Universiteit Utrecht

Utrecht University

GPU performance prediction using parametrized models

Andreas Resios

A thesis submited in partial fulfillment of the requirements
for the degree of Master of Science

Supervisor:
Utrecht University Prof. Dr. S. Doaitse Swierstra
Daily supervisors:
Utrecht University Dr. Jurriaan Hage
Vector Fabrics Drs. Stefan Holdermans

July 2011

Contents

1 Introduction

1.1 Motivation

Compilation on modern architectures has become an increasingly difficult challenge with the evolution of computers and computing needs. In particular, programmers expect the compiler to produce *optimized* code for a variety of hardware, making the most of their theoretical performance.

For years this was not a problem because hardware vendors consistently delivered increases in clock rates and instruction-level parallelism, so that single-threaded programs achieved speedup on newer processors without any modification.

Nowadays to increase performance and overcome physical limitations, the hardware industry favours multi-core CPUs and massively parallel hardware accelerators (GPUs, FPGAs), and software has to be written explicitly in a multi-threaded or multi-process manner to gain performance.

Thus, the performance problem has shifted from hardware designers to compiler writers and software developers who now have to perform parallelization. Such a transformation involves identifying and mapping independent data and computation to a complex hierarchy of memory, computing, and interconnection resources.

When performing parallelization it is important to take into account the overhead introduced by communication, thread spawning, and synchronization. If the overhead is high the introduced optimization can lead to a performance loss.

Thus, an important question in this process is to evaluate whether the optimization brings any performance improvements. The answer is usually computed using a performance model which is an abstraction of the target hardware [29, 30].

Our research addresses this problem in the context of parallelizing sequential programs to GPU platforms. The main result is a GPU performance model for data-parallel programs which predicts the execution time and identifies bottlenecks of GPU programs. During the thesis we will present the factors which influence GPU performance and show how our model takes them into account.

We validated our model in the context of a production ready analysis tool *vfEmbedded* [33] which combines static and dynamic analyses to parallelize C code for heterogeneous platforms. Since the tool has an interactive compilation work-flow, our model not only estimates execution time but also computes several metrics which help users decide if their program is worth porting to the GPU.

1.2 Research question and goals

The main research question that we aim to answer can be defined using the following questions

> Given a way to port a sequential program to the GPU:
>
> - how fast will it run on the GPU?
> - are there any performance issues?
> - how much of the peak performance does it achieve?

In order to answer these questions, we must investigate modern GPUs and develop a model which captures their main performance factors. Because of the way GPU architectures have evolved, this poses many challenges which we need to address.

First of all, GPUs are optimized to execute data-parallel programs in which the workload is evenly split between threads. Detecting that a program is data-parallel is only half the story. The other important part, involves how individual threads are distributed. This aspect is usually abstracted as much as possible by the GPU programming model and users are instructed to perform experiments in order to see what is the best configuration. Since mapping influences performance, we plan to help the users by suggesting a suitable mapping computed using metrics which characterize the factors involved.

Second, in order to make predictions for a GPU we must identify and study the factors which influence performance. That is, we must be able to estimate the time of each operation of a GPU program. In order to do this we must measure the timings of operations under various scenarios and calibrate our model based on the experiments. In this case the word *scenario* refers to a GPU performance factor which is determined by the hardware architecture and influences the execution time of operations.

Finally, we need to take into account that the GPU programming model assumes a heterogeneous environment. This implies communication between the host (part which runs on the CPU) and the device (part executed on the GPU) in the form of data transfers. It is important to model the overhead of these transfers when predicting performance. This is because the transfers are performed over a bus which is slow when compared to the CPU and the GPU. Thus it is important to detect the cases where the overhead introduced by parallelization is higher than the sequential execution.

Based on the aforementioned challenges our main goal is to *create a GPU performance estimation framework* which not only estimates execution time, but also helps users identify potential GPU bottlenecks. Our main sub-goals are as follows:

Accurate GPU analytical model. The main part of our research involves the development of an accurate abstract cost model for the GPU. The GPU architecture poses many challenges so it is important to balance the precision of the model with the amount of computation required to generate it. In order to calibrate our model we need to develop a set of benchmarks which compute the required parameters for a target GPU.

Integration with a production ready analysis tool. Our research is performed in the context of a production ready analysis tool, *vfEmbedded* [33], which parallelizes sequential C code. Since the analysis tool uses interactive compilation, we do not aim at cycle-accurate estimations, but focus on predicting relative speedup and identifying potential performance bottlenecks. In order to compute the parameters of the model we build on top of the existing infrastructure which provides both static and dynamic analyses.

Modular performance prediction platform. From a software engineering point of view, we aim to develop a modular performance prediction platform composed of interchangeable components. That is, it is possible to change modules which simulate GPU components (e.g., scheduling policies) in order to experiment with different scenarios. Also the developed set of benchmarks should be applicable to a range of GPUs.

1.3 Thesis structure

Our thesis is structured in three main parts in which we present our results using a top-down approach.

The first part deals with the choice of hardware architecture and the experiments which reveal performance factors. More specifically, Section 2.2 introduces the CUDA platform, our target GPU architecture, by explaining the programming model, the hardware architecture used in our experiments, and the execution model. Furthermore, Section 2.3 gives an overview of the performance prediction techniques which we will employ and discusses the role that program analysis plays. In Section 3 we explain the factors which influence GPU performance and present the experiments which measure their impact.

The second part presents the model and the implementation. Section 4 presents an overview of the model and describes how it takes into account the identified performance factors. The implementation and instantiation of the model is described in Section 5. The program analysis infrastructure from which we compute values for the model parameters is described in Section 6.

The last part contains the validation methodology (Section 7) and a comparison between our approach and related work (Section 8). Section 9 presents our conclusions and future work.

2
Context

The context of our research is defined by the following fields: *performance prediction*, *GPU architecture* and *program analysis*. In order to understand our research question and goals we give a short introduction to each field. Furthermore, this section serves as a foundation for the notions needed in later chapters. It contains a description of our target architecture which introduces concepts that will be used in the model.

We start by describing how GPUs have evolved into programmable coprocessors and explain their architecture and programming model. To better understand how performance is estimated, we give an introduction to the current approaches in Section 2.3. Section 2.4 introduces concepts from program analysis which are used to compute the parameters of our model.

2.1 Evolution of GPUs

In recent years, graphics processing units (GPUs) have evolved into general-purpose processors which can perform a wide range of computations that were traditionally performed on the CPU.

This trend has led to the creation of a new research direction, general-purpose GPU (GPGPU) programming, which focuses on performing efficient computations using GPUs. In order to understand the concepts involved in this field, we present how the GPUs have evolved into parallel architectures and what programming models they support.

Traditionally the purpose of the GPU was to render images. Typically, it would receive as input a list of geometric primitives (composed of vertices) which were shaded and mapped onto the screen in the form of a picture using a canonical pipeline. The pipeline was initially fixed and contained the following stages:

- Vertex operations
- Primitive assembly
- Rasterization
- Fragment operations
- Composition

Since each vertex, respectively pixel, can be computed independently of each other, these operations are performed in parallel by dedicated hardware. From a computational point of view, this is the main source of parallelism in a GPU.

The first step towards general-purpose computation on the GPU was the evolution of the fixed-function pipeline into a programmable one. The key idea was replacing the fixed-function vertex and fragment operations with user-specified programs called shaders. Over the years, the shaders became increasingly more complex by providing less restrictive limits on their size and resource consumption, by utilizing fully featured instruction sets, and by allowing more flexible control-flow operations.

The next important milestone in GPU evolution was the development of the *unified shader architecture*. Traditionally the vertex and fragment shaders had different instruction sets that where processed by separate hardware units. As their functionality converged, the vertex and fragment processing hardware were combined into a unified programming unit.

As a result GPU architectures have migrated from strict pipelined task-parallel architectures to ones that are built around sets of unified data-parallel programmable units.

2.2 CUDA platform

In order to make performance predictions we must first understand how we write GPU programs and how they are executed.

Motivated by benchmarks and literature study we choose to target the *Compute Unified Device Architecture* (CUDA) platform developed by NVIDIA. As a representative of this architecture we use version 2.1 codenamed *Fermi*. The main sources of architectural information are [24, 25, 26].

In what follows, we first introduce the CUDA programming model which explains how we can develop GPU programs. The role of the programming model conflicts with our goal of performance prediction, because it abstracts factors and parameters which play a vital role for performance prediction. Thus we go one step further and explain the hardware architecture and the execution model.

2.2.1 Programming model

The complexity of the GPU architecture is hidden from the application developer by means of a heterogeneous programming model. The model allows the user to focus on the algorithm and abstracts as much as possible the GPU architecture. This is achieved through a small set of abstractions regarding thread grouping, memory hierarchy, and synchronization primitives.

The philosophy of the CUDA programming model is to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into smaller pieces that can be solved cooperatively in parallel by the threads of a block. Because blocks are independent of each other, the model achieves good scalability.

In what follows we describe the CUDA programming model using a small example. For a more in depth treatment of CUDA we recommend [26]. Since CUDA is based on ANSI C99 we also give the equivalent C program. The example represents a one-dimensional vector scaling operation. Figure 2.1 illustrates both the C and the CUDA version and emphasizes important concepts.

As in other programming models for the GPU (OpenCL, DirectCompute, ATI Stream), a CUDA program is composed from a *host part* which is executed on the CPU and a *device part* which is executed on the GPU. The device part is composed of several computational parts called *kernels* which can be invoked by the host. The programming model assumes that both the host and the device have their own separate memory spaces, and the host part manages the device memory through calls to the CUDA runtime.

At runtime the host program specifies the parameters for each kernel invocation. The required parameters

CPU

```
void scale(int n, float f, float* a){
  int i;
  for(i=0; i < n; i++)
    a[i] *= f;
}
```

CUDA

```
__global__ void cuda_scale(int n, float f, float* a){
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  if (i < n)
    a[i] *= f;
}
void foo(int n, float f, float* a){
 float* da = NULL;
 dim3 blockSize(8);
 dim3 gridSize(ceil(n / (float) 8);
 cudaMalloc(da, n * sizeof(float));
 cudaMemcpy(da, a, ...);
 cuda_scale<<<gridSize, blockSize>>>(n, f, da);
 cudaMemcpy(a, da, ...);
}
```

Host code | Device code | CUDA call | Kernel call

Figure 2.1: CUDA example program.

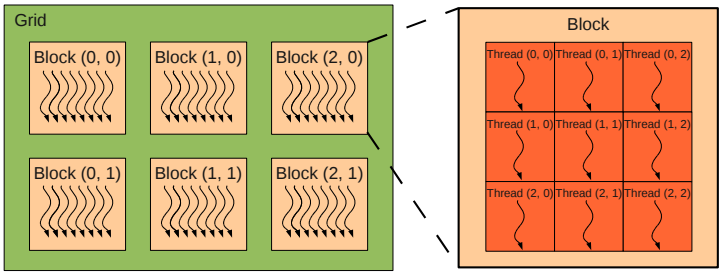


Figure 2.2: CUDA hierarchical thread grouping.

represent the grid and block dimensions. Optionally the user can also specify how much shared memory is available per-block.

