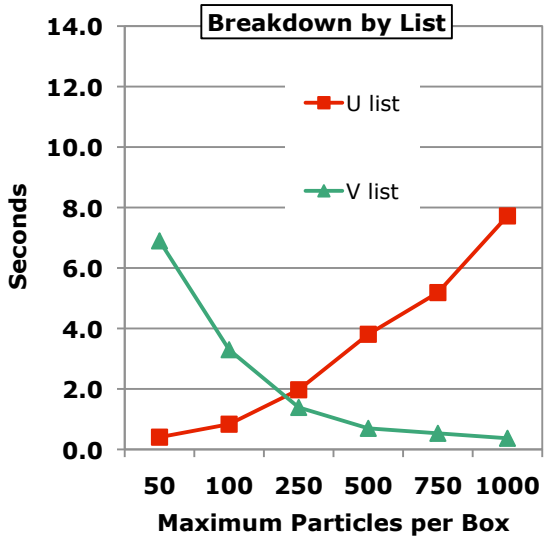


Figure 3.3: The measured trade-off between the costs of the U-list and V-list phases of the FMM, as a function of the algorithmic tuning parameter, q (maximum number of points per leaf node). This figure appeared in Chandramowlishwaran et al. (2010) [23].

3.3 Designing a parallel and I/O-efficient algorithm

The parallelism of the U-list phase (Algorithm 1) is easy to see. First, all target points are independent, so the iterations of the two outermost loops (Algorithm 1, lines 1–2) may be executed in parallel. Secondly, the innermost loops constitute a reduction, given the target point t. This loop nest performs no additional work, so the overall parallel work $W(n) = \mathcal{O}(nq)$ is the same as the sequential algorithm. Regarding the depth, the outermost loops have unit depth while the innermost loops have $\log n$ depth, owing to the reduction. Thus, the overall depth is $D(n) = \mathcal{O}(\log n)$. The available parallelism is then $\mathcal{O}(nq/\log n)$, which grows as n increases. Thus, for a sufficiently large n, there should be plenty of parallelism.

What about I/O complexity? We begin with the sequential case, and furthermore assume a 3-D problem with a uniform point distribution. In that case, each point is 4 words—3 coordinates plus either a potential for a target point or a density for a source point—and each target node B will have $|\mathcal{U}(B)| = 27$, since there will be a source node at the same location and 26 neighbors a distance of 1 node away.

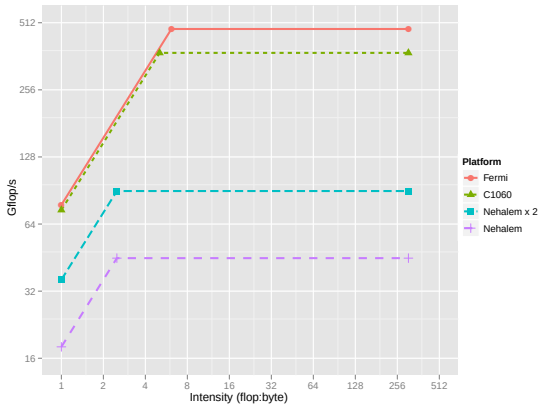
Suppose there is just enough cache or local storage, Z, to store all of the points for both one target node and one source node. That is, $Z = q \times (4 \text{ words/point}) \times 2 = 8q$. In practice, a cache or local store is often at least this large, or q can be chosen so that this equality is true. Then, for each target node B, we will load q target points, $27q$ source points, perform $27q^2$ interactions, and lastly store q target densities. Thus, the entire U-list computation, over all n $qmax$ target nodes, will transfer $Q(n; Z = 8q) = 116n$ words of data. If each node-to-node interaction requires κq^2 flops, then the overall intensity of $\kappa q/116$ flops per word.

3.4 A baseline implementation

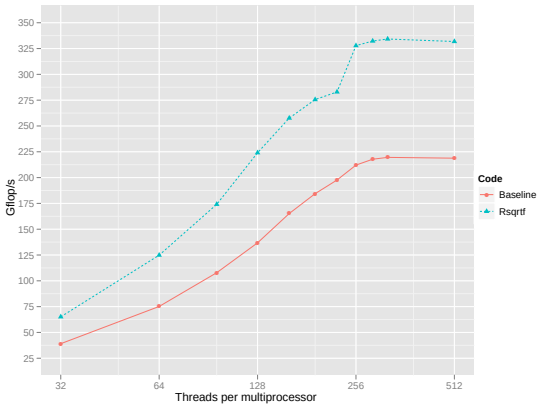
Given the algorithm sketched above, we can now translate Algorithm 1 into a concrete baseline code implementation, such as the CUDA implementation of Listing 3.1. This is single program multiple data (SPMD) style code that every GPU thread executes.

Listing 3.1: An initial CUDA kernel implementation of Algorithm 1. Color-ize, map data structures to algorithm.

```
1  int di = threadIdx.x; // Local thread ID
2  int tid_min = Tptr[tnodeid]; // First point in this box
3  int tid_max = Tptr[tnodeid+1]; // First point in the next box
4  for (int tid = tid_min + di; tid < tid_max; tid += q) {
5    float4 tgt = T[tid]; // Load point (x, y, z, potential)
6    for (int snode = snode_min; snode < snode_max; ++snode) {
7      int snodeid = Ulist[snode];
8      int sid_min = Sptr[snodeid];
9      int sid_max = Sptr[snodeid+1];
10     for (int sid = sid_min; sid < sid_max; ++sid) {
11       float4 src = S[sid]; // Load source (x, y, z, density)
12       float3 dr = make_float3 (tgt.x − src.x, tgt.y − src.y, tgt.z − src.z);
13       float rsq = (dr.x∗dr.x) + (dr.y∗dr.y) + (dr.z∗dr.z);
14       float r = sqrtf (rsq);
15       tgt.w += src.w / r;
16     } // sid
17   } // snode
18   T[tid].w = tgt.w; // Store potential
19 } // tid
```



(a) Roofline analysis helps to bound performance, and distinguish between memory bandwidth- vs. compute-boundedness.



(b) An empirical test on an NVIDIA C1060 to estimate compute-bound performance for the specific instruction mix of interest.

Figure 3.4: Setting an optimization goal. As discussed in Section 3.5, we expect that by tuning q, we can make Algorithm 1 compute-bound with a flop:byte ratio of at least 10.

3.5 Setting an optimization goal

Our first-order task is to use our concurrency and I/O analysis to set an optimization goal.

For instance, consider that a typical GPU today is capable of about 1 Tflop/s with a bandwidth of about 100 GB/s, or an "ideal" flop-to-byte balance ratio of about 10 flops per byte. For something like a gravitational or electrostatic potential, $\kappa \approx 11$ flops as shown in Listing 3.1. Thus, if the data is single-precision (4 bytes per word), then to match 10 flops per byte we need $q \geq (10 \text{ flops / byte}) \times (464 \text{ bytes}) / (11 \text{ flops}) \approx 422$ points, or about 1.6 KB in order to be compute-limited rather than bandwidth limited. This capacity is well within the typical local store size on a GPU multiprocessor, we should expect it may be possible to be compute-limited, though we will need to tune q empirically to ensure such a large value does not put us too far away from the true overall time minimizer (recall Figure 3.3).

We begin by considering a pure CPU baseline implementation, running on a dual-socket Intel x86 platform based on Nehalem processors, parallelized using OpenMP and explicit SIMD (SSE) vectorization. For this code, we observe performance between 60-90 billion floating-point operations per second (Gflop/s) in single-precision. As it happens, this is very close to single-precision peak for our platform, as Figure 4.8 confirms.

For our GPU implementation, which we said previously should be compute-limited, we can construct a microbenchmark that contains only the compute operations of lines 12–15 of Listing 3.1. That is, this microbenchmark omits any memory references, in order to estimate the maximum throughput we might expect for a truly compute-limited kernel. The results of such an experiment appear in Figure 3.4b, for two cases. In the first case, we test the instruction mix as shown in Listing 3.1. In the second case, we replace the separate `sqrtf` and division operations with a single call to the combined reciprocal square root function, `rsqrtf`. This replacement boosts performance by nearly an additional 50%, and our absolute performance target is 340 Gflop/s.

By contrast, the baseline GPU implementation of Listing 3.1 achieves roughly 70 Gflop/s on an NVIDIA C1060, which is on-par with the tuned CPU baseline as noted above. The innermost loop loads 16 bytes of data (line 11) for every 11 flops, giving a flop:byte ratio of $11/16 \approx 0.7$ flops per byte. Comparing against the roofline of Figure 4.8, we see that value is indeed roughly what we should expect.

3.5.1 Identifying candidate optimizations

Given that we expect compute-bound performance, we focus initially on memory hierarchy optimizations. Well-known techniques include local-store blocking (tiling), prefetching, and unroll-and-jam techniques, all of which bring data closer to the processor to improve reuse [5]. Each of these is, additionally, parametrized by a tuning parameter, e.g., block or tile size, prefetch distance, and unrolling depth. Additionally, for GPUs there is some folk-evidence that one might wish to consider whether to use an array-of-structures (AOS) or structure-of-arrays layout (SOA). (The reference code of Listing 3.1 packs coordinates into arrays of `float4` data, and so is an example of an AOS layout.) Once we have applied these memory hierarchy optimizations to put us in the compute-bound regime, we can then consider lower-level techniques, such as the use of a combined reciprocal square root operation. We briefly summarize how we might apply these optimizations next, in the context of the U-list kernel; similar transformations for other kernels of the FMM may be

applied similarly.

Blocking (tiling) for shared memory. As noted in Figure 3.2, there is a natural blocking boundary at the level of target node-to-source node interactions. Thus, a natural idea is to load as many source and target points as possible into the GPU multiprocessor local store ("shared memory" in NVIDIA CUDA parlance). For Listing 3.1, it would be natural to have every thread within a thread block load one or more source points into shared memory, with an appropriate synchronization call (`__syncthreads()`) to ensure the data is available. This transformation is very similar to what is performed for the direct n-body problem, a well-known example in the GPU literature [9, 75].[5]

Prefetching. As an alternative to blocking, which is a form of prefetching, one might also consider the following simpler style of prefetching. In particular, imagine that we simply modify Listing 3.1 to prefetch the next source point into a local register variable. This method might be expected to work well if the latency of the innermost loop is large enough to hide the source point load.

Unroll-and-jam. The baseline code uses an owner-computes strategy, assigning just one output (target) point per thread and relies on many threads to hide latency. We could instead assign multiple outputs per thread, in order to increase instruction-level parallelism (ILP) and register-level locality. Doing so requires unrolling the target point loop at line 4 of Listing 3.1, and then fusing the innermost loops. The cost is an increased instruction count and increased register pressure. However, by increasing ILP, we can also decrease the number of threads needed, and thereby actually increase the number of available registers per thread.

AOS vs. SOA. The main potential benefit of considering an SOA layout over an AOS layout is to increase the degree of aggregation of memory transactions. Recall our theoretical

[5] See also http://http.developer.nvidia.com/GPUGems3/gpugems3_ch31.html.

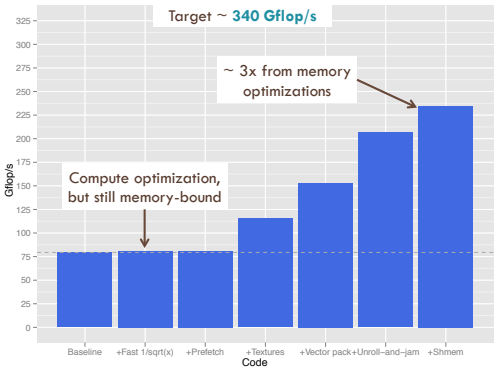
discussion of memory hierarchy traffic in Section 2.3, where we noted that it can be more cost-effective per word to increase memory transaction width over local store capacity; thus, the choice of SOA over AOS may be measurable.

3.5.2 Exploring the optimization space

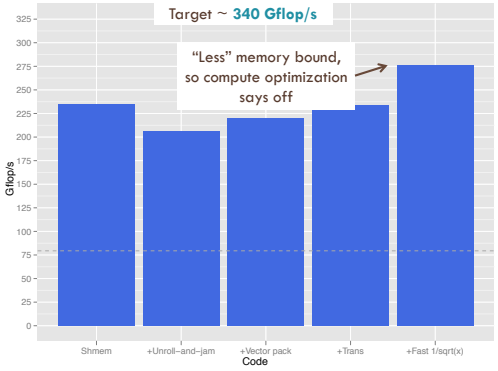
The optimizations of Section 3.5.1 define a *space* of candidate optimizations, but do not tell us which ones will improve performance and by how much. In this section, we consider an ad hoc (heuristic) method for exploring that space. This particular method happens to find the optimal implementation one a particular platform, but by no means guarantees it. More refined models of the interaction among the algorithm, code, and architecture are an active area of research; we review some of this work as it applies to GPGPU platforms in Section 4.2.

The ad hoc method is simple, and is based loosely on the design of experiments techniques from statistics [18]. Let C_0 denote the baseline code, and let $S = \{s_1, s_2, \ldots, s_l\}$ be the set of all optimizations we are considering. Suppose for simplicity that these optimizations may be combined in all $l!$ combinations, assuming ordering is unimportant. To choose a subset of these optimizations, we begin by trying each of the l optimizations individually. Suppose s_a is the best one. Then, we create a new code C_1 that includes the optimization s_a. Using C_1, we repeat the same experiment, using only the optimizations that remain when we remove s_a from S, i.e., from $S - \{s_a\}$. If the best of these is s_b, we then generate a new code C_2 from C_1, so that it includes both optimizations $s_a \in S$ and $s_b \in S - \{s_a\}$. We repeat this process until no optimizations remain to consider.

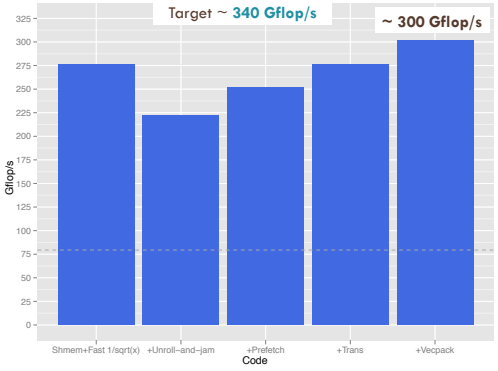
Applying this method to Listing 3.1 using the optimizations of Section 3.5.1 yields the sequence of experiments shown in Figure 3.5. Each plot represents a stage in which we have a set of candidate optimizations, which we apply separately to an initial code; the initial code is the leftmost bar in each subplot. Observe that in the first stage, Figure 3.5a, the local store-based optimization has the biggest pay-off and brings the code implementation the closest to being compute bound; by contrast, reciprocal square root has no pay-off, but



(a) Just one optimization applied.



(b) Two optimizations applied: the best from Figure 3.5a and paired with all remaining optimizations.



(c) Three optimizations applied: the best pair from Figure 3.5b combined with each of the remaining.

Figure 3.5: Applying the ad hoc optimization search method of Section 3.5.2, to a baseline code with a performance of 75 Gflop/s.

mostly because the code is not yet in the compute-bound regime, not because it cannot help. The final implementation achieves around 300 Gflop/s through a combination of blocking for shared memory, reciprocal square root, and use of SOA, the last of which is labeled "vecpack" in the figure. Interestingly, SOA was never optimal until combined with the other two optimizations. Overall, we achieve a level of performance that seems reasonably close to our microbenchmark-based estimate of a 340 Gflop/s upper-bound.

3.5.3 Summary

The case study in this section sketches how one might begin with an algorithm analysis and use that to drive a subsequent code implementation and tuning process. The key ingredients are:

- beginning with a principled analysis of concurrency and locality;
- using microbenchmarks to help establish bounds on performance;
- enumerating a "space" of candidate optimizations;
- and lastly, systematically exploring that space.

Still, the space of potential optimizations may in general be very large. The open question, which we take up in the next section, is to what extent we can use modeling or other quantitative analysis to help define and explore this space.

Chapter 4

Using Detailed Performance Analysis to Guide Optimization

In previous sections, we explained how to identify performance optimization candidates in a high-level algorithm and performance analysis. The work-depth model considers primarily the actual computation work and memory operations, which are represented at a high-level. In this chapter, we look at the performance behavior which exploits low-level information, such as instructions and memory transactions, instead of or in addition to high-level information, and then discuss how such information can guide optimization.

4.1 Instruction-level Analysis and Tuning

In this section, we describe an instruction-level analysis that also includes details of the microarchitecture aspects of processors. Much performance modeling has been done for both CPUs and GPUs. A summary of other work is discussed in Section 4.2. Here, we explain a model based on the approach of Sim et al. [85]. First, we describe how to model the performance behavior of GPGPUs and then show how to identify performance bottlenecks that could be optimized. The performance model is based on NVIDIA's Fermi architecture.

4.1.1 Execution Time Modeling

First, we define T_{exec} as the overall execution time, which is a function of T_{comp}, T_{mem}, and T_{overlap}, as shown in Equation (4.1).

$$T_{\text{exec}} = T_{\text{comp}} + T_{\text{mem}} - T_{\text{overlap}} \tag{4.1}$$

The execution time is calculated by adding computation and memory costs while subtracting the overlapped cost due to the multi-threading feature in GPGPUs. Figure 4.1 illustrates the equation when four warps are running. Each of the three inputs of Equation (4.1) is described in the following.

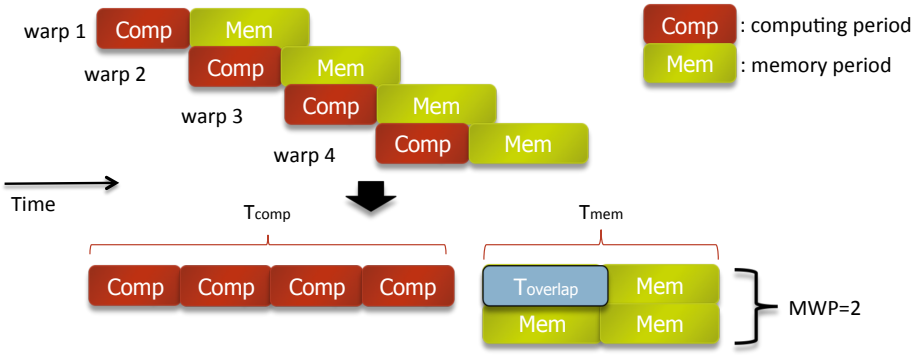


Figure 4.1: Visualization of Execution Time (There are four memory requests but because of memory-warp level parallelism, only two memory requests contribute to the total execution time.

Calculating the Computation Cost, T_{comp}

T_{comp} is the amount of time to execute compute instructions (excluding memory operation waiting time, but including the cost of executing memory instructions) and is evaluated using Equations (4.2) through (4.10).

We consider the computation cost as two components, a parallelizable base execution time plus overhead costs due to serialization effects:

$$T_{\text{comp}} = \underbrace{W_{\text{parallel}}}_{\text{Base}} + \underbrace{W_{\text{serial}}}_{\text{Overhead}}. \tag{4.2}$$

The base time, W_{parallel}, accounts for the number of operations and degree of parallelism, and is computed from basic instruction and hardware values as shown in Equation (4.3):

$$W_{\text{parallel}} = \underbrace{\frac{\#\text{insts} \times \#\text{total\_warps}}{\#\text{active\_SMs}}}_{\text{Total instructions per SM}} \times \underbrace{\frac{\text{avg\_inst\_lat}}{\text{ITILP}}}_{\text{Effective throughput}}. \tag{4.3}$$

The first factor in Equation (4.3) is the total number of dynamic instruction for one thread, and the second factor indicates the *effective* instruction throughput. Regarding the latter, the average instruction latency, avg_inst_lat, can be approximated by the latency of FP operations in GPGPUs. When necessary, it can also be precisely calculated by taking into account the instruction mix and the latency of individual instructions on the underlying hardware. The value, ITILP, models the possibility of inter-thread instruction-level parallelism in GPGPUs. The concept of ITILP was introduced in Baghsorkhi [11]. In particular, instructions may issue from multiple warps on a GPGPU; thus, we consider global ILP (i.e., ILP among warps) rather than warp-local ILP (i.e., the ILP of one warp). That is, ITILP represents how much ILP is available among all executing threads to hide the pipeline latency.

ITILP can be obtained as follows:

$$\text{ITILP} = \min\left(\text{ILP} \times N, \text{ITILP}_{\text{max}}\right) \tag{4.4}$$

$$\text{ITILP}_{\text{max}} = \frac{\text{avg\_inst\_lat}}{\text{warp\_size}/\text{SIMD\_width}}, \tag{4.5}$$

where N is the number of active warps on one SM, and SIMD_width and warp_size represent the number of vector units and the number of threads per warp, respectively. On the Fermi architecture, SIMD_width $=$ warp_size $=$ 32. ITILP cannot be greater than $\text{ITILP}_{\text{max}}$, which is the ITILP required to fully hide pipeline latency.

We model serialization overheads, W_{serial} from Equation (4.2) as

$$W_{\text{serial}} = O_{\text{sync}} + O_{\text{SFU}} + O_{\text{CFdiv}} + O_{\text{bank}}, \tag{4.6}$$

where each of the four terms represents a source of overhead—synchronization, SFU resource contention, control-flow divergence, and shared memory bank conflicts. We describe each overhead below.

Synchronization Overhead, O_{sync}: When there is a synchronization point, the instructions after the synchronization point cannot be executed until all the threads reach that point. If all threads progress at the same rate, there would be little overhead for the waiting time. Unfortunately, each thread (warp) progresses on its own progress based on the availability of source operands. Consequently a range of progress exists, which can sometimes vary widely. The causes of this range are mainly different DRAM access latencies (delay in queues, DRAM row buffer hit/misses, etc.) and control-flow divergences. As a result, when a high number of memory instructions and synchronization instructions exist, the overhead increases as shown in Equations (4.7) and (4.8):

$$O_{\text{sync}} = \frac{\#\text{sync\_insts} \times \#\text{total\_warps}}{\#\text{active\_SMs}} \times F_{\text{sync}} \tag{4.7}$$

$$F_{\text{sync}} = \Gamma \times \text{avg\_DRAM\_lat} \times \underbrace{\frac{\#\text{mem\_insts}}{\#\text{insts}}}_{\text{Mem. ratio}}, \tag{4.8}$$

where Γ is a machine-dependent parameter. Sixty-four is assumed for the modeled architecture.