The concept of a CUDA grid is illustrated in Figure 2.2. From a programmer point of view, a grid consists of a number of equally-shaped blocks where each block consists of a group of threads which execute the instructions specified in the kernel body. Depending on how threads are grouped, blocks and grids can be one-, two- or three-dimensional. This scheme provides a natural way to partition computation that involves 1D, 2D or 3D arrays.

Threads within the same block are assigned unique identifiers based on their coordinates. In a similar manner, the blocks which form the grid are assigned identifiers based on their coordinates. A thread is uniquely identified in a grid by combining its thread ID with the ID of the block to which it belongs.

Threads within a block can share data through shared memory and can perform barrier synchronization using special CUDA API calls.

CUDA threads can access data from different memory spaces as illustrated in Figure 2.3: each thread has

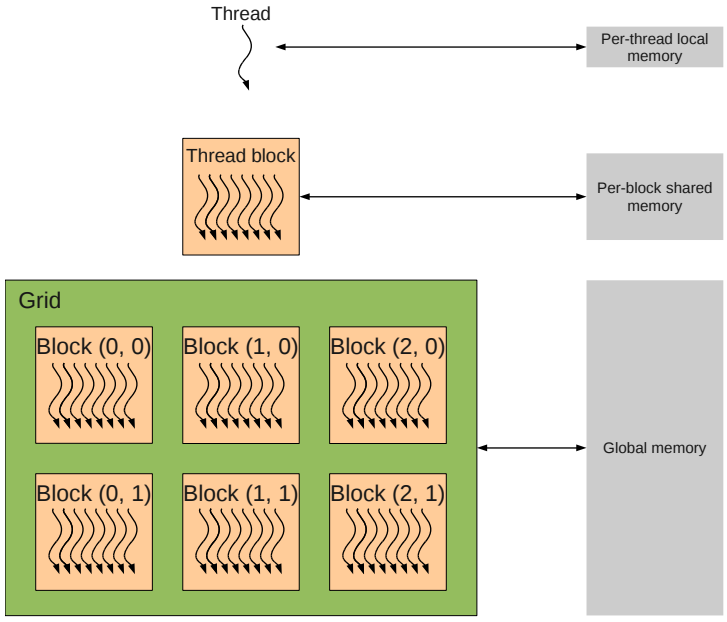


Figure 2.3: CUDA memory hierarchy.

private local memory, each block has shared memory which is accessible by all threads of the block, and all threads have access to global, constant, and texture memory which are optimized for different usage scenarios. Global memory is used for generic read-write operations, while constant and texture memory is read-only and benefits from special on-chip caches. Texture memory also offers different addressing modes and filtering for specific data formats.

During the same program, the global, constant, and texture memory spaces are persistent across different kernel calls, while the shared memory and the private memory change at each invocation.

2.2.2 Architecture

The main role of the programming model conflicts with our goal of performance prediction. This is because GPU performance is heavily influenced by the details abstracted away by the programming model. In what follows we present the architecture of the GF104 chip from NVIDIA, the GPU which we use to perform our research. We focus on presenting concepts which directly influence performance.

Figure 2.4 presents an overview of the GF104 GPU. We use colours to differentiate between GPU components: memory units are coloured blue, scheduling units are coloured orange, and compute units are coloured green.

The main part of the GPU is composed of 8 computational components which are called *streaming multiprocessors* (SMs). The GPU has three 64-bit DRAM modules connected using a 192-bit memory interface. A host interface connects the GPU to the CPU via the PCI-Express bus. The workload is expressed in terms of thread blocks which are distributed to the SMs by the GigaThread scheduler. Thread blocks are scheduled independently on each SM by using an on-chip scheduler.

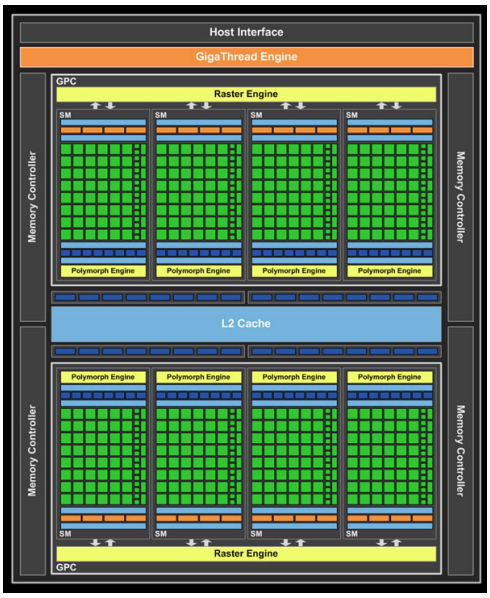


Figure 2.4: NVIDIA GF104 GPU.

The main components of SM are detailed in Figure 2.5. In each SM, computation is performed by 48 execution units called *streaming processors* (SP) or *CUDA cores*. Each CUDA core has a fully pipelined 32-bit integer arithmetic-logic unit (ALU) and 64-bit floating point unit (FPU).

The CUDA cores have access to a private set of registers allocated from a register file. The register file has a capacity of 128 kB divided into 32768 32-bit entries which can be accessed directly without performing any load or store operations.

The SMs access memory using 16 load-store units which can fetch data from cache or DRAM. In order to compute transcendental functions (sin, cos, etc.) each SM has 8 Special Function Units (SFU).

In order to hide memory latency, the GPU provides a complex memory hierarchy. The hierarchy is composed of per-SM L1 caches, a common L2 cache and the device memory.

Each SM features four types of caches: instruction, data, constant, and texture. Of these four caches, the most important one is the data cache. It has a total capacity of 64 kB which can be split into two partitions, one managed implicitly by the GPU (called the *L1 cache*) and the other managed explicitly by the application (called the *shared memory*). The data cache supports the following configurations: 16 kB L1 cache and 48 kB shared memory or 48 kB L1 cache and 16 kB shared memory. Since memory is shared between all the CUDA cores of a SM it can be used to perform intra-SM synchronization.

Besides its size, cache performance is also influenced by the following architectural details: line size, associativity and replacement policy. To understand this we give a brief introduction to how caches work. The data stored in the cache is organized in equal size blocks called *cache lines* which correspond to different memory locations. In addition to this, each line has a tag which contains the address of the memory location that is cached. When a request is issued, the cache is probed by comparing line tags with the requested address. If a match occurs, then the data is served from the cache (cache hit) otherwise it is loaded from memory (cache miss).

In order to make tag searching faster, caches restrict the number of lines to which an address can be stored. This is called the *cache associativity*. Associativity is a trade-off related to the replacement

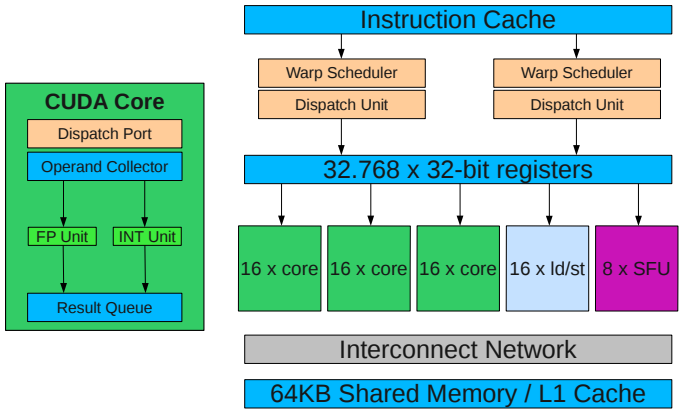


Figure 2.5: GF104 core (left) and symmetric multiprocessor (right).

policy. It represents the number of locations which must be searched in order to see if an address is in the cache. For example, if there are ten places to which the replacement policy could have mapped a memory location, then ten cache entries must be searched. Checking more places takes more time. On the other hand, caches with higher associativity suffer fewer misses, because an entry is less likely to be evicted.

In order to make room for the new entry on a cache miss, the cache has to evict one of the existing entries. The heuristic that it uses to choose the entry to evict is called the *replacement policy*. The fundamental problem with any replacement policy is that it must predict which existing cache entry is least likely to be used in the future. A commonly used policy is *least recently used* (LRU) which replaces the entry which has not been used recently.

Regarding the GF104 cache, the only other official information regarding the L1 cache is the line size, which is 128 bytes. There is no official information regarding the replacement policy or the associativity, but our experiments from Section 3 indicate a LRU replacement policy and high associativity.

The load, store, and texture requests of all the SM are serviced through a unified 384 kB L2 cache. This cache provides efficient, high speed data sharing across the GPU. As in the L1 case, NVIDIA only specifies the line size, which is 32 bytes.

The last layer of the GPU memory hierarchy is the device memory. This holds all the device data including texture and constant data. Although the constant data and textures are stored in device memory they have dedicated on-chip caches which minimize their latency.

2.2.3 Execution model

The previous section presented a static view on the GF104 hardware. In this section we focus on the dynamics of the platform in order to see how it executes GPU programs.

The CUDA execution model achieves performance by efficiently distributing the thread blocks of a kernel among SMs. As illustrated in Figure 2.6, at each kernel invocation the blocks of the grid are distributed to the SMs that have the necessary resources. The SMs execute in parallel the threads of multiple blocks. As soon as all the threads of a block terminate, a new block is launched.

Hardware resources play an important role because they limit the number of blocks which can be executed

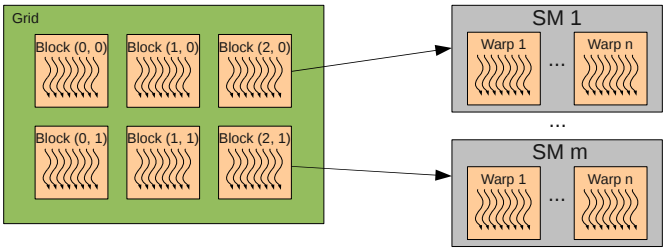


Figure 2.6: Block dispatching.

by the SM, thus limiting parallelism. Resources are used to keep track of the context of all executing warps. The context of a warp includes the program counters, the used registers, and the used shared memory. The registers are allocated from the register file which is partitioned among the warps, and shared memory is allocated for each of the thread blocks. Since these resources are limited, the number of blocks and warps that can be executed together on a SM is limited.

The main focus of the SMs is to maximize parallelism, either by executing instructions from different threads (thread-level parallelism – TLP), or by executing independent instructions within the same thread (instruction-level parallelism – ILP). We note that the GPU focuses more on TLP than ILP because even though instructions are pipelined, they are executed in order and without branch prediction or speculative execution.