SFU Resource Contention Overhead, O_{SFU}: This cost is primarily caused by the characteristics of special function units (SFUs) and is computed using Equations (4.9) and (4.10) below.

For most GPGPUs, expensive math operations such as transcendentals and square roots can be handled with dedicated execution units called *special function units* (SFUs). Since the execution of SFU instructions can be overlapped with other floating point (FP) instructions, with a good ratio between SFU and FP instructions, the cost of SFU instructions can almost be hidden. Otherwise, SFU contention can degrade performance. So, the visible execution cost of SFU instructions depends on the ratio of SFU instructions to others and the number of execution units for each instruction type. In Equation (4.9), the *visibility* is modeled by F_{SFU}, which is in $[0, 1]$. A value of $F_{\text{SFU}} = 0$ means none of the SFU execution costs is added to the total execution time. This occurs when the SFU instruction ratio is less than the ratio of special function to SIMD units, as shown in Equation (4.10).

$$O_{\text{SFU}} = \frac{\text{\#SFU\_insts} \times \text{\#total\_warps}}{\text{\#active\_SMs}} \times \underbrace{\frac{\text{warp\_size}}{\text{SFU\_width}}}_{\text{SFU throughput}} \times F_{\text{SFU}} \tag{4.9}$$

$$F_{\text{SFU}} = \min\left\{\max\left\{\underbrace{\frac{\text{\#SFU\_insts}}{\text{\#insts}}}_{\text{SFU inst. ratio}} - \underbrace{\frac{\text{SFU\_width}}{\text{SIMD\_width}}}_{\text{SFU exec. unit ratio}}, 0\right\}, 1\right\}. \tag{4.10}$$

Control-Flow Divergence and Bank Conflict Overheads, O_{CFdiv} and O_{bank}: The overhead of control-flow divergence (O_{CFdiv}) is the cost of executing additional instructions due, for instance, to divergent branches [50]. This cost is modeled by counting all the instructions in both paths. The cost of bank conflicts (O_{bank}) can be calculated by measuring the number of shared memory bank conflicts. Both O_{CFdiv} and O_{bank} can be measured using hardware counters.

Calculating the Memory Access Cost, T_{mem}

T_{mem} represents the amount of time spent on memory requests and transfers. This cost is a function of the number of memory requests, memory latency per each request, and the degree of memory-level parallelism. We model T_{mem} using Equation (4.11),

$$T_{\text{mem}} = \underbrace{\frac{\text{\#mem\_insts} \times \text{\#total\_warps}}{\text{\#active\_SMs} \times \text{ITMLP}}}_{\text{Effective memory requests per SM}} \times \text{AMAT}, \tag{4.11}$$

where AMAT (average memory access time) models the average memory access time, accounting for cache effects. We compute AMAT using Equations (4.12) and (4.13):

$$\text{AMAT} = \text{avg\_DRAM\_lat} \times \text{miss\_ratio} + \text{hit\_lat} \tag{4.12}$$

$$\text{avg\_DRAM\_lat} = \text{DRAM\_lat} + (\text{avg\_trans\_warp} - 1) \times \Delta. \tag{4.13}$$

avg_DRAM_lat represents the average DRAM access latency and is a function of the baseline DRAM access latency, DRAM_lat, and transaction departure delay, Δ. In GPGPUs, memory requests can split into multiple transactions. In the described model, avg_trans_warp represents the average number of transactions per memory request in a warp. Note that it is possible to expand Equation (4.12) for multiple levels of cache, which we omit for brevity.

We model the degree of memory-level parallelism through the notion of inter-thread MLP, denoted as ITMLP, which we define as the number of memory requests per SM that are concurrently serviced. Similar to ITILP, memory requests from different warps can be overlapped. Since MLP is an indicator of intra-warp memory-level parallelism, we need to consider the overlap factor of multiple warps. ITMLP can be calculated using Equations (4.14) and (4.15).

$$\text{ITMLP} = \min\left(\text{MLP} \times \text{MWP}_{\text{cp}}, \text{MWP}_{\text{peak\_bw}}\right) \tag{4.14}$$

$$\text{MWP}_{\text{cp}} = \min\left(\max\left(1, \text{CWP} - 1\right), \text{MWP}\right) \tag{4.15}$$

In Equation (4.14), MWP_{cp} represents the number of warps whose memory requests are overlapped during one computation period. MWP represents the *maximum* number of warps that can simultaneously access memory. However, depending on CWP, the number of warps that can concurrently issue memory requests is limited.

$\text{MWP}_{\text{peak\_bw}}$ represents the number of memory warps per SM under peak memory bandwidth. Since the value is equivalent to the maximum number of memory requests attainable per SM, ITMLP cannot be greater than $\text{MWP}_{\text{peak\_bw}}$.

Calculating the Overlapped Cost, T_{overlap}

T_{overlap} represents how much memory access cost can be hidden by multithreading. In the GPGPU execution, when a warp issues a memory request and waits for the requested data, the execution is switched to another warp. Hence, T_{comp} and T_{mem} can be overlapped to some extent. For instance, if multithreading hides all memory access costs, T_{overlap} will equal T_{mem}. That is, in this case, the overall execution time, T_{exec}, is solely determined by

the computation cost, T_{comp}. In contrast, if none of the memory accesses can be hidden in the worst case, then T_{overlap} is 0.

We compute T_{overlap} using Equations (4.16) and (4.17). In these equations, F_{overlap} approximates how much T_{comp} and T_{mem} overlap and N represents the number of active warps per SM as in Equation (4.4).

Note that F_{overlap} varies with both MWP and CWP. When CWP is greater than MWP (e.g., an application limited by memory operations), then F_{overlap} becomes 1, which means all of T_{comp} can be overlapped with T_{mem}. On the other hand, when MWP is greater than CWP (e.g., an application limited by computation), only part of the computation costs can be overlapped.

$$T_{\text{overlap}} = \min(T_{\text{comp}} \times F_{\text{overlap}}, T_{\text{mem}}) \tag{4.16}$$

$$F_{\text{overlap}} = \frac{N-\zeta}{N}, \quad \zeta = \begin{cases} 1 \ (\text{CWP} \leq \text{MWP}) \\ 0 \ (\text{CWP} > \text{MWP}) \end{cases} \tag{4.17}$$

Calculating CWP and MWP

$$\text{CWP} = \min\left(\text{CWP\_full}, N\right) \tag{4.18}$$

$$\text{CWP\_full} = \frac{\text{mem\_cycles} + \text{comp\_cycles}}{\text{comp\_cycles}} \tag{4.19}$$

$$\text{comp\_cycles} = \frac{\#\text{insts} \times \text{avg\_inst\_lat}}{\text{ITILP}} \tag{4.20}$$

$$\text{mem\_cycles} = \frac{\#\text{mem\_insts} \times \text{AMAT}}{\text{MLP}} \tag{4.21}$$

mem_cycles: memory waiting cycles per warp

comp_cycles: computation cycles per warp

#insts: number of instructions per warp (excluding SFU insts)

#mem_insts: number of memory instructions per warp

$$\text{MWP} = \min\left(\frac{\text{avg\_DRAM\_lat}}{\Delta}, \text{MWP}_{\text{peak\_bw}}, N\right) \tag{4.22}$$

$$\text{MWP}_{\text{peak\_bw}} = \frac{\text{mem\_peak\_bandwidth}}{\text{BW\_per\_warp} \times \text{\#active\_SMs}} \tag{4.23}$$

$$\text{BW\_per\_warp} = \frac{\text{freq} \times \text{transaction\_size}}{\text{avg\_DRAM\_lat}} \tag{4.24}$$

mem_peak_bandwidth: bandwidth between the DRAM and GPU cores (e.g., 144.0 GB/s in Tesla C2050)

freq: clock frequency of the SM processor

(e.g., 1.15 GHZ in Tesla C2050)

transaction_size: transaction size for a DRAM request

(e.g., 128B in Tesla C2050)

BW_per_warp: bandwidth requirement per warp

Potential Benefit Prediction

The potential benefit metrics indicate performance improvements when it is possible to eliminate the delta between the ideal performance and the current performance. Equations (4.25) and (4.26) are used to estimate the ideal compute and memory performance (time). Alternatively, an algorithm developer might provide these estimates.

$$T_{\text{fp}} = \frac{\text{\#FP\_insts} \times \text{\#total\_warps} \times \text{FP\_lat}}{\text{\#active\_SMs} \times \text{ITILP}} \tag{4.25}$$

$$T_{\text{mem\_min}} = \frac{\text{size\_of\_data} \times \text{avg\_DRAM\_lat}}{\text{MWP}_{\text{peak\_bw}}} \tag{4.26}$$

Then, the benefit metrics are obtained using Equations (4.27)-(4.30), where $\text{ITILP}_{\text{max}}$ is defined in Equation (4.5):

$$B_{\text{itilp}} = W_{\text{parallel}} - \frac{\#\text{insts} \times \#\text{total\_warps} \times \text{avg\_inst\_lat}}{\#\text{active\_SMs} \times \text{ITILP}_{\text{max}}} \tag{4.27}$$

$$B_{\text{serial}} = W_{\text{serial}} \tag{4.28}$$

$$B_{\text{fp}} = T_{\text{comp}} - T_{\text{fp}} - B_{\text{itilp}} - B_{\text{serial}} \tag{4.29}$$

$$B_{\text{memlp}} = \max\left(T'_{\text{mem}} - T_{\text{mem\_min}}, 0\right). \tag{4.30}$$

Model Parameter	Definition	Source
#insts	# of total insts. per warp (excluding SFU insts.)	Hardware performance counters
#mem_insts	# of memory insts. per warp	Hardware performance counters
#sync_insts	# of synchronization insts. per warp	Instruction analyzer
#SFU_insts	# of SFU insts. per warp	Instruction analyzer
#FP_insts	# of floating point insts. per warp	Instruction analyzer
#total_warps	Total number warps in a kernel	program's input
#active_SMs	# of active SMs	hardware specification
N	# of concurrently running warps on one SM	hardware performance counters
AMAT	Average memory access latency	hardware performance counters
avg_trans_warp	Average memory transactions per memory request	hardware performance counters
avg_inst_lat	Average instruction latency	hardware specification
miss_ratio	Cache miss ratio	hardware performance counters
size_of_data	The size of input data	source code
ILP	Inst.-level parallelism in one warp	Instruction analyzer
MLP	Memory-level parallelism in one warp	Instruction analyzer
MWP (Per SM)	Max #warps that can concurrently access memory	outcome of equations
CWP (Per SM)	# of warps executed during one mem. period plus one	outcome of equations
$\text{MWP}_{\text{peak\_bw}}$ (Per SM)	MWP under peak memory BW	equations
warp_size	# of threads per warp	32
Γ	Machine parameter for sync cost	64
Δ	Transaction departure delay	Measured machine parameter
DRAM_lat	Baseline DRAM access latency	Measured machine parameter
FP_lat	FP instruction latency	Hardware specification
hit_lat	Cache hit latency	Hardware specification
SIMD_width	# of scalar processors (SPs) per SM	Hardware specification
SFU_width	# of special function units (SFUs) per SM	Hardware specification

Table 4.1: Summary of input parameters used in equations.