In contrast to the CPU, the GPU can better exploit thread-level parallelism. This is because on the CPU, the operating system needs to swap the thread context in order to schedule another thread. As we previously mentioned, on the GPU the thread context is always available, so a context switch does not imply any data swapping.

SMs execute thread blocks in groups of 32 threads called *warps*. Each warp is scheduled for execution by a warp scheduler (also known as dispatcher), which issues the same instruction for all the warp's threads. When threads have diverged because of control-flow, all the divergent paths are executed serially with threads which are inactive in a path disabled.

In CUDA nomenclature, this type of execution is known as SIMT (Single Instruction, Multiple Threads). It is similar to SIMD (Single Instruction, Multiple Data) in that a single instruction is issued for multiple processing elements. The only difference is that SIMT makes the vectorization implicit for the programmer. That is, for the purposes of correctness, the programmer can essentially ignore the SIMT behaviour; however, performance improvements can be realized by taking care that the code seldom requires threads in a warp to diverge.

Superscalar execution. A SM dispatcher can issue instructions to a group of 16 CUDA cores, 16 load-store units or 8 SFUs (see Figure 2.5). This means that two dispatchers can issue instructions to 32 CUDA cores. Since the GF104 has 48 cores, it appears that 16 of them are idle. In order to avoid this scenario, the GPU employs a form of superscalar execution.

In a nutshell, superscalar execution is a method of extracting instruction-level parallelism (ILP) from a thread. If the next instruction in a thread is not dependent on the previous instruction, it can be issued to an execution unit for completion at the same time as the instruction preceding it.

In the GF104, this is done by the dispatchers which are responsible for analysing the next instruction of a warp to determine if that instruction is ILP-safe, i.e., is independent from executing instructions, and whether there is an execution unit available to handle it. NVIDIA calls this form of superscalar execution

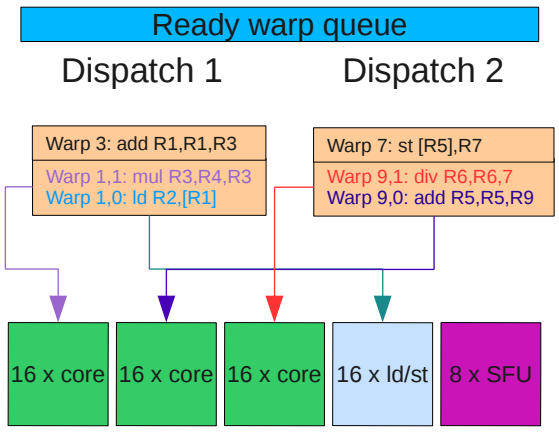


Figure 2.7: Example of superscalar execution on the GF104.

dual dispatch.

An example scenario is illustrated in Figure 2.7. The 48 CUDA cores are kept busy by three independent instructions from two warps: warp 1 executes the `mul` on 16 cores, and warp 9 executes `add` and `div` on the other 32 cores.

Memory instructions. In order to keep the execution units busy the GPU must be able to sustain a high transfer rate to and from memory. In order to achieve the necessary bandwidth, the GPU employs special techniques which reduce the amount of data that needs to be transferred between the global memory and the SMs. To understand how this is performed, we present how a memory operation is executed on the GPU. As is the case for compute operations, a memory operation is executed at the warp level. Depending on the type of memory which is accessed the operation is treated differently.

Global memory access. In the case of global memory, the process starts by computing the number of memory transactions which have to be issued in order to satisfy the requests of all threads which form the warp. Since access to global memory is expensive, the GPU tries to group requests from multiple threads in as few transactions as possible. The grouping depends on the size of the memory accessed by each thread and the distribution of addresses across the threads. This process is known as *global memory coalescing*.

Figure 2.8 illustrates the concept for the GF104. In this architecture transactions are 128 bytes long, which favours a 4 byte access pattern (i.e., $4 \times 32 = 128$). The first case depicts the ideal scenario, where each thread accesses 4 bytes and the accesses fit within a single line. This means that all the 128 transferred bytes are used. In the second case, the accesses are scattered across four lines, which means that from the 512 bytes transferred only 128 are used.

Transactions are not issued directly to global memory, but instead they first pass through a two-level cache hierarchy. If the requested data is in the cache it is serviced at the cache throughput, otherwise it is served at the global memory throughput.

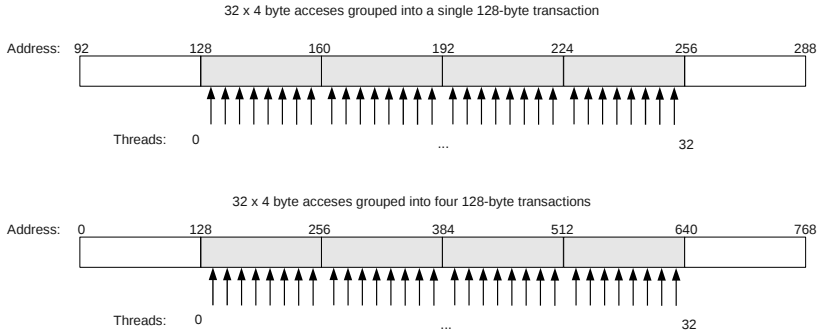


Figure 2.8: Global memory coalescing on the GF104.

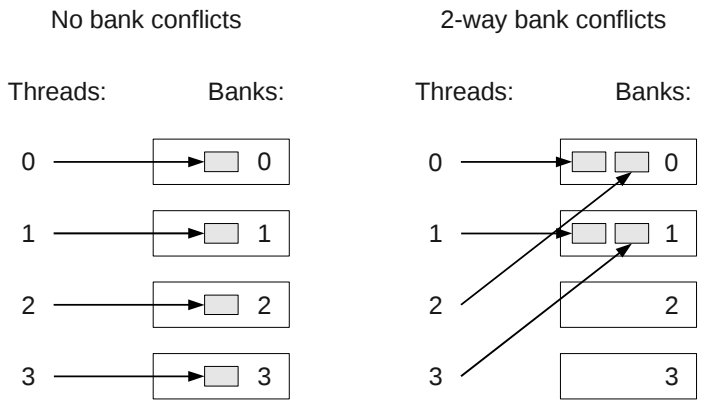


Figure 2.9: Shared memory bank conflicts.

Shared memory access. When accessing shared memory, the accesses are issued individually for each thread and no coalescing is performed. Under certain conditions, requests can be served in parallel, increasing performance. In order to understand when this happens the hardware implementation must be taken into account.

On the GF104, shared memory is divided into 32 equally-sized banks which are organized such that successive 32-bit words are assigned to successive banks. The speed of shared memory comes from the fact that each bank can be accessed independently of the others. When two different threads access different 32-bit words belonging to the same bank a *bank conflict* occurs, and the hardware serializes the request, decreasing performance. Figure 2.9 illustrates how bank conflicts can occur using 4 banks and 4 threads. We see that in the first case each of the threads access a different bank. In this case the requests will be served in parallel, achieving peak performance. In the second case, a group of two threads access different words within the same bank. Requests belonging to the same bank will be serialized, yielding only half of the peak performance.

2.3 Performance prediction

Performance prediction of computer systems plays an important role in many computer science fields: hardware developers need to study future architectures [5], compiler writers need to evaluate the performance of generated code for different machines [30], and software developers need insight in order to tune their code for a particular architecture [18]. In all these fields the main goal is to provide an estimate of the run time of a program on a given architecture.

2.3.1 Approaches

Performance prediction techniques can be classified into three major categories: profile-based, simulation-based, and analytical models. Common to all techniques is the notion of a *model* which abstracts the target architecture. The level of abstraction defines the precision of the model and the efficiency with which predictions can be made with it.

Since we make use of techniques from all the categories, we give a short introduction to each one.

Analytical models Analytical models represent a mathematical abstraction of the program's execution. The result is usually represented in the form of a function which takes into account the target architecture, the input program and the input data.

Predictions are made by evaluating the model for various inputs. The accuracy is very sensitive with respect to the level of abstraction encoded in the model.

There are different techniques for developing a prediction model, ranging from manual construction [5], to program analysis [20] to machine learning [13].

Profile-based Profile-based techniques are most common and involve two phases. In the first phase the program is executed under an *instrumentation tool* which generates statistics for a given program run. In the second phase, the statistics are fed into an *analysis tool* which uses the data to compute an estimation of the program's run time on a specific platform.

The instrumentation phase annotates each basic block of the program with instructions that count the number of times that block was executed. The introduced overhead is acceptable, being proportional to the number of basic blocks.

The analysis phase computes the program's run time by combining the gathered statistics with an estimation of the run time instruction scheduling. The scheduling is usually estimated per basic block using a simple pipeline simulator. The total run time for a basic block is obtained by multiplying this estimation with the number of times the block was executed. The program's run time is the sum of the run time of all the basic blocks. The efficiency of this phase depends on how efficient the scheduling can be computed.

The profile-based approaches work well for sequential architectures, but give poor results in the parallel case. This is because the simulator which is used to compute the schedule does not take into account dependencies between basic blocks and features of modern processors (out-of-order execution, super-scalar execution, caches, etc.).

An example of a modern profiler is Valgrind [22] which is used not only for predicting performance, but also for exposing program errors [14].

Simulation-based Simulation-based performance prediction techniques run every dynamic instruction of the program through a program that models the architecture being studied. Since every instruction is simulated, this approach can generate very accurate predictions. Because each instruction requires substantial computation, this is also the main drawback of simulators.

There are two main classes of simulators: emulators and trace-based simulators. An emulator is a software program that behaves exactly as the studied architecture. It provides an environment in which programs for that architecture can be executed and performance is predicted by collecting performance metrics. This approach gives the most precise results but it also the most expensive in terms of run time.

A trace-based simulator performs the simulation based on a stored trace. Using a similar instrumentation tool as those used in profile-based approaches, a trace of the important program events is generated and saved. Using the trace the program is executed in the simulator and performance metrics are collected. This approach is faster than emulation and it also gives repeatability. The main drawback is that it is difficult to handle data dependent effects and it is impossible to have feedback from the timing model affect the trace.

Example of simulation platforms are the PACE [15], PROTEUS [4], and WARPP [11].

2.3.2 Performance metrics

The benefits of parallelization can be measured by computing several parallel metrics. We only present a few metrics which we will use for estimating the GPU performance. For an in-depth treatment of the subject we refer the reader to [32].

Speedup. The most common is *speedup* which represents the ratio of sequential execution time to parallel execution time. Since there are multiple definitions of sequential and parallel execution times, there are different definitions of speedup. The most common used is *relative speedup*. It refers to how much a parallel algorithm is faster than a corresponding sequential algorithm. It is defined by the following formula:

$$S_p = \frac{T_1}{T_p}$$

where T_1 is the execution time of the sequential algorithm, T_p is the execution time on p parallel processors, and S_p is the speedup. Note that T_p usually contains two components: the actual execution time and the overhead of parallelization.