4.1.2 Applying the Model to FMM

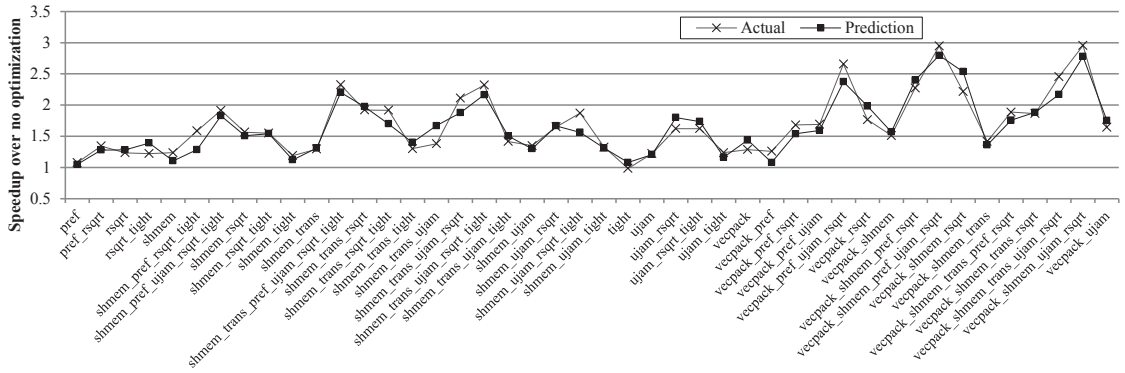


Figure 4.2: Speedup over the baseline of actual execution and model prediction on 44 different optimizations.

We apply this model to an implementation of the *fast multipole method* (FMM), which was described in Chapter 3. We apply the same set of optimizations that are described in Chapter 3.5.1. Figure 4.2 shows the speedup over the baseline kernel of actual execution and its prediction using the proposed model. The x-axis shows the code optimization space, where 44 optimization combinations are presented. The results show that, overall, the model closely estimates the speedup of different optimizations. It also shows that the analytical model successfully identifies the best performing optimization combination.

4.1.3 Performance Optimization Guide

Based on the presented performance model, we could identify performance bottleneck information and estimate the potential gains from reducing or eliminating these bottlenecks. It does so through four potential benefit metrics, whose impact can be visualized using a chart as illustrated by Figure 4.3. The x-axis shows the cost of memory operations and the y-axis shows the cost of computation. An application code is a point on this chart (here, point A). The sum of the x-axis and y-axis values is the execution cost, but because computation and memory costs can be overlapped, the final execution cost of T_{exec} (e.g., wallclock time) is a different point, A', shifted relative to A. The shift amount is denoted as T_{overlap}. A diagonal line through $y = x$ divides the chart into *compute bound* and *memory bound* zones, indicating whether an application is limited by computation or memory operations, respectively. From point A', the benefit chart shows how each of the four different potential benefit metrics moves the application execution time in this space.

A given algorithm may be further characterized by two additional values. The first is the ideal computation cost, which is generally the minimum time to execute all of the essential computational work (e.g., floating point operations), denoted T_{fp} in Figure 4.3. The second is the minimum time to move all data from the DRAM to the cores, denoted by $T_{\text{mem\_min}}$. When memory requests are prefetched or all memory service is hidden by other computation, we might hope to hide or perhaps eliminate all of the memory operation costs. Ideally, an algorithm designer or programmer could provide estimates or bounds on T_{fp} and

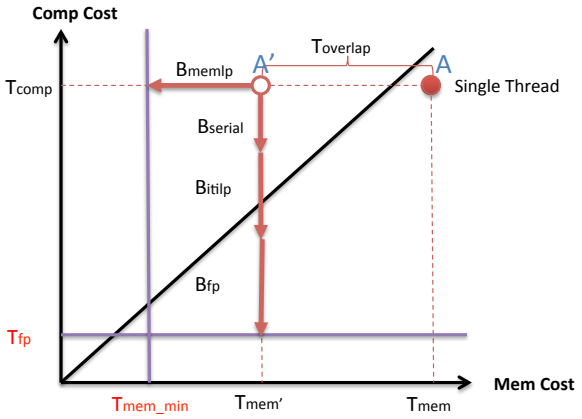


Figure 4.3: Potential performance benefits, illustrated.

$T_{\text{mem\_min}}$. However, when the information is not available, we could try to estimate T_{fp} from, say, the number of executed FP instructions in the kernel.[1]

Suppose that we have a kernel at point A' having different kinds of inefficiencies. Our computed benefit factors aim to quantify the degree of improvement possible through the elimination of these inefficiencies. We use four potential benefit metrics, summarized as follows.

- B_{itilp} indicates the potential benefit by increasing inter-thread instruction-level parallelism.
- B_{memlp} indicates the potential benefit by increasing memory-level parallelism.
- B_{fp} represents the potential benefit when we ideally remove the cost of inefficient computation. Unlike other benefits, we cannot achieve 100% of B_{fp} because a kernel must have some operations such as data movements.
- B_{serial} shows the amount of savings when we get rid of the overhead due to serialization effects such as synchronization and resource contention.

B_{itilp}, B_{fp}, and B_{serial} are related to the computation cost, while B_{memlp} is associated with the memory cost. These metrics are summarized in Table 4.2.

[1] In our evaluation we also use the number of FP operations to calculate T_{fp}.

Name	Description	Unit
T_{exec}	Final predicted execution time	cost
T_{comp}	Computation cost	cost
T_{mem}	Memory cost	cost
T_{overlap}	Overlapped cost due to multi-threading	cost
T'_{mem}	$T_{\text{mem}} - T_{\text{overlap}}$	cost
T_{fp}	Ideal T_{comp}	ideal cost
$T_{\text{mem\_min}}$	Ideal T_{mem}	ideal cost
B_{serial}	Benefits of removing serialization effects	benefit
B_{itilp}	Benefits of increasing inter-thread ILP	benefit
B_{memlp}	Benefits of increasing MLP	benefit
B_{fp}	Benefits of improving computing efficiency	benefit

Table 4.2: Summary of performance guidance metrics.

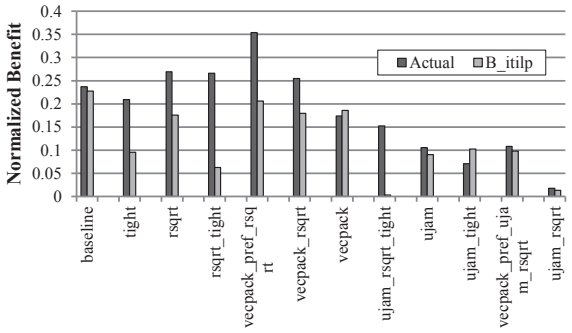


Figure 4.4: Actual performance benefit and B_{itilp} when $shmem$ is applied to each optimization in the x-axis.

Potential Benefit Prediction

To understand the performance optimization guide metrics, we first compute potential benefit metrics for the baseline, which are as follows: $B_{\text{serial}} = 0$, $B_{\text{memlp}} = 0$, $B_{\text{itilp}} = 6068$, and $B_{\text{fp}} = 9691$. Even from the baseline, the kernel is already limited by computation. Hence, techniques to reduce the cost of computation are critical.

Figure 4.4 shows actual performance benefits and B_{itilp} when the shared memory optimization ($shmem$) is applied on top of different combinations of optimizations. For example, $ujam$ indicates the performance delta between $ujam$ and $ujam + shmem$ optimizations. In the graph, both $Actual$ and B_{itilp} are normalized to the execution time before $shmem$

is applied. Using the shared memory improves both MLP and ILP. It increases the reuse of source points, which also increases ILP. Hence, the benefit of using the shared memory can be predicted using B_{itilp}, because $B_{\text{memlp}} = 0$. As shown in the figure, B_{itilp} predicts the actual performance benefit closely for most of the optimization combinations except $tight$ optimizations. The interaction between $shmem$ and $tight$ optimizations should be analyzed further.

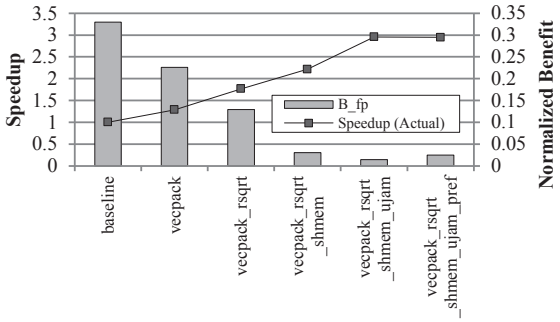


Figure 4.5: Performance speedup and potential benefits when applying optimizations to the baseline one by one.

Figure 4.5 represents the performance speedup and potential benefits normalized to the baseline execution time when optimizations are applied one by one.

For the baseline, a high B_{fp} value indicates that the kernel might have the potential to be improved. Hence, it is reasonable to try to optimize the baseline kernel. As we apply adequate optimization techniques that improve the kernel, the potential benefit decreases. When $vecpack$, $rsqrt$, $shmem$ and $ujam$ have been applied, the potential benefit metric indicates that the kernel may a have very small amount of inefficient computation and the potential performance gain through further optimizations might be limited. As shown in the last bar, $pref$ does not lead to a performance improvement. Rather, B_{fp} slightly increases due to the inefficient computation for prefetching code.

4.2 Other Performance Modeling Techniques and Tools

Beyond the model of the previous section, there have been a flurry of studies on GPGPU performance modeling. This section surveys the current landscape of metrics and tools that

are available for understanding GPGPU performance.

4.2.1 Limited Performance Visibility

Efficiently programming GPUs remains intellectually challenging and requires significant effort. The major challenge in programming GPUs is usually saturating the high-bandwidth computational processing elements by providing them with the required data in a timely manner. To achieve this goal, it is important for GPGPU applications to make effective use of the GPU memory hierarchy. Therefore, information about the interaction of an application with the memory system is necessary to optimize memory accesses and their performance.

Developers need to manually examine the GPGPU kernel source code and identify the sources of performance degradation such as long latency loads on the critical path of a tight loop. This kind of analysis is often done through guesswork and trial-and-error experiments.

Furthermore, multiple levels of concurrency and the high number of concurrent events, such as in flight memory operations, can make it hard for either the developer or even the optimizing compiler reason about performance. A highly multithreaded GPU acts like a complex, nonlinear system with tightly coupled dynamics; a microarchitectural event such as a memory load should be evaluated in a meaningful context with regard to other events from the co-existing threads.

Therefore, the performance evaluation process is very time consuming for even experienced application developers and much more difficult and tedious for less-informed programmers. With millions of units currently in use, GPUs have become the most widely deployed parallel systems. As more users experiment with GPUs and their use grows in all classes of computing systems, their cost will continue to decrease. This trend may in turn expand the GPGPU computing community and necessitate the development of techniques that systematically analyze the source code in the context of a highly multithreaded execution environment to replace manual investigation of the code.