Bandwidth. Another commonly used metric is *bandwidth*. Bandwidth represents the rate at which data can be transferred. A high bandwidth usually implies that the program processes more elements in a time unit. Performance is predicted by comparing *theoretical bandwidth* to *effective bandwidth*. When the latter is much lower than the former, performance is drastically reduced.

Theoretical bandwidth represents the maximum bandwidth that the hardware can achieve and is usually available in the product documentation. Effective bandwidth is computed by timing the execution of the program and measuring the amount of data which it processes:

$$\text{Effective bandwidth} = \frac{B_r + B_w}{T_p}$$

where T_p is the execution time on p processors, B_r is the number of bytes read, and B_w is the number of bytes written.

2.4 Role of program analysis

Program analysis is the process of inferring properties by analysing the behaviour of computer programs. It is mainly used in compilers for performing program optimizations and validations. Automatic program analysis can be performed statically (at compile time) or dynamically (at run time).

Most performance prediction tools use program analysis approaches in order to compute parameters for their model. In general, dynamic analysis is preferred because it provides more accurate results for the specific execution which the model wants to estimate [20]. In some cases, static analysis is preferred when the main goal is to generate a model for the program which can be instantiated for different executions [1].

In our case we use program analysis to compute values for the parameters of our model. We mostly rely on the existing infrastructure which contains both static and dynamic analyses. In what follows we only give a short summary of program analysis techniques.

Static program analysis

Static program analysis represents a group of techniques which statically compute safe approximations of the run time behaviour of a computer program. This is done by inspecting the source of the program and inferring properties using formal methods. The program can be in source code form or object code. We refer the interested reader to [23] which provides an in depth introduction to the field.

Dynamic program analysis

For real-world programs, static analysis may give poor results do to the fact that it computes approximations. This limitation is a direct consequence of Rice's theorem [31] which states that only trivial properties of programs are algorithmically computable.

The precision can be improved by using dynamic program analysis which computes properties by studying the execution of computer programs. Dynamic analyses sacrifice soundness for accuracy by observing only a portion of the program's execution behaviour. In order for the results to be relevant, the code coverage of the target program must represent a significant part.

Dynamic analysis is also used to determine program invariants [9], software bugs [22] or provide insight for parallelization [33, 18].

3

GPU performance factors

In this chapter we explore the factors which influence the performance of GPUs. Our experiments rely on the concepts introduced in the previous chapter about the CUDA platform and the GF104 GPU.

The performance factors which we need to take into account are discussed in the *CUDA best practices* guide [25]. The guide suggests that a GPU application should be optimized using the following strategies:

1. maximising parallel execution
2. optimizing memory usage to achieve maximum memory bandwidth
3. optimizing instruction usage to achieve maximum instruction throughput

Maximising parallel execution. This means structuring the algorithm such that it exposes the maximum amount of data parallelism. Also, the parallelism needs to be mapped to the hardware as efficiently as possible, by choosing the correct kernel execution parameters.

Optimizing memory usage. This means minimizing data transfers between the host and the device because those transfers have much lower bandwidth than internal device data transfers. Kernel access to global memory also should be minimized by maximizing the use of shared memory on the device. Sometimes, the best optimization might even be to avoid any data transfer in the first place by simply recomputing the data whenever it is needed. The effective bandwidth can vary by an order of magnitude depending on the access pattern for each type of memory.

The next step in optimizing memory usage is therefore to organize memory accesses according to the optimal memory access patterns. This optimization is especially important for global memory accesses, because latency of access costs hundreds of clock cycles. Shared memory accesses, in contrast, are usually worth optimizing only when there exists a high degree of bank conflicts.

Optimizing instruction usage. This optimization focuses on the usage of arithmetic instructions which have high throughput. This implies trading precision for speed when it does not affect the end result, such as using intrinsics instead of regular functions (e.g., replacing `sin()` function calls with `__sin()` intrinsic calls) or single precision instead of double precision. Finally, particular attention must be paid to control flow instructions due to the SIMT (single instruction multiple thread) nature of the device.

Methodology. Since our goal is to predict performance we must measure the impact of these factors on performance. In order to achieve this we designed a set of benchmarks which aim to measure how these factors influence performance. The main purpose of these benchmarks is to reveal the impact of performance factors without going too much into hardware details. We draw conclusions by analysing the execution times. We use `cuobjdump` [28] to inspect native instruction sequences generated by the CUDA compiler and to analyze code.

The general structure of a benchmark consists of GPU kernel code containing timing code around a code section (typically an unrolled loop running multiple times) that exercises the hardware being measured. A benchmark kernel runs through the entire code twice, disregarding the first iteration to avoid the effects of cold instruction cache misses. In all cases, the kernel code size is small enough to fit into the instruction cache.

For measuring time of the entire kernel execution we use the GPU timers which have increased precision [25, Sec. 2.1.2]. In order to measure time while executing a kernel we use the `clock()` GPU intrinsic [26, Sec. B.10]. This returns the value of a per-SM counter which is updated every clock cycle. The clock values are first stored in registers, then written to global memory at the end of the kernel to avoid slow global memory accesses from interfering with the timing measurements. In order to convert the result from cycles to seconds we multiply the value using the GPU clock, which in our case is 1.35GHz.

The main characteristics of our testing system are:

OS	Ubuntu 10.04 64-bit edition
CPU	Intel Core i5-760
GPU	NVIDIA GTX 460 (GF104 chipset with 768 MB)
BUS	PCI-Express 2.0 $\times$16
CUDA	CUDA version 3.2

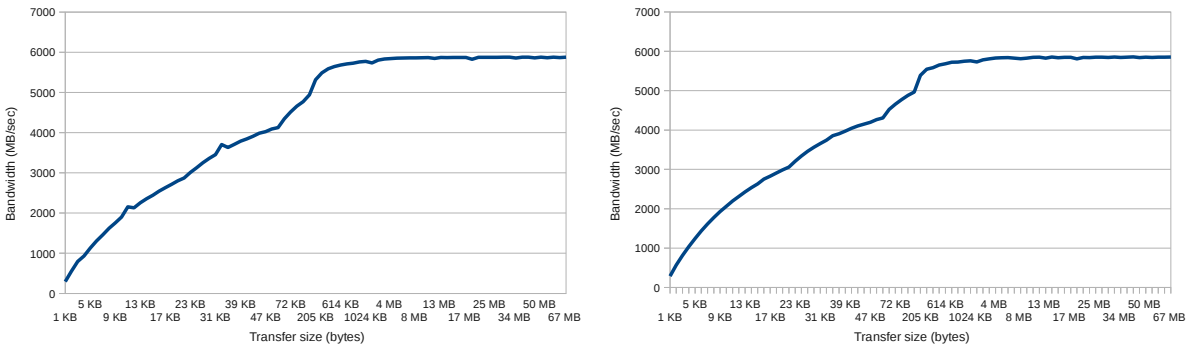
3.1 GPU parallelization overhead

According to [25], the overhead introduced by GPU parallelization is one of the main factors which influences parallel execution and memory usage. Through experimentation we identified the following types of overhead:

- transfer overhead
- CUDA context initialization
- kernel launch overhead

Transfer overhead. In Section 2.2.1 we saw how the host (CPU) and the device (GPU) have separate memory spaces. This implies that data is transfers between them. The transfer is done over the PCI-Express bus which connects the two devices. Note that we are not interested in the hardware details of the bus, but only on its impact on performance. In this case, this translates to an experiment which measures the transfer time for different transfer sizes. In order to copy data between devices we use the CUDA API call `cudaMemcpy` [26, Sec. 3.2.1].

The results of our experiment are summarized in Figure 3.1. The results show that until we saturate the bus (i.e., copy enough data) the bandwidth increases linearly. Once we reach the saturation point, the bandwidth remains constant.



(a) CPU to GPU

(b) GPU to CPU

Figure 3.1: CPU-GPU transfer bandwidth.

CUDA context initialization overhead. Communication between the host and the device is managed through the CUDA context [26, Sec. 3.3.1]. Context initialization is performed transparently when the first CUDA API call is encountered. We measured this overhead between 65ms and 250ms, where values higher than 65ms were only encountered when running the program for the first time. This suggests that the context is also initialized at the GPU driver level.

Kernel launch overhead. Another overhead which is introduced by CUDA is represented by the time it takes to invoke a GPU kernel. In order to measure this we timed the execution of an empty kernel. In our experiment we obtained an overhead of $4\mu s$.

3.2 GPU memory system overhead

The information presented in Section 2.2.2 about the GPU memory hierarchy is not sufficient in order to understand how to predict its performance. To achieve this we use standard memory performance metrics: latency and bandwidth. Latency represents the execution time for a single memory instruction. Bandwidth represents the maximum volume of data which can be transferred in a unit of time. Besides performing latency and bandwidth measurements, we also perform experiments which measure the effect of global memory coalescing and bank conflicts.

Memory latency. In order to measure the latency of global and shared memory we employ a traditional pointer chasing benchmark which is normally used to measure CPU cache parameters [21]. The benchmark consists of a pointer `i` which traverses an array `A` by executing `i = A[i]` in an unrolled loop. Although the measurements include the overhead of address calculation, the total time is dominated by the latency of the memory access. In order to utilize only one GPU SM, the benchmark is executed by a single scalar thread. Cache characteristics are measured by initializing the array as `A[i] = i + stride % N`, where `N` is the size of the array and `stride` is the stride parameter. The experiment assumes a least recently used replacement policy, a set-associative cache, and no prefetching.

The array size defines the cache working set and indicates the size of the cache. The stride size reveals the cache line size because it influences the lines which are fetched; i.e., accessing the array using strides smaller than the cache line means that a single line will contain multiple array elements.

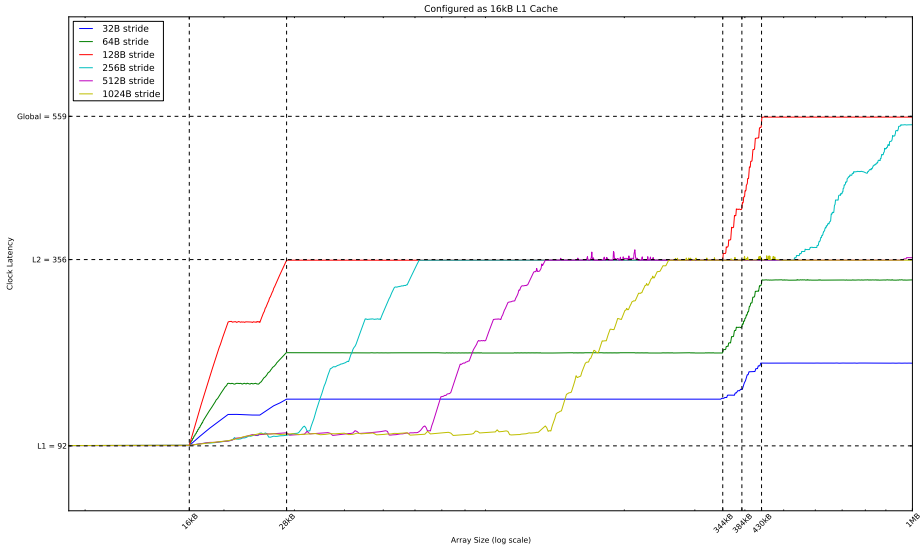


Figure 3.2: GPU memory latency plot results.