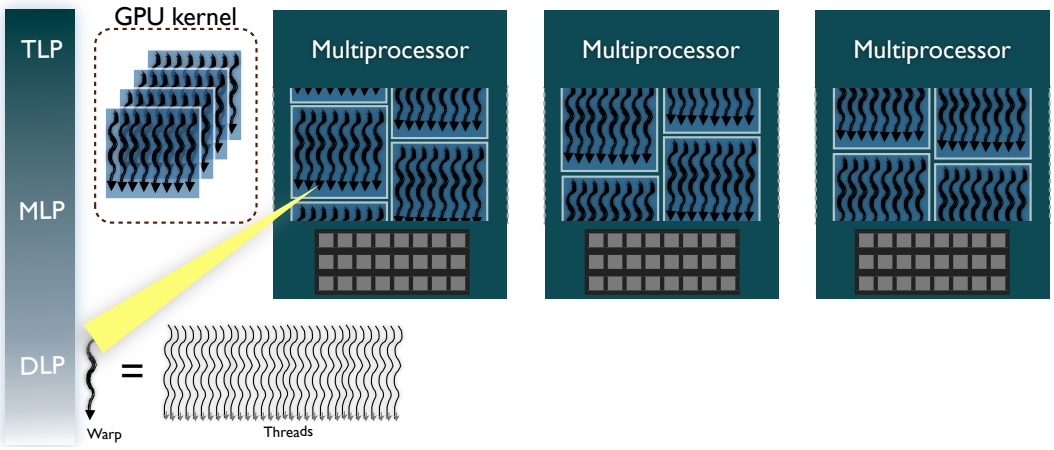


Figure 4.6: Multi-level parallelism in a GPGPU Kernel (a) Interleaving of group of threads within a block (warps) resembles the thread-level parallelism. (b) High number of in-flight memory operations exhibit memory-level parallelism (c) Data-level parallelism is exploited by operating on multiple memory banks simultaneously to fetch memory words through SIMD instructions. (d) At the thread level, instruction-level parallelism can partially covering intra-warp stalls.

To provide a formal framework to study this problem, Baghsorkhi et al. introduced the concept of *balanced GPGPU computation* [11]. This model represents a GPGPU computation using the computation carried by an average warp. In carrying out a computation the GPU is said to be balanced if

1. the computation core pipelines are not stalled due to data hazards (register dependencies or long latency global memory loads), structural hazards (shared memory bank conflicts), or control hazards (SIMD branch divergence);

2. the *computation time* equals the *memory time*. The former is the time to complete the computation assuming an ideal memory system, while the latter is the time needed to flow all data required by computation cores to and from memory.

One may view this framework as specializing the balance principles framework of Chapter 2 for GPGPUs with a refined processor model.

4.2.2 Work Flow Graphs

To identify bottlenecks that result in an imbalanced computation, Baghsorkhi et al. combine the amount of available concurrency in a kernel (TLP, DLP and ILP) – considering the

effect of coexisting threads – and the latencies of the SIMD pipeline and the memory system such that the interactive effects of different performance factors are preserved [11]. For this task, they introduced the *work flow graph* (WFG). The WFG is an extension of the control flow graph of a GPGPU kernel. Nodes in the WFG are a long-latency (global) memory operations, low-latency (scratch-pad) memory operations, barrier synchronizations, blocks of n continuous computational instructions, or synthetic entry and exit nodes. In addition to edges that correspond to the control flow graph (transition arcs), the WFG also contains data-dependence arcs that connect global memory loads to their corresponding uses.

The initial WFG abstracts the computational kernel independent of the underlying hardware. No weight is assigned to transition arcs initially. A transition arc only indicates that the destination node can be executed immediately after the source node. The WFG represents an average warp in a GPGPU kernel. In the case of a branch divergence, different threads may follow different control flow paths. Therefore, proper weight should be assigned to the diverging control flow paths to indicate the fraction of warps that execute each path. Similarly, if there are long-latency or low-latency memory accesses along these diverging paths, the number of memory transactions or bank conflict serialization delays should be adjusted according to the portion of warps and pattern of threads that issue the memory instructions.

To estimate the performance on a particular GPU, Baghsorkhi et al. proposed specializing the arcs in the WFG with information gathered through symbolic evaluation and runtime inspection of a GPGPU kernel based on hardware parameters such as memory bandwidth, read latency, memory coalescing rules, memory bank configuration, SIMD work granularity, SIMD engine width, and pipeline latency, among other factors. The resulting WFG binds the GPGPU kernel code profile to the GPU hardware configuration. After specializing the WFG for the specific hardware, the weight assigned to each WFG arc indicates the number of cycles required on average to execute the instruction(s) at the source node in the granularity of a warp.

4.2.3 Stochastic Memory Hierarchy Model

Graphics processors are optimized for throughput oriented workloads, which allowed early GPUs to omit traditional data caches. More recent GPUs have added small per-core L1 caches to capture inter-thread reuse, and larger unified L2 caches to exploit inter-core sharing. This is a significant step forward to better support a more diverse set of workloads and reduce some of the performance discrepancies that previously existed. Nevertheless, the performance of most GPGPU applications is strongly dependent on the efficient use of the memory subsystem. Current generation of GPUs provide a set of performance counters that collect statistics such as the overall number of misses for the L1 and the L2 caches, but the counter values cannot be monitored or sampled during a program execution. In addition, a time-based sampling can be built on top of the provided instrumentation interfaces to identify hotspots in a program. While a high sampling rate will likely distort the accuracy of the profile data, it is also difficult to achieve high-resolution performance information at low sampling rates in the presence of thousands of concurrently running threads. To overcome this problem, Baghsorkhi et al. presented a new software-based approach for monitoring the memory hierarchy performance in highly multithreaded general-purpose graphics processors [12].

The proposed analysis is based on memory traces collected for snapshots of an application execution. Memory traces are collected by instrumenting the GPU kernel at the source code level to record the memory addresses accessed during the kernel execution. The instrumentation framework is designed as a source-to-source transformation module. Static probes are inserted within the GPU kernel source code, as shown in Listing 4.1, to record the memory addresses read from and written to by active threads for a subset of thread blocks as the GPU kernel is executed.

To understand the performance of a GPU kernel with respect to the memory hierarchy, it is necessary to collect enough traces to capture both intra-thread and inter-thread interactions of memory operations. To capture inter-thread interferences the execution snapshot for trace collection is extended horizontally. They collect traces for thread blocks that are

Listing 4.1: Instrumented SpMV – device code

```
__global__ void SpMV(_sampling_device_buffer *
      _samples, float *x, const float *val, ...)
{
  _sampling_status _this_thread_status;
  _init_sampling_status(samples, &this_thread_status);

  tid = threadIdx.y;
  bid = blockIdx.y;
  t=0;
  myi = bid * BLOCKSIZE + tid;

  if ( myi < (numRows) ){

    _sample_mem_index(samples, &this_thread_status, 0,
        &(rowInd[myi]));
    lb = rowInd[myi];

    _sample_mem_index(samples, &this_thread_status, 2,
        &(rowInd[myi + 1]));
    ub = rowInd[myi+1];

    for (j=lb; j<ub; j++) {

      _sample_mem_index(samples, &this_thread_status, 4,
          &(indices[j]));
      ind = indices[j];

      _sample_mem_index(samples, &this_thread_status, 6,
          &(y[ind]));
      yval = y[ind];

      _sample_mem_index(samples, &this_thread_status, 8,
          &(val[j]));
      t += val[j] * yval;

    }
    _sample_mem_index(samples, &this_thread_status, 1,
        &(x[myi]));
    x[myi] = t;

  }

  _release_sampling_buffer(&_this_thread_status);
}
```

potentially scheduled close together on different cores (streaming multiprocessors). To account for intra-thread interactions the execution snapshot is extended such that large enough traces are collected within a single thread. If required, the execution snapshot is extended across the boundaries of multiple thread blocks that are scheduled back-to-back on a single core. With this approach, a subset of the memory traces is collected, but the subset is detailed enough to reflect locality and interactions within the group of concurrently running threads.

Collected traces exhibit precise intra-warp (intra-thread) ordering of memory references. But traces do not maintain any information about the relative order of memory references issued from different warps or thread blocks as their approach does not rely on the execution order of memory loads and stores when collecting traces. The rationale for not relying on the ordering during the execution of the instrumented GPU kernel is that adding static probes to the kernel source code:

1. changes the kernel resource usage (number of registers), which may consequently alter the number of concurrently active thread blocks on each streaming multiprocessor.

2. increases the number of inflight memory operations, which will distort the state of caches and the level of congestion in the memory hierarchy.

3. changes the instruction mix of the GPU kernel and introduces spurious synchronization or stall points.

As a result, instrumenting the kernel will distort the execution order of the memory operations. To resolve this problem, the order of memory requests arriving at each level of the memory hierarchy is reconstructed via a Monte Carlo method, an efficient sampling approach for systems with individual behaviors highly coupled together. Traces are driven into each level of the memory hierarchy in the cache simulator according to the randomly sampled ordering in each run according to the following steps:

1. Given a pool of memory requests waiting to be serviced at each level of the memory hierarchy:

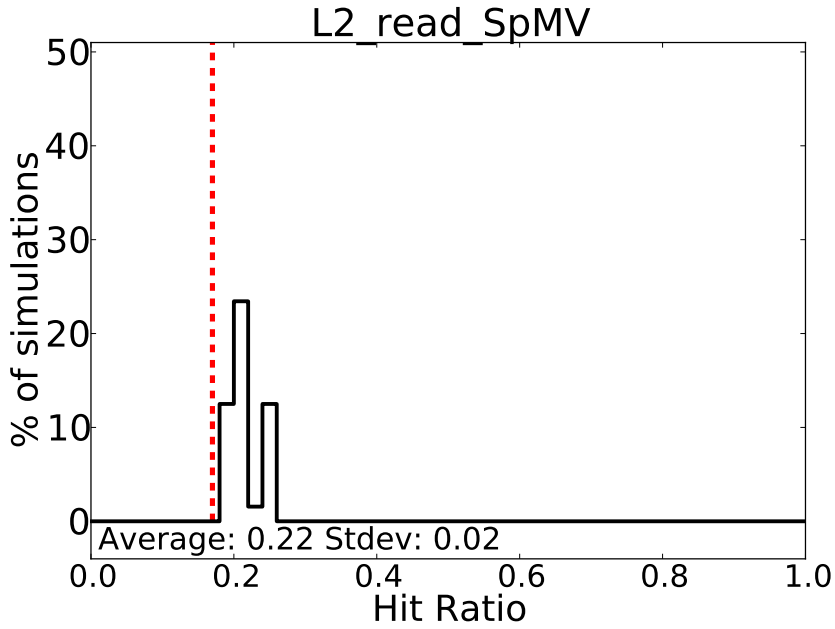


Figure 4.7: Probability distribution of the L2 Load Hit Ratio for Sparse Matrix Vector Multiplication Kernel

(a) Generate a valid random ordering from the pool of available memory requests.

(b) Drive traces to the current memory hierarchy module following the ordering derived in step (a).

(c) Obtain performance estimations for the current level and prepare the pool of memory request to be serviced by the next memory hierarchy level.

2. Repeat step 1 for a sufficiently large number of times.

3. Determine the probability distribution of results using histograms and summarize the confidence of the predictions.

The output of the above model is a probabilistic performance behavior of the memory system such as hit ratios for the first and second level caches. Figure 4.7 shows the L2 cache hit ratio distribution for the SpMV kernel computed through the above approach. If certain performance behaviors are most frequently observed – even with limited knowledge about the exact relative ordering of inter-thread memory requests – they are statistically sound representatives of the system performance. If different random orderings result in

noticeably different performance statistics (a wide spread histogram) then the predictions are not reliable. Otherwise, though Baghsorkhi et al. [12] do not follow the actual inter-thread ordering when driving traces into the simulator, they can set up an execution context similar and close enough to the actual one.