Figure 3.2 illustrates the results of our benchmark for stride configurations between 32 bits and 1024 bits and for arrays between 16 kB and 1 MB. The size of the caches (L1 = 16 kB – chosen as a kernel parameter, and L2 = 384 kB) are confirmed by the latency plateaus that correspond to arrays which fit into L1, respectively L2. Notice that for strides smaller than the cache line size, the plateaus are independent of the used stride.

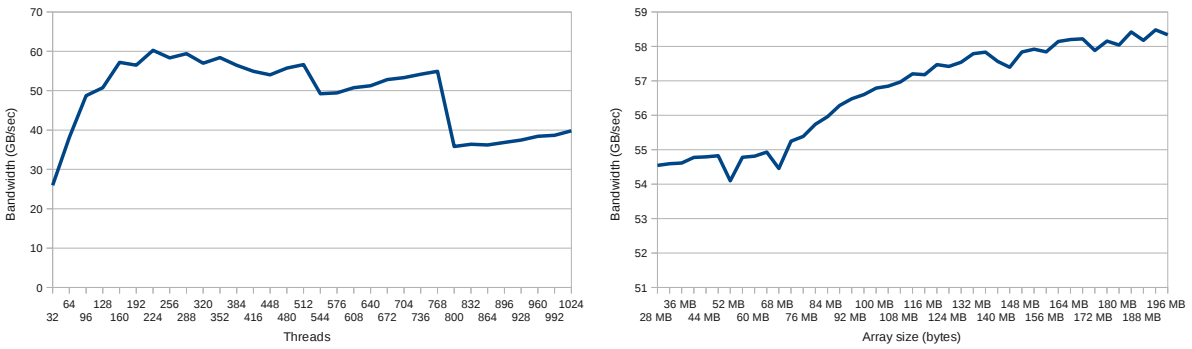
The experiment also shows that the cache line size is 128 bytes, because latency stabilises at the same value for larger strides. Notice that in the 32-, and 64-byte stride plot the L2 latency is roughly 4, respectively 2 times lower. This is explained by the fact that a single cache line contains 4, respectively 2 array entries.

In order to measure the latency of shared memory we performed the same benchmark on an array stored in shared memory. We measured a latency of 50 cycles, independent of array size or stride which indicates that no caches are present.

Notice that the latency of shared memory (50) is different from that of the L1 cache (92), although the same hardware component is used in both cases. This "anomaly" is explained by the fact that the requests get compiled to different instructions. By inspecting the generated assemblies using `cuobjdump` [28], the shared memory access is compiled to a load and a left shift, while normal access is compiled to a load, multiply, and an addition. To get the actual values we subtract the latencies [1] of these instructions from the measured values. The adjusted latencies are: for L1 (shared memory) – 36 cycles, for L2 – 300 cycles, and for global memory – 500 cycles.

Global memory bandwidth. In order to measure bandwidth we use a standard stream copy benchmark [21]. In contrast to the latency experiment, the strategy behind this benchmark is to generate a very large number of memory requests in order to how many can be processed in parallel. To achieve

[1] Details about measuring instructions' latency in Section 3.3



(a) Varying number of threads

(b) Varying transfer size

Figure 3.3: Bandwidth measurements.

this we use the kernel from Listing 3.1 which copies data between two arrays stored in global memory. The large number of memory requests are generated using a high number of threads and a large array.

```
1 int tid = blockDim.x * blockIdx.x + threadIdx.x;
2 int stride = gridDim.x * blockDim.x;
3 for(int i = tid; i < N; i+=stride){
4   dst[i] = src[i];
5 }
```

Listing 3.1: Bandwidth test kernel

We run experiments with varying array size from 28 MB to 196 MB and the number of threads within a block from 32 to 1024. The upper limits of the intervals represent the maximum values which where accepted by the GPU. To get timings only for global memory we disabled caches. The results of our experiments are depicted in Figure 3.3. Note that for GF104, the theoretical bandwidth of global memory is 86.4 GB/sec. The values obtained by our experiment are lower because of the overhead introduced by our measurements (e.g., kernel call overhead, kernel also contains compute instructions, etc.)

Our first experiment (Figure 3.3a) measures the bandwidth when bus contention is high,i.e., many threads access memory. This is achieved by keeping the array size fixed to 196 MB and varying the number of threads within a block. The experiment reveals that 256 threads are enough to achieve the maximum bandwidth. If the number of threads is increased beyond 256, performance drops because requests get serialized.

The second experiment (Figure 3.3b) tests if the bandwidth depends on the amount of transfered data. In order to avoid contention, the number of threads is fixed to 256 (value obtained from the previous experiment) and the array size varies between 28 Mb and 196 MB. The results show that the bandwidth increases linearly with size until it reaches the peak. This is expected, because the number of request is directly proportional with the array size. Thus as long as there is no contention, bandwidth increases because there are more requests served in parallel.

Shared memory bandwidth. Unfortunately the same experiments cannot be used to measure shared memory bandwidth. The reason behind this is that we are not able to generate enough memory requests,

without reaching the hardware limitations imposed by the GPU.

We know that bandwidth measures the amount of memory requests which can be served in parallel. Thus, in order to reach the bandwidth limit a lot of parallel request must be generated. These requests can be generated by independent instructions within a thread or by independent threads. Both resources are limited by the GPU: the maximum number of independent threads is 1024, and the maximum number of independent instructions is 63 (because independent instructions increase the need for registers and a thread can use at most 63 registers). Using the maximum of both values the experiment measures a very small bandwidth of 12 GB/sec. This value is lower than that obtained for global memory because we add overhead by copying values between global and shared memory.

To estimate shared memory bandwidth, we use the data from [25, Sec. 3.2.2.1] which mentions:

> Shared memory banks are organized such that successive 32-bit words are assigned to successive banks and each bank has a bandwidth of 32 bits per clock cycle. The bandwidth of shared memory is 32 bits per bank per clock cycle.

We can use the following formula to estimate the total bandwidth:

$$\frac{4 \cdot n_{block} \cdot n_{sm} \cdot clk}{c} = \frac{4\text{bytes} \cdot 32 \cdot 7 \cdot 1.35\text{GHz}}{1} = 1.2\text{TB/sec}$$

where n_{block} is the number of banks, n_{sm} is the number of SMs, 4 represents the number of bytes read from a single bank, 1.35GHz is the clock frequency, and c represents the time it takes for a block to process the request.

Global memory coalescing. Another important memory performance factor is represented by coalescing. Coalescing is influenced by the access pattern of threads within a warp. In the previous benchmarks coalescing problems where avoided by using unitary stride accesses, which achieves perfect coalescing. In contrast, this experiment (Listing 3.2) uses non-unitary stride accesses which generate multiple requests for a single warp. The number of requests is controlled using the `stride` parameter, e.g., when `stride = 2` there will be 2 request generated for a warp. The number of requests is increased until 32 which corresponds to the scenario where each load of a warp generates a request.

```
1 __global__ void strideCopy(float *src, float *dst, int len, int
      stride)
2 {
3   int i  = ((blockDim.x * blockIdx.x + threadIdx.x) * stride) % len;
4   dst[i] = src[i];
5 }
```

Listing 3.2: Coalescing test kernel

The influence of coalescing on bandwidth can be seen in Figure 3.4: bandwidth decreases linearly with coalescing. Note that for strides higher than 32, bandwidth remains constant. This proves that each thread within a warp issues individual requests.

Shared memory bank conflicts. As mentioned in Section 2.2.2, to achieve high memory bandwidth for concurrent accesses, shared memory is divided into equally sized memory modules (banks) that can be accessed simultaneously. Therefore, any memory load or store of n addresses that spans n distinct

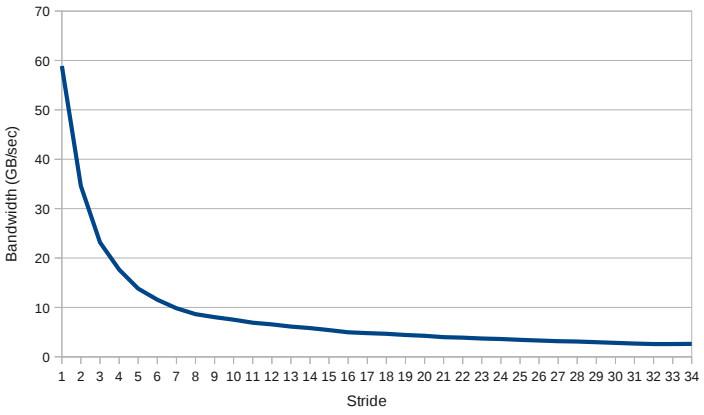


Figure 3.4: Impact of coalescing on bandwidth.

memory banks can be serviced simultaneously, yielding an effective bandwidth that is n times as high as the bandwidth of a single bank.

As in the coalescing benchmark, bank conflicts are influenced by the memory access pattern. The code for this benchmark is illustrated in Listing 3.3. It uses a 32-bit element array (`S`) in order to map each array access to a single 32-bit word of a bank. Thus the targeted bank is influenced by the index of the access. In order to avoid tainting the results, the access to global memory (line 5) is fully coalesced. The number of bank conflicts is controlled using the variable `bank`. A bank conflict occurs if threads tid and $tid + n$ access the same bank. Translating this using the access pattern gives us:

$$tid \cdot bank \equiv_{32} (tid + n) \cdot bank \iff bank \cdot n \equiv_{32} 0$$

This occurs when n is a multiple of $32/gcd(32, bank)$, where $gcd(a, b)$ is the greatest common divisor of a and b. Since a warp contains 32 threads, $n = \overline{0, 31}$. This interval contains $gcd(32, bank)$ multiples of $32/gcd(32, bank)$. In particular, there are no bank conflicts if $bank$ is odd.

```
1 __global__ void conflicts(float *a, int len, int bank){
2   __shared__ float S[1024];
3   int tid = (blockDim.x * blockIdx.x + threadIdx.x);
4   int idx = (tid * bank) % 1024;
5   a[tid % len] = S[idx];
6 }
```

Listing 3.3: Bank conflicts test kernel

The results of the benchmark are presented in Figure 3.5. As expected, no bank conflicts occur when $bank$ is odd and for even values we get $gcd(32, bank)$ conflicts. The benchmark also shows that the delay increases linearly with the number of bank conflicts, i.e., 32 bank conflicts are two times more expensive than 16 bank conflicts.

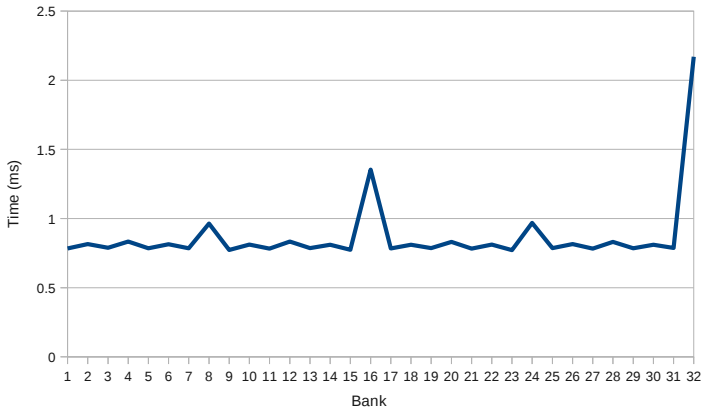


Figure 3.5: Impact of bank conflicts on execution time.

3.3 GPU computation overhead

In order to optimize instruction usage we must identify their cost. Since the GF104 uses pipelined execution units (Section 2.2.2), we adopt metrics which are used also in CPU benchmarking [21]: latency and throughput. The latency of a compute instruction is analogous to the latency of a memory instruction and represents the time of a single instruction. Throughput is analogous to bandwidth and represents the number of instructions which can be executed in an unit of time.

Measuring instruction latency. The latency of an instruction corresponds to the time it takes for it to go through the pipeline. To measure this we execute a number of dependent instructions such as `a += b;b += a;` many times in an aggressively unrolled loop, using a single thread. Using `cuobjdump`, we ensured that we have a one-to-one mapping with respect to native instructions. We made sure that arithmetic does not overflow, but assume that the hardware is not optimized for special values, i.e., 0 or 1.

Listing 3.4 shows the main portion of the kernel used to benchmark `add` instructions. The main part is line 3 which contains an unrolled loop of dependent instructions. By having dependent instructions we ensure that the GPU cannot issue the next instruction until the previous instruction has finished. The chain of instructions is executed multiple times in order to avoid poisoning because of instruction cache misses.

```
1 for (int i=0;i<its;i++){
2   start = clock();
3   repeat128(a+=b; b+=a;); // unroll macro
4   stop = clock();
5 }
```

Listing 3.4: Instruction latency/throughput kernel.

Table 3.1 show the results for various instructions. The results are consistent with the value of 24 cycles mentioned in [26] for the average latency of instructions. Notice that for the `div` instruction the latency is an order of magnitude higher. This is explained in [26, Sec. 5.4.1] by the fact that GPUs do not have a native `div` instruction, and a `div` is compiled to multiple instructions.

Instruction	Latency (cycles/op)	Throughput (ops/cycle)	#warps needed for peak
add	16	32	16
mul	20	16	16
madd	22	16	11
div	317	1.8	5
and	16	32	16
fadd	16	32	16
fmadd	18	32	16
fmul	16	32	16
fdiv	711	0.75	4
sqrt	269	1.6	5

Table 3.1: Latency and throughput of various GPU instructions.

Measuring instruction throughput. In order to measure throughput we must ensure a high number of parallel instructions. We follow the same strategy as in the bandwidth benchmark, that is we increase the number of parallel threads. In order to avoid imprecision caused by block scheduling we use a single block.

The code of the kernel is the same as that for latency (Listing 3.4). The difference comes from the launch parameters: when measuring latency we use a single thread (i.e., grid and block size are 1), and when measuring throughput we use a single block with the maximum number of threads (i.e., grid size is 1, block size is 1024).

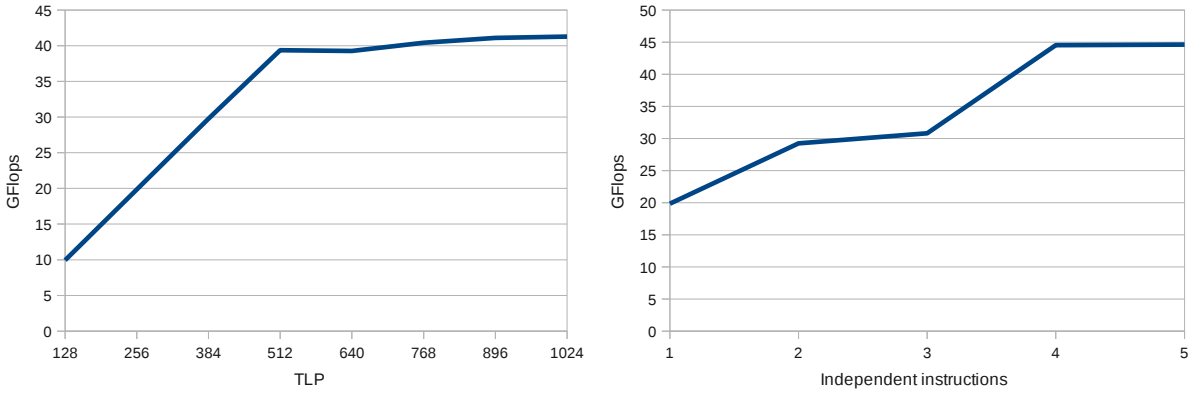
Table 3.1 shows the values obtained for different instructions. Note that although there are 48 CUDA cores the maximum throughput is 32. By varying the types of instructions in the chain we managed to achieve a throughput of 48. The only combination which got this throughput was a `madd` followed by an independent `mul`. This result is explained by the dual issue feature introduced in CUDA 2.1 [26, Sec. G.4.1], where independent instructions within each warp can be issued in the same cycle. This feature is similar to *superscalar* execution present on CPUs (details in Section 2.2.3).

We also conducted experiments to find the maximum number of independent instructions needed to get maximum throughput. We investigated two sources of parallelism: thread level and instruction level.

At thread level, parallelism is achieved by executing the independent instructions of threads. As mentioned in Section 2.2.3, this is the main performance factor for GPUs. At instruction level, parallelism is achieved by executing independent instructions within threads.

Our results are summarized in Figure 3.6 and in Table 3.1. In the TLP experiment (Figure 3.6a) we use threads which contain a long chain of dependent instructions. The results show that 512 threads are sufficient to reach maximum performance. In the ILP experiment (Figure 3.6b) we use a fixed number of threads 256, and vary the number of independent instructions within a thread. The results show that 4 independent instructions are enough to reach peak performance.

With respect to the number of warps needed to reach peak performance we observe that 16 warps are enough for instructions which have high throughput. As a rule of thumb, we observe that instructions which have lower throughput need fewer warps.



(a) TLP

(b) ILP

Figure 3.6: Reaching peak GPU performance using parallelism.

Conclusions

In this section we have thoroughly investigated the factors which determine GPU performance. We used benchmarks to measure the overhead of parallelization and specific factors which influence GPU performance. Regarding overhead we measured: context initialization, kernel launch, and transfer time between CPU and GPU. For performance factors we investigated the role of parallelism for compute and memory instructions and computed the latency and bandwidth/throughput of instructions. Also we established how performance degrades when the kernel suffers from coalescing or bank conflicts problems.

The obtained results indicate a linear relation between the amount of parallelism (either thread level or instruction level) and the performance factors. This is expected because the GPU architecture is designed to execute data-parallel programs. The values and observations obtained through these experiments will serve as the basis for our analytical model.

4
GPU model

By running the experiments presented in the previous section, we now understand how the GPU behaves under different scenarios. In order to reach our goal of performance prediction the next step is to formalize the observed behaviour into a model.

Our general approach is to develop an *analytical model* which represents an abstraction of the GPU architecture. As described in Section 2.3, the performance of analytical models is very sensitive to the level of abstraction which we use. In our case, this translates to how well we model the GPU performance factors identified in the previous section.

Having this in mind, we have chosen a modular model which captures the performance factors. We achieve this by developing individual formulas for each of the factors and combine them in order to characterize the whole GPU kernel.

The formulas are parametrized over variables which represent the performance factors and can be grouped intro three major categories based on the type of the modelled factor.

The first category represents the overhead introduced by parallelization. This contains formulas which model CUDA context initialization, kernel invocation, and overhead introduced by transferring data between CPU and GPU.

The second category deals with the delay of memory instructions on the GPU. This entails global and shared memory latency and bandwidth, coalescing and bank conflicts effects.

The third category models the delay of instructions which do not involve memory access, i.e., compute instructions. This covers latency and throughput of instructions as described in Table 3.1.

In what follows we explain our model in a top-down manner. We start in Section 4.1 by presenting the design choices which are common to all categories. In the following sections 4.2, 4.3 we go into details and explain the particularities of each category.

4.1 Approach overview

By inspecting the results of our experiments and studying the GPU architecture it is clear that *parallelism* is the most important performance factor. This holds both for memory and compute instructions as demonstrated by the fact that increasing parallelism (either by increasing the number of threads or the number of independent instructions) leads to an increase in performance.

Unfortunately this increase is bounded by the limits imposed by hardware. Continuing to increase parallelism after that point does not improve performance. On the contrary, as demonstrated by our

memory benchmarks, parallelism can have a negative impact on performance by creating *contention* for hardware resources (Figure 3.3a).

Since this behaviour is central to GPUs, the formulas of our model will follow a similar pattern. Thus we need to establish bounds for performance and predict the changes in performance with respect to parallelism.

To establish bounds we use the metrics from our experiments: for memory instructions we use latency and bandwidth, and for compute instructions we use latency and throughput. For both instruction types latency corresponds to bad performance, because the execution is sequential and most of the hardware remains idle. On the other end, throughput (for compute instructions) and bandwidth (for memory instructions) correspond to peak performance, because the execution of instructions happens in parallel. By reaching this peak value using parallelism we reach the hardware limits of the GPU. Since bandwidth and throughput both represent an upper bound, we will collectively refer to them as *throughput*.

Our experiments indicate that performance increases linearly with parallelism. This is the case for transfer overhead (Figure 3.1), memory bandwidth (Figure 3.3) and instruction throughput (Figure 3.6). The common behaviour is: performance starts at a low value which corresponds to latency, increases linearly with parallelism, and stabilizes at the peak which corresponds to maximum throughput and high bandwidth.

Using this general pattern we now proceed to explain the main parts of the model. The main formula of our model computes the execution time of the GPU kernel by summing the delay of the overhead and the kernel execution time:

$$\text{TIME} = \text{OVERHEAD} + \text{EXECUTION} \tag{4.1}$$

As mentioned, our formulas are grouped in three categories: overhead, memory and compute instructions. In what follows we present the formulas for each category and how they are combined to estimate execution time.