They also propagate the source code locations of memory loads and stores. As a result, they can produce precise and high resolution profile statistics for individual memory operations in the GPU kernel source code.

4.2.4 Roofline Model

Williams et al. proposed the roofline model, which is useful for visualizing of compute-bounded or memory-bounded multicore architectures [97].[2] The roofline sets an upper bound on the performance of a kernel, depending on the kernel's operational intensity. When operational intensity as a column hits the roof, either it hits the flat part of the roof, which means performance is compute bound, or performance is ultimately memory bound. Figure 4.8 shows an example of the roofline for different architectures. Equation (4.31) represents the model.

$$\texttt{Attainable Gflop/s} = min(\texttt{Peak Gflop/s}, \texttt{Stream BW} \times \texttt{actual flop : byte ratio}) \tag{4.31}$$

The roofline is not limited to just characterizing peak. One can map specific architectural features to other roofs and ridges below the peak roofline, and thereby visually understand the impact of changing intensity and adopting particular architectural features on performance.

[2]The first primitive "roofline diagram" drawn in the context of performance analysis appears as Figure 2 of Hockney and Curington [49].

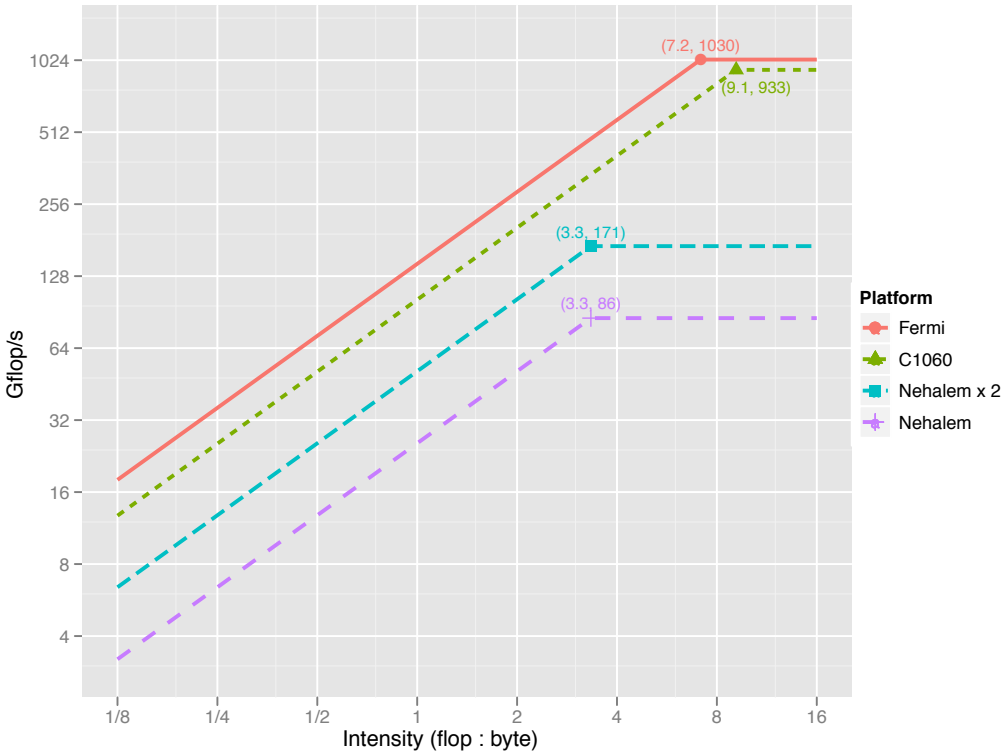


Figure 4.8: An example of roofline graph

4.2.5 Profiling and Performance Analysis of CUDA Workloads Using Ocelot [33]

GPU Ocelot is an open-source dynamic JIT compilation framework for GPU compute applications targeting a range of GPU and non-GPU execution targets. Ocelot supports unmodified CUDA applications through its custom implementation of the CUDA Runtime API. Internally, Ocelot parses CUDA kernels structured as PTX modules. PTX, NVIDIA's Parallel Thread Execution (PTX) virtual instruction set, is used to provide a device-independent program representation that captures the explicitly parallel semantics of CUDA's single instruction multiple thread (SIMT) execution model. Ocelot supports several backend execution targets—a PTX emulator, NVIDIA GPUs, AMD GPUs, and a translator to LLVM for efficient execution of GPU kernels on multicore CPUs. Device

portability is illustrated in Figure 4.9. By executing CUDA applications via GPU Ocelot, these device backends may be leveraged to provide enhanced analysis, profiling, and simulation of CUDA workloads to understand performance characteristics and bottlenecks as well as to apply optimizations for improved performance on native GPU execution targets.

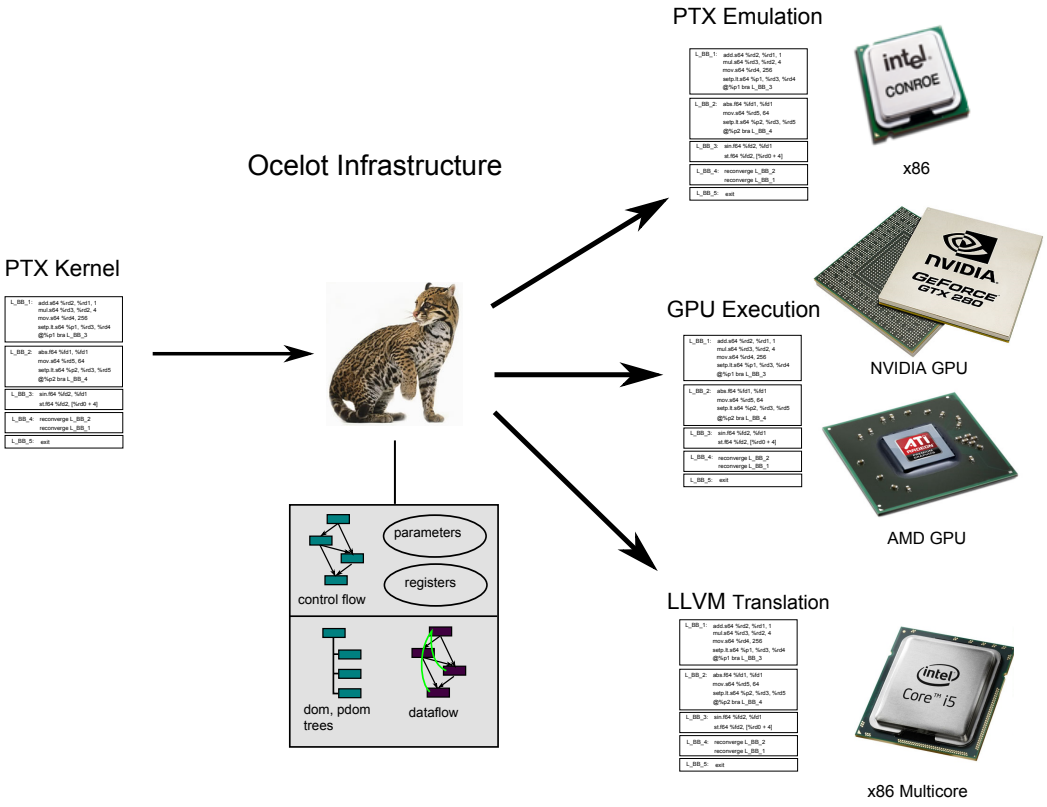


Figure 4.9: GPU Ocelot dynamic compilation and execution infrastructure.

PTX Emulation and Trace Analysis GPU Ocelot's PTX emulator executes CUDA kernels at the PTX level and provides the complete architectural state of a GPU for each dynamically executed instruction. Event trace analyzers are written to process a stream of execution events to create user-defined trace generators which react to dynamic instruction traces as the program is executing enabling real-time workload characterization and correctness checks. Ocelot current has over a dozen event trace analyzers for that provide support for

memory access checks, race detection, an interactive debugger, and feedback for performing tuning.

The sequence in which trace generator event handlers is called for a given dynamic instruction stream is illustrated in Figure 4.10. All registered trace generators are invoked when a kernel is launched, enabling them to examine the table of memory allocations, analyze the kernel's internal representation, and observe parameter values. As each instruction is executed, trace generators are called before and after the emulator updates its architectural state. When the kernel has exited, trace generators are again invoked to perform a final analysis and possibly write results to external data stores.

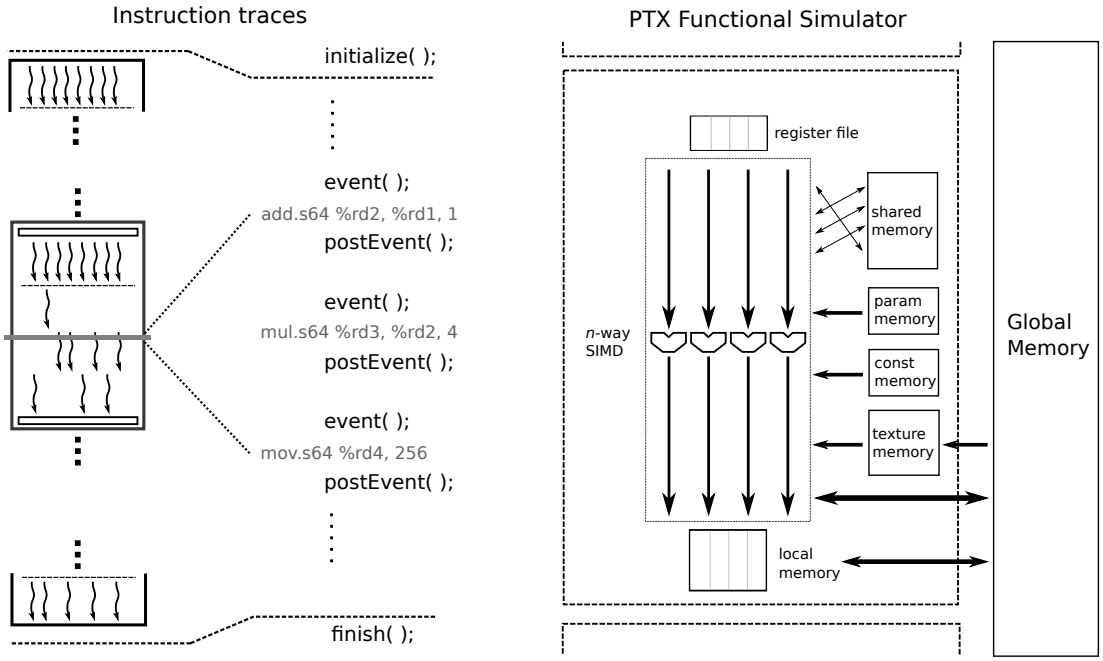


Figure 4.10: PTX Emulator trace generation facilities with abstract machine model.