4.2 Modelling parallelization overhead

As the name suggests, overhead formulas model the penalty introduced by GPU parallelization. In Section 3.1 we identified the following types of overhead:

- transfer overhead
- CUDA context initialization
- kernel launch overhead

Context initialization and kernel launch overhead can be modelled with constants, since their values do not change. From Section 3.1 we know that CUDA initialization takes $65ms$, and a kernel launch takes $4\mu s$.

On the other hand, the overhead introduced by copying data varies with the size of the transfer. By analysing the data from Figure 3.1 we see that the transfer bandwidth increases linearly until it reaches the peak. To reason in terms of transfer time, we use the fact that $bandwidth = size/time$, thus $time = size/bandwidth$. By interpreting the results using this transformations we see that until peak bandwidth is reached, transfer time remains almost constant, and after that it increases linearly.

We model this behaviour using the following equation:

$$\text{TRANSFER}(size) = \frac{size}{\min(BW_{peak}, BW_{inc}(size))} = \frac{size}{\min(5000, 100 \cdot size + 692)} \tag{4.2}$$

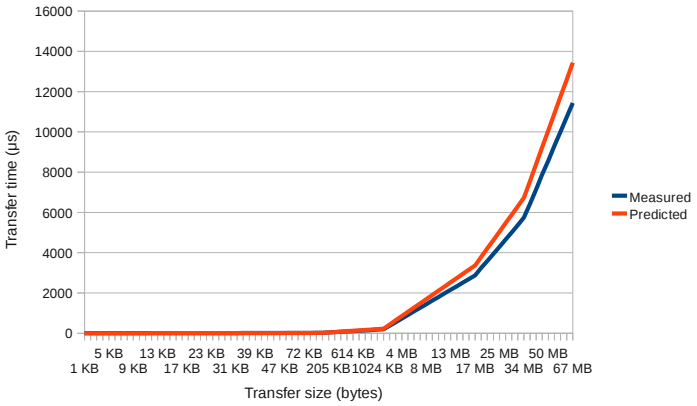


Figure 4.1: Comparison between predicted and measured transfer time.

where, $size$ is the number of bytes transferred, BW_{peak} is the maximum bandwidth (in our case this is 5000 MB/sec), and $BW_{inc}(size)$ is a function which computes the bandwidth obtained for a transfer of $size$ bytes. Since our experiments show a linear increase, we use linear regression to construct the bandwidth function. Using our values this yields the function:

$$BW_{inc}(size) = 100 \cdot size + 692$$

The validation of our function is shown in Figure 4.1. Our function has good behaviour when bandwidth has not reach the maximum. After the peak point, it looses some precision, because of microarchitectural effects which we do not model.

By combining the functions for all three types of overhead we obtain

$$\text{OVERHEAD} = \text{TRANSFER}(size) + \text{CUDA} + \text{LAUNCH} = \text{TRANSFER}(size) + 65ms + 4\mu s \quad (4.3)$$

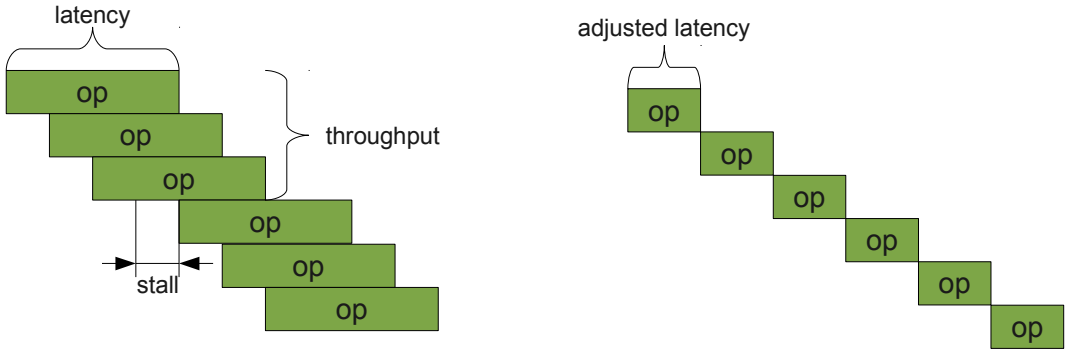
4.3 Modelling kernel execution time

The other important part of our model is the kernel execution time. At an abstract level, a GPU kernel can be viewed as a sequence of instructions. These instructions are executed on the GPU by a number of parallel threads. The execution time is the time it takes for all threads to complete execution. A way to estimate this is by predicting the time for each instruction and summing over the values. Since the delay is computed in cycles we divide using the clock speed in order to convert the results into seconds.

$$\text{EXECUTION} = n_{threads} \cdot \sum_{i \in insns} \frac{delay(i)}{clk} \quad (4.4)$$

A problem which arises in this case, is what value do we use as the delay of an instruction. We know that the delay of an instruction can vary between latency and throughput. As mentioned before, latency corresponds to bad performance and throughput corresponds to peak performance.

Using either values can lead to gross inaccuracies. If we use latency then the execution time is very pessimistic because it assumes that there is no parallelism. If we use throughput, we hit the other extreme, that is our execution time will be very optimistic. Our experiments indicate that the delay of



(a) GPU pipeline execution

(b) Model linear execution

Figure 4.2: Comparison between executing instruction on the GPU and in the model.

instructions is greatly influenced by the amount of parallelism and to what extent the GPU can exploit it. Thus we seek to create a delay formula which takes into account latency, throughput and parallelism.

On the GPU platform, parallelism can come from two major sources: instruction level and thread level. At instruction level, we have independent instructions within a thread which can be used to hide latency and achieve good performance. At the thread level, having more threads means there are more independent instructions.

Another important factor that we must take into account is the hardware which is used to execute the instructions. For example, a multiplication will be executed by a CUDA core, while a load will be executed by the memory subsystem. The two subsystems have different latencies, throughputs and more importantly have different factors which influence performance. For example, the performance of the global memory is influenced by coalescing while that of a CUDA core is not. To account for this we need to distinguish between memory and compute instructions.

4.3.1 Compute instructions

We start by presenting how we model the delay of compute instructions. As mentioned in the previous section, the most important factor is parallelism which can be provided either by independent instructions or by multiple threads.

Using the values reported by our experiments we know that performance can vary between latency and throughput. Since the GPU uses pipelines, instructions can be executed in parallel as illustrated in Figure 4.2a. That is, instructions are executed in parallel until the hardware limit is reached (in our example the limit is 3). After that point, the next instruction has to wait until a slot in the pipeline becomes empty.

Since we do not know the limit for the GF104, we model a pipeline with a limit of 1 and adjust the latencies of instructions accordingly. We have two cases to model. The first corresponds to the case when the pipeline is not full. In this case the total delay remains constant because the instructions are executed in parallel. In the second case a stall occurs when the pipeline is full. We account for this by adding a penalty factor.

Since our pipeline corresponds to sequential execution and we have parallel instructions, we use an

adjusted delay which equally divides the total delay between the parallel instructions. This behaviour is captured by the following formula and illustrated in Figure 4.2b:

$$delay(i) = \begin{cases} \dfrac{latency_i}{ILP \cdot TLP} & \text{if } ILP \cdot TLP \leq max_i \\ \dfrac{latency_i}{ILP \cdot TLP \cdot max_i} + \dfrac{warp_{size}}{throughput_i} & \text{otherwise} \end{cases} \tag{4.5}$$

where $warp_{size}$ is the size of a warp (e.g., 32), ILP and TLP represent the amount of instruction-, respectively thread-level parallelism, $throughput_i$ and $latency_i$ represent the throughput and latency of instruction i, and max_i represents the level of parallelism at which peak throughput is obtained (details in Section 3.3).

The penalty term $warp_{size}/throughput_i$ in the second case is computed by reasoning about what happens when there are multiple parallel instructions. Since there are many parallel instructions, the total time can be expressed as the number of instructions divided by the throughput. Equating this into our formula we obtain:

$$latency_i + n_{ops} \cdot penalty = \frac{n_{ops} \cdot warp_{size}}{throughput_i} \iff penalty = \frac{warp_{size}}{throughput_i} - \frac{latency_i}{n_{ops}}$$

Since we have many instructions we can ignore the $latency_i/n_{ops}$ term which leaves us with:

$$penalty = \frac{warp_{size}}{throughput_i}.$$

The two branches of the formula capture the possible scenarios:

1. Latency is hidden by parallelism. This corresponds to the average case when the GPU can execute multiple instructions in parallel. That is, until we reach a peak throughput the latency of the instruction remains constant.
2. The other scenario is when there is enough parallelism to completely hide latency. From our experiments, we know that increasing parallelism after this point does not lead to any gain in performance. Thus the work of the extra threads is serialized. We model this by adding a penalty for each parallel instruction. The penalty is equal to the reciprocal throughput since the hardware runs at peak performance.

The two scenarios are separated using a threshold for the level of parallelism – max_i. The threshold varies depending on the type of instruction. Our experiments have shown that the most common value is 16 independent instructions (see Table 3.1).

Notice that the delay of an instruction changes from one segment of code to the other based on the amount of provided parallelism.

To validate our formula we measured the throughput for a kernel which contains a number of dependent `madd` instructions by varying the number of warps.

Let us consider a kernel with n dependent `madd` instructions. The throughput is the number of instructions divided by the execution time. The total number of instructions is equal to the number of `madd`s in a kernel (n) multiplied by the number of spawned threads (TLP). Since we deal with dependent instructions $ILP = 1$. Using our formula we have that the total execution time is $TLP \cdot \sum_{i=1}^{n} delay(madd)$. By dividing the two values we obtain:

$$\frac{n \cdot TLP}{n \cdot TLP \cdot delay(madd)} = \frac{1}{delay(madd)}$$

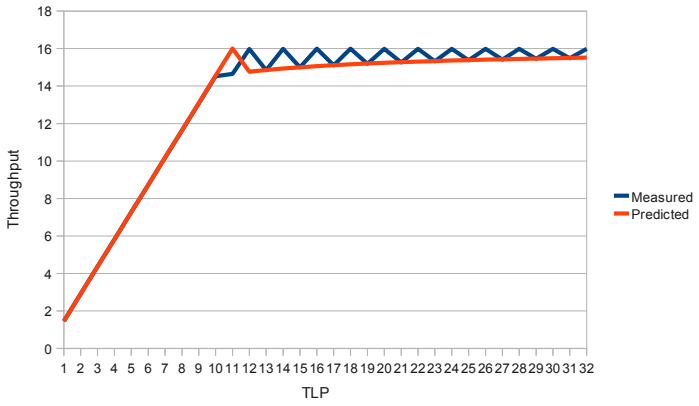


Figure 4.3: Comparison between predicted and measured throughput.

Using the information from Table 3.1 we know that `madd` has a latency of 22 cycles, a throughput of 16 instructions per cycle, and the number of warps needed to reach peak is 11. Plugging these numbers into our formula we obtain:

$$delay(madd) = \begin{cases} \frac{22}{TLP} & \text{if } TLP \leq 11 \\ \\ \frac{22}{TLP \cdot 11} + \frac{32}{16} & \text{otherwise} \end{cases}$$

The validation results can be seen in Figure 4.3. Note that until we reach the threshold, which in this case is 11 warps, throughput increases linearly. After this point, the throughput remains constant. Although we show the validation for `madd`, this behaviour is common to all GPU instructions.

4.3.2 Memory instructions

The other important part of our model is the delay of memory instructions. Memory instructions are handled differently because they are not only influenced by parallelism, but also by coalescing and bank conflicts.

Since our experiments indicate that memory behaves similarly to compute, we use a similar approach to obtain the following formula:

$$delay(i) = \begin{cases} \frac{latency}{ILP \cdot TLP} & \text{if } ILP \cdot TLP \leq max_p \\ \\ \frac{latency}{ILP \cdot TLP} + penalty & \text{otherwise} \end{cases} \quad (4.6)$$

where $latency$ is the memory latency, max_p is the level of parallelism needed to reach peak performance, ILP and TLP are the instruction-, respectively the thread-level parallelism factors, and $penalty$ is the serialization term at peak performance.

We now focus on computing the $penalty$ term. The major difference between compute and memory comes from the way we compute the serialization term. In the compute case only parallelism played a role. Therefore we used only the throughput to calculate it. In the memory case, we also need to account for coalescing and bank conflicts effects.

In general, the memory performance factors can be expressed using two parameters. The first parameter, $transfer$, represents the amount of data that is read or written to memory. This amount can be higher than the requested size because of bad coalescing and bank conflicts. The second parameter, *bandwidth* represents the speed of the transfer. This parameter captures the effects of parallelism in the memory subsystem. That is, having more requests means higher bandwidth, until the peak is reached, after which requests get serialized.

To compute the value for *penalty* we again reason about what happens when there are many memory requests. Since there are many data transfers, the delay can be computed by dividing the transfer size to the bandwidth. By equating this into our formula, we obtain:

$$latency + n_{req} \cdot penalty = \frac{n_{req} \cdot transfer}{BW} \iff penalty = \frac{transfer}{BW} - \frac{latency}{n_{req}}$$

where n_{req} is the number of requests which captures both ILP and TLP, $transfer$ is the amount of data transferred per request, and BW is the memory bandwidth. Since we have many requests we can ignore the $latency/n_{req}$ term, which leaves us with

$$penalty = \frac{transfer}{BW}. \tag{4.7}$$

The amount of transferred data depends on the type of the accessed memory. For global memory transfer size can be increased by bad coalescing, and for shared memory transfer size can be increased by a large number of bank conflicts.

Global memory. In the global memory case, the important factor which influences performance is coalescing. We know from Figure 3.4 that bandwidth drops linearly when requests are scattered. As explained in Section 2.2.3, bad coalescing means that there are more requests issued per warp (see Figure 2.8). We account for this in our transfer formula by introducing a coalescing term:

$$penalty = \frac{size \cdot c}{BW} \tag{4.8}$$

where $size$ is the line size of a global memory transaction, c is the number of requests generated by bad coalescing, and BW is the global memory bandwidth.

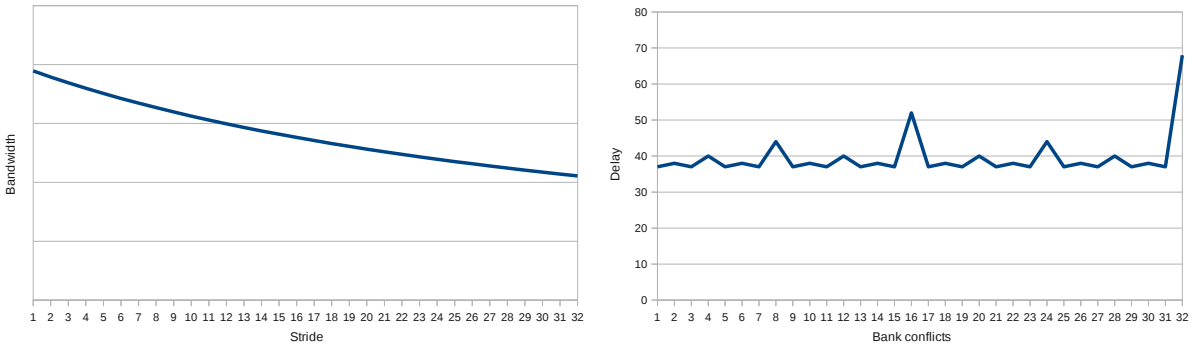
Figure 4.4a depicts the behaviour of our formula in the same conditions as the coalescing experiment from Section 3.2. We instantiate the formula using the values obtained from our experiments: BW is 86.4 GB/s, max_p is 8, and latency is 500 cycles. To use the bandwidth in our formula we express it in bytes/cycle using the GPU clock, which gives us $BW = 86.4/1.35 = 64$. Thus we obtain:

$$delay(i) = 500 + 8 \cdot \frac{128 \cdot c}{64} = 500 + 16 \cdot c$$

To make the comparison easier with Figure 3.4, we plot the bandwidth which is the inverse of the delay. As expected, bandwidth drops when coalescing is increased.

Shared memory. In the shared memory case, we do not have coalescing issues, but instead we have bank conflicts. Having bank conflicts means that requests to the same bank are serialized. We account for this by having two components in our penalty formula:

$$penalty = \frac{warp_{size} \cdot size}{n_{banks} \cdot BW} + \frac{c \cdot size}{BW} \tag{4.9}$$



(a) Coalescing

(b) Bank conflicts

Figure 4.4: Predicted values for coalescing and bank conflicts.

where $size$ is the number of transferred bytes, n_{banks} is the number of banks, c is the number of bank conflicts and BW is the bandwidth of a single shared memory bank.

The first term corresponds to the requests which are served in parallel by all banks, while the second term represents the requests which are serialized.

Figure 4.4b shows how our formula behaves in the same situation as the shared memory experiment depicted in Figure 3.5. As parameters we use the values measured in our experiments: latency is 36 cycles, bank bandwidth 32 bytes per cycle, 32 banks, and for the peak value we use 8 warps. Applying the formula using these values we obtain:

$$delay(i) = 36 + 8 \cdot \left(\frac{4}{32} + \frac{4 \cdot c}{32} \right) = 37 + c$$

The figure shows that the function has the same shape as the experiment.

4.4 Summary

In this section we have presented the analytical model which is used to predict GPU execution time. Besides the execution time, we model also the overhead introduced by parallelization. The formulas were created by identifying performance factors and quantifying their effect using benchmarks. Table 4.1 shows a summary of the parameters of our model.

In the next chapters we will explore how our model is instantiated and present how we compute values for the parameters.

Target	Parameters	Formula
Overhead	• CUDA context initialization • kernel call overhead • CPU-GPU communication	4.3
Execution	Common: • instruction-level parallelism • thread-level parallelism • peak parallelism Compute: • latency • throughput Memory: • latency • bandwidth • coalescing • bank conflicts	4.4 4.5 4.6 4.8 4.9

Table 4.1: Summary of model parameters.

5
Model instantiation

Now that we have an analytical model we are half way to reaching our goal of performance prediction. All that is missing is to instantiate the model by computing values for the parameters.

As with every analytical model, the accuracy is governed by two major factors. The first is the level of abstraction encoded in the model. The second represents the values with which the model is evaluated. In the previous section, we saw that our model takes into account all major GPU parameters. In this section we present ways of computing values for those parameters.

To do this, we must understand what are the sources of these parameters. Because our model is an abstraction of the hardware, a single parameter corresponds to a suite of microarchitectural factors. The effects are related in nature, but can come from a multitude of sources.

Let us take for example memory bandwidth. In reality bandwidth is influenced by the whole memory subsystem and not just by coalescing and bank conflicts. It is also influenced by channel conflicts, the path used for issuing a request and lots of other hardware details which our model abstracts from.

This illustrates the conflict between accuracy and abstraction in an analytical model. Higher accuracy comes from using multiple parameters. But, having to compute multiple parameters can lead to performance loss and/or make the model only applicable to a single architecture.

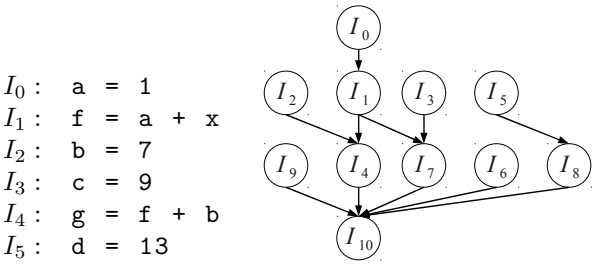
To balance this, we abstract the minor factors by incorporating their average behaviour into the values of major parameters. An example of this, is that we do not use the theoretical memory bandwidth value in our model, but instead use a measured value. Since the value is measured it already contains a penalty for the effects which we do not model explicitly.

In what follows, we describe the sources of different parameters (Section 5.1), present the instantiation workflow of the whole system (Section 5.2) and explain the trade-offs involved. We conclude this section by showing what metrics can be computed using our model (Section 5.3).

5.1 Parameters

In the previous section we validated the formulas of our model by instantiating the parameters using values computed from benchmarks. In this section we explain what are the sources of the parameters and how we can compute values for them.

From a modelling point of view we can group the parameters into three categories based on their source: hardware, kernel, and runtime. We now proceed to explain what factors fall in each category and for each source we describe the methodology for computing values.



(b) DFG

```
I0 :  a = 1
I1 :  f = a + x
I2 :  b = 7
I3 :  c = 9
I4 :  g = f + b
I5 :  d = 13
I6 :  e = 19
I7 :  h = f + c
I8 :  j = d + y
I9 :  z = -1
I10 : JMP L1
```

(a) Program

U_1	U_2	Cycle
I_0	I_2	0
I_1	I_3	1
I_5	I_9	2
I_4	I_7	3
I_8	I_6	4
–	–	5
I_{10}	–	6

(c) First schedule

U_1	U_2	Cycle
I_0	I_2	0
I_1	I_5	1
I_3	I_8	2
I_4	I_7	3
I_9	I_6	4
I_{10}	–	5
–	–	6

(d) Second schedule

Figure 5.1: Instruction scheduling example.

Hardware factors. In the first category, we have factors which are influenced entirely by hardware. These factors do not depend on the kernel, or the values of the input parameters, and typically represent hardware limitations. In our model these factors are: hardware constants (e.g., warp size, number of banks), latency of instructions, throughput for compute instructions, and memory bandwidth.

Values for these parameters are obtained from hardware specifications like the *CUDA C Programming guide* [26] or *CUDA C Best Practices guide* [25] (e.g., warp size, number of banks