GPU Ocelot enables the creation and application of several analysis tools. They profile kernel execution and provide visual feedback in the form of a heat map illustrating which part of the program dominates execution time. Statistics related to code within those regions provide an indication to the programmer of what the potential bottlenecks are. They may also assess the ratio of computation to memory transfer and determine whether kernels are definitely memory bound or definitely compute bound.

One common purpose for workload characterization is to identify compute intensive regions within a kernel and understand potential causes for inefficient execution. Several metrics of interest include SIMD utilization, memory efficiency, bank conflicts, and low compute instruction density. Measuring application behaviors provides the programmer feedback for performing optimizations. They may, for instance, restructure control behaviors of their kernel to improve SIMD utilization, reduce the number of bank conflicts to shared memory by skewing static memory allocations, or modifying thread access patterns. Compute intensity may be increased through other optimizations such as loop unrolling, pointer arithmetic, and barrier elision. The effects of these optimizations on dynamic instruction counts may be precisely measured and visualized, as presented in Figure 4.11.

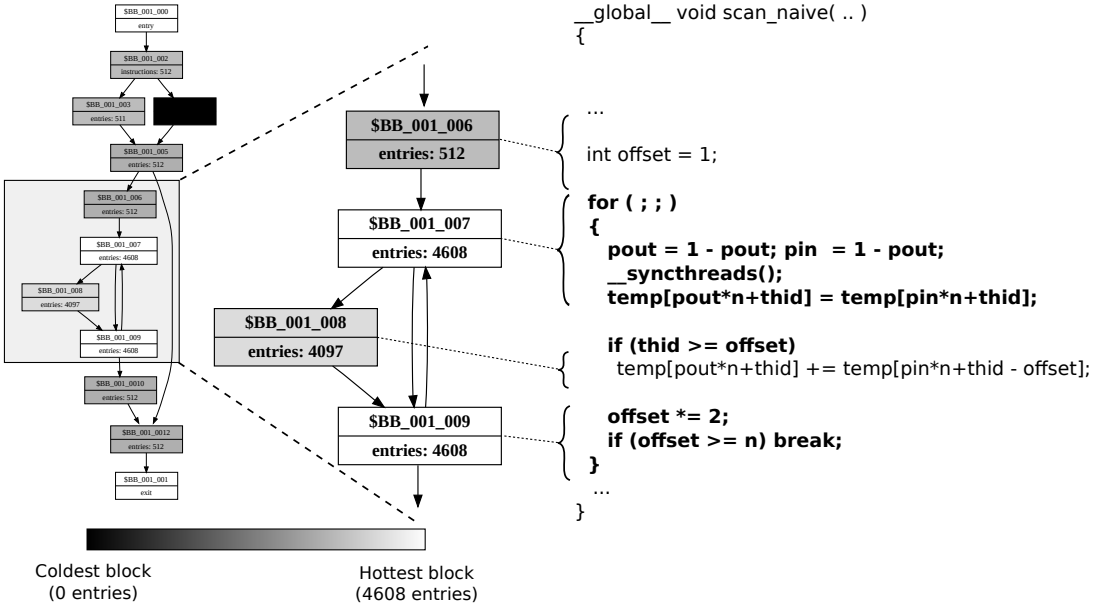


Figure 4.11: Hot region profile of CUDA SDK Scan application.

Instrumentation GPU Ocelot also defines an interface for implementing PTX transformation passes capable of inserting instrumentation code into kernels before JIT compiles them for execution on GPUs or other Ocelot device backends. Additionally, callbacks may be registered with Ocelot to construct, initialize, and analyze data structures associated with the transformation pass. This enables online dynamic instrumentation of GPU kernels

offering considerable speedup over Ocelot's emulator when evaluating the same analysis metric. Constructing transformations using Ocelot's PTX IR can be cumbersome, so a custom C-to-PTX compilation tool was developed to simplify the process. This relies on enumerated types and labels in C code to inform the compiler about where in the kernel to place the instrumentation probe. For example, an instrumentation code block could be inserted into every basic block of the original kernel or only at loop headers. Built-in functions enable divergence-free reductions across shared memory buffers, so behaviors related to the warp-based SIMT execution model may be observed and measured.

Remarks GPU Ocelot is available under the new BSD open-source license from its project site: `http://code.google.com/p/gpuocelot/`. Its PTX emulator device backend provides very detailed instruction and memory traces of CUDA workloads, which may be used to directly profile applications or drive other analysis tools such as trace-driven timing simulators. Several examples of correctness validation tools integrated within Ocelot and enabled by default detect synchronization and out-of-bounds errors within kernels, and other tools provide detailed profiling results. PTX instrumentation enables similar capabilities while executing on native GPU devices without modifying the original application. Ocelot's complete implementation of the CUDA Runtime API, rich set of PTX analysis passes, and kernel transformation pass manager offer a powerful platform for developing additional profiling and analysis tools for GPU compute workloads.

4.2.6 Other GPGPU Performance Modeling Techniques

Zhang and Owens presented a performance model in which they measured the execution time spent on the instruction pipeline, shared memory, and global memory to find the bottlenecks [106]. They also target bottleneck identification, but their method does not directly estimate performance benefits.

There has been a rich body of work on optimizations and tuning of GPGPU applications [24, 34, 68, 76, 84]. Ryoo et al. [84] introduced two metrics to prune optimization

space by calculating the utilization and efficiency of GPGPU applications. Choi et al. proposed a performance model for a sparse matrix-vector multiply (SpMV) kernel for the purpose of autotuning [24].

Performance Analysis Tools for CUDA

Plenty of tools are available to analyze the performance of CUDA applications. GPUs provide more and more hardware performance counters as we speak.

Kim and Shrivastava [58] presented a tool that can be used to analyze the memory access patterns of a CUDA program. They model the major memory effects such as memory coalescing and bank conflict. This is compile-time analysis, lacking dynamic information. Meng et al. [67] proposed a GPGPU performance projection framework. Given CPU code skeletons, the framework predicts the cost and benefit of GPGPU acceleration.

There are also GPU simulators that can be used for performance analysis. Bakhoda et al. [13] implemented a GPU simulator (GPGPU-Sim) and analyzed the performance of CUDA applications using the simulation output. A G80 functional simulator called *Barra* by Collange et al. [27] can execute NVIDIA CUBIN files while collecting statistics. GPGPU-Sim [13] has been widely used to study CUDA workload characteristics. Recently a number of CPU+GPU simulators have become available. Multi2sim [90], FusionSim [104], MV5 [45], and MacSim [1]. These are all cycle-level simulator, which can provide details of performance and even power behavior.

4.2.7 Performance Analysis Tools for OpenCL

In OpenCL, a couple of tools such as ATI Stream Profiler [77], NVIDIA's Parallel Nsight [71] and Visual Profiler provide statistics of OpenCL programs. The recently released Intel OpenCL SDK [52] also includes analysis tools for OpenCL kernels such as Graphics Performance Analyzer and VTune Amplifier XE 2011. Programmer can use these tools to store a kernel's execution data into a trace for off-line analysis and to analyze assembly

kernel code.

Bibliography

[1] MacSim. http://code.google.com/p/macsim/.

[2] Umut A Acar, Guy E Blelloch, and Robert D Blumofe. The data locality of work stealing. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures - SPAA '00*, pages 1–12, New York, New York, USA, July 2000. ACM Press.

[3] Advanced Micro Devices, Inc. AMD Brook+. http://ati.amd.com/technology/streamcomputing/AMD-Brookplus.pdf.

[4] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, August 1988.

[5] Randy Allen and Ken Kennedy. *Optimizing compilers for modern architectures*. Morgan-Kaufmann Publishers, San Francisco, CA, USA, 2002.

[6] AMD. Fusion. http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx.

[7] Gene M Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *Proc.˜AFIPS Joint Computer Conf.*, volume 30, pages 483–485, Atlantic City, NJ, USA, April 1967.

[8] Lars Arge, Michael T Goodrich, Michael Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures - SPAA '08*, page 197, New York, New York, USA, 2008. ACM Press.

[9] Nitin Arora, Ryan P. Russell, and Richard W. Vuduc. Fast sensitivity computations for numerical optimizations. In *Proc. AAS/AIAA Astrodynamics Specialist Conference*, AAS 09-435, Pittsburgh, PA, USA, August 2009.

[10] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, and Onur Mutlu. Staged memory scheduling: achieving high performance and scalability in heterogeneous systems. In *Proceedings of the 39th International Symposium on Computer Architecture*, ISCA '12, pages 416–427, Piscataway, NJ, USA, 2012. IEEE Press.

[11] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wenmei W. Hwu. An adaptive performance modeling tool for gpu architectures. In *PPoPP*, 2010.

[12] Sara S. Baghsorkhi, Isaac Gelado, Matthieu Delahaye, and Wen-mei W. Hwu. Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 23–34, New York, NY, USA, 2012. ACM.

[13] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163 –174, april 2009.

[14] G.E. Blelloch, R.A. Chowdhury, P.B. Gibbons, V Ramachandran, S Chen, and M Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, volume pp, pages 501–510. Society for Industrial and Applied Mathematics, 2008.

[15] Guy E Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, March 1996.

[16] Guy E Blelloch, Phillip B Gibbons, and Harsha Vardhan Simhadri. Low depth cache-oblivious algorithms. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures - SPAA '10*, page 189, New York, New York, USA, June 2010. ACM Press.

[17] W.J. Bouknight, S.A. Denenberg, D.E. McIntyre, J.M. Randall, A.H. Sameh, and D.L. Slotnick. The illiac iv system. *Proceedings of the IEEE*, 60(4):369 – 388, april 1972.

[18] George E.P. Box, J. Stuart Hunter, and William G. Hunter. *Statistics for Experimenters: {Design}, Innovation, and Discovery*. Wiley-Interscience, 2nd edition, 2005.

[19] Richard P. Brent. The Parallel Evaluation of General Arithmetic Expressions. *Journal of the ACM*, 21(2):201–206, April 1974.

[20] N. Brunie, S. Collange, and G. Diamos. Simultaneous branch and warp interweaving for sustained gpu performance. In *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, pages 49 –60, june 2012.

[21] David Callahan, John Cocke, and Ken Kennedy. Estimating interlock and improving balance for pipelined architectures. *Journal of Parallel and Distributed Computing*, 5(4):334–358, August 1988.

[22] Aparna Chandramowlishwaran, Kamesh Madduri, and Richard Vuduc. Diagnosis, tuning, and redesign for multicore performance: A case study of the fast multipole method. In *Proc. ACM/IEEE Conf. Supercomputing (SC)*, New Orleans, LA, USA, November 2010.

[23] Aparna Chandramowlishwaran, Samuel Williams, Leonid Oliker, Ilya Lashuk, George Biros, and Richard Vuduc. Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures. In *Proc. IEEE Int'l. Parallel and Distributed Processing Symp. (IPDPS)*, Atlanta, GA, USA, April 2010.

[24] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *PPoPP*, 2010.

[25] Rezaul Alam Chowdhury, Francesco Silvestri, Brandon Blakeley, and Vijaya Ramachandran. Oblivious algorithms for multicores and network of processors. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.

[26] George Chrysos. Intel$^{(R)}$ xeon phi^{TM} coprocessor (codename knights corner). http://www.slideshare.net/IntelXeon/under-the-armor-of-knights-corner-intel-mic-architecture-at-hotchips 2012.

[27] Sylvain Collange, Marc Daumas, David Defour, and David Parello. Barra: A parallel functional simulator for gpgpu. *Modeling, Analysis, and Simulation of Computer Systems, International Symposium on*, 0:351–360, 2010.

[28] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, 3rd edition, 2009.

[29] J.-L. Cruz, A. Gonzalez, M. Valero, and N.P. Topham. Multiple-banked register file architectures. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 316 –325, june 2000.

[30] Kenneth Czechowski, Casey Battaglino, Chris McClanahan, Aparna Chandramowlishwaran, and Richard Vuduc. Balance principles for algorithm-architecture co-design. In *Proc. USENIX Wkshp. Hot Topics in Parallelism (HotPar)*, Berkeley, CA, USA, May 2011.

[31] Wen-mei Hwu David Kirk. *Programming Massively Parallel Processors: A Hands-on Approach, Second Edition*. Morgan Kaufmann, 2012.

[32] Gregory Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. Simd re-convergence at thread frontiers. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 477–488, New York, NY, USA, 2011. ACM.

[33] Gregory Diamos, Andrew Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems. In *PACT-19*, 2010.

[34] Yuri Dotsenko, Sara S. Baghsorkhi, Brandon Lloyd, and Naga K. Govindaraju. Auto-tuning of fast fourier transform on graphics processors. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 257–266, New York, NY, USA, 2011. ACM.

[35] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine multicomputer. In *Proceedings of the 28th annual international symposium on Microarchitecture*, pages 146–156, 1995.

[36] Matteo Frigo, C.E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Sci-*

ence (Cat. No.99CB37039), pages 285–297, New York, NY, USA, October 1999. IEEE Comput. Soc.

[37] Matteo Frigo and Volker Strumpen. Cache oblivious stencil computations. *Proceedings of the 19th annual international conference on Supercomputing ICS 05*, 1(212):361, 2005.

[38] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 407–420, Washington, DC, USA, 2007. IEEE Computer Society.

[39] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation: Efficient mimd control flow on simd graphics hardware. *ACM Trans. Archit. Code Optim.*, 6(2):7:1–7:37, July 2009.

[40] W.W.L. Fung and T.M. Aamodt. Thread block compaction for efficient simt control flow. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 25 –36, feb. 2011.

[41] Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindholm, and Kevin Skadron. Energy-efficient mechanisms for managing thread context in throughput processors. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 235–246, New York, NY, USA, 2011. ACM.

[42] Alexander G Gray and Andrew W Moore. 'N-Body' problems in statistical learning. In *Proc.˜Advances in Neural Information Processing Systems (NIPS)*, Vancouver, British Columbia, Canada, December 2000.

[43] L GREENGARD and V ROKHLIN. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2):325–348, December 1987.

[44] Leslie Greengard and Vladimir Rokhlin. A new version of the Fast Multipole Method for the Laplace equation in three dimensions. *Acta Numerica*, 6:229, November 2008.

[45] LAVA group. The MV5 simulator. https://sites.google.com/site/mv5sim/benchmarks.

[46] T.D. Han and T.S. Abdelrahman.

[47] Mark D Hill and Michael R Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7):33–38, July 2008.

[48] W. Daniel Hillis. Balancing a Design. *IEEE Spectrum*, 1987.

[49] Roger W. Hockney and Ian J. Curington. f1/2: A parameter to characterize memory and communication bottlenecks. *Parallel Computing*, 10(3):277–286, May 1989.

[50] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *ISCA*, 2009.

[51] Wenmei W. Hwu and John Stone. A programmers view of the new GPU computing

capabilities in the Fermi architecture and CUDA 3.0. White paper, University of Illinois, 2009.

[52] Intel Corporation. Intel OpenCL SDK. http://software.intel.com/en-us/articles/intel-opencl-sdk/.

[53] Dror Irony, Sivan Toledo, and Alexander Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput.*, 64:1017–1026, 2004.

[54] Joseph JàJà. *Introduction to parallel algorithms*. Addison-Wesley, 1992.

[55] Min Kyu Jeong, Chander Sudanthi, Nigel Paver, and Mattan Erez. A qos-aware memory controller for dynamically balancing gpu and cpu bandwidth use in an mpsoc. In *the Proceedings of the 2012 Design Automation Conference (DAC12)*, June 2012.

[56] Hong Jia-Wei and H T Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing - STOC '81*, pages 326–333, New York, New York, USA, May 1981. ACM Press.

[57] David Kanter. Inside fermi: Nvidias hpc push. http://www.realworldtech.com/fermi/6/.

[58] Yooseong Kim and Aviral Shrivastava. Cumapz: A tool to analyze memory access patterns in cuda. In *DAC '11: Proc. of the 48th conference on Design automation*, June 2011.

[59] H T Kung. Memory requirements for balanced computer architectures. In *Proceedings of the ACM Int'l. Symp. Computer Architecture (ISCA)*, Tokyo, Japan, 1986.

[60] Ilya Lashuk, Aparna Chandramowlishwaran, Harper Langston, Tuan-Anh Nguyen, Rahul Sampath, Aashay Shringarpure, Richard Vuduc, Lexing Ying, Denis Zorin, and George Biros. A massively parallel adaptive fast multipole method on heterogeneous architectures. In *Proc. ACM/IEEE Conf. Supercomputing (SC)*, Portland, OR, USA, November 2009.

[61] Jaekyu Lee and Hyesoon Kim. TLP aware cache management schemes in a cpu-gpu heterogeneous architecture. In *HPCA-18*, 2012.

[62] Adam Levinthal and Thomas Porter. Chap - a simd graphics processor. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, pages 77–82, New York, NY, USA, 1984. ACM.

[63] Samuel Liu, John Erik Lindholm, Ming Y Siu, Brett W. Coon, and Stuart F. Oberman. Operand collector architecture. U.S. Patent Number 7,834,881, 2010.

[64] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a c-like language, 2003.

[65] John McCalpin. Memory Bandwidth and Machine Balance in High Performance Computers. *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter*, December 1995.

[66] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '02, pages 57–68, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[67] Jiayuan Meng, Vitali Morozov, Kalyan Kumaran, Venkatram Vishwanath, and Thomas Uram. Grophecy: Gpu performance projection from cpu code skeletons. In *SC'11*, February 2011.

[68] Jiayuan Meng and Kevin Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In *ICS*, 2009.

[69] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *ISCA-32*, 2010.

[70] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. Improving gpu performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 308–317, New York, NY, USA, 2011. ACM.

[71] NVIDIA. NVIDIA Parallel Nsight. http://developer.nvidia.com.

[72] NVIDIA Corporation. *CUDA Programming Guide, V4.0*.

[73] NVIDIA Corporation. CUDA Toolkit, 2012. Version 4.2 as of Sep. 2012, http://developer.nvidia.com/cuda/cuda-downloads.

[74] CORPORATE OpenGL Architecture ReviewBoard. *OpenGL reference manual: the official reference document for OpenGL, release 1*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1992.

[75] Matt Pharr and Randima Fernando. *GPU Gems 2*. Addison-Wesley Professional, 2005.

[76] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *SC '10*, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[77] Budirijanto Purnomo, Norman Rubin, and Michael Houston. Ati stream profiler: a tool to optimize an opencl kernel on ati radeon gpus. In *ACM SIGGRAPH 2010 Posters*, SIGGRAPH '10, pages 54:1–54:1, New York, NY, USA, 2010. ACM.

[78] Abtin Rahimian, Ilya Lashuk, Aparna Chandramowlishwaran, Dhairya Malhotra, Logan Moon, Rahul Sampath, Aashay Shringarpure, Shravan Veerapaneni, Jeffrey Vetter, Richard Vuduc, Denis Zorin, and George Biros. Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures. In *Proc. ACM/IEEE Conf. Supercomputing (SC)*, New Orleans, LA, USA, November 2010.

[79] Minsoo Rhu and M. Erez. Capri: Prediction of compaction-adequacy for handling control-divergence in gpgpu architectures. In *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, 2012.

[80] Michael D. Root and James R. Boer. *Directx Complete*. McGraw-Hill, Inc., New York, NY, USA, 1st edition, 1998.

[81] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, January 1978.

[82] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2008.

[83] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, and Wen mei W. Who. Program optimization study on a 128-core gpu. In *GPGPU-1*, 2007.

[84] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen mei W. Hwu. Program optimization space pruning for a multithreaded gpu. In *CGO-6*, pages 195–204, 2008.

[85] Jae Woong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. "gpuperf: A performance analysis framework for identifying performance benefits in gpgpu applications". In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallal Programming (PPoPP)*, 2012.

[86] Burton J. Smith. Readings in computer architecture. chapter Architecture and applications of the HEP mulitprocessor computer system, pages 342–349. Morgan Kaufmann Publishers Inc., 2000.

[87] Allan Snavely, Larry Carter, Jay Boisseau, Amit Majumdar, Kang Su Gatlin, Nick Mitchell, John Feo, and Brian Koblenz. Multi-processor performance on the Tera MTA. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–8, 1998.

[88] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 325–335, New York, NY, USA, 2006. ACM.

[89] James E. Thornton. *Design of a Computer: The Control Data 6600*. Foresman Press, 1970.

[90] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2012.

[91] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.

[92] Leslie G Valiant. A bridging model for multi-core computing. In *Proceedings of the European Symposium on Algorithms (ESA)*, volume LNCS 5193, pages 13–28, Universität Karlsruhe, Germany, September 2008. Springer Berlin / Heidelberg.

[93] Jeffrey Scott Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001.

[94] Jeffrey Scott Vitter. Algorithms and data structures for external memory. In *Foundation and Trends in Theoretical Computer Science*, volume 2, pages 305–474. now Publishers Inc., Hanover, MA, USA, 2006.

[95] Vasily Volkov. Better performance at lower occupancy. http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf.

[96] Vasily Volkov and James W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press.

[97] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65, April 2009.

[98] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 235 –246, march 2010.

[99] Dong Hyuk Woo and Hsien-Hsin S Lee. Extending Amdahl's Law for energy-efficient computing in the many-core era. *IEEE Computer*, 41(12):24–31, December 2008.

[100] Sven Woop, Jörg Schmittler, and Philipp Slusallek. Rpu: a programmable ray processing unit for realtime ray tracing. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 434–444, New York, NY, USA, 2005. ACM.

[101] Lexing Ying, George Biros, and Denis Zorin. A kernel-independent adaptive fast multipole algorithm in two and three dimensions. *Journal of Computational Physics*, 196(2):591–626, May 2004.

[102] Kamen Yotov, Tom Roeder, Keshav Pingali, John Gunnels, and Fred Gustavson. An experimental comparison of cache-oblivious and cache-conscious programs. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures - SPAA '07*, page 93, New York, New York, USA, 2007. ACM Press.

[103] Georgia L. Yuan, Ali Bakhoda, and Tor M. Aamodt. Complexity effective memory access scheduling for many-core accelerator architectures. In *MICRO-42*, pages 34–44, New York, NY, USA, 2009. ACM.

[104] Vitaly Zakharenko. Fusionsim simulator. www.fusionsim.ca.

[105] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 369–380, New York, NY, USA, 2011. ACM.

[106] Yao Zhang and John D. Owens. A quantitative performance analysis model for GPU architectures. In *HPCA*, 2011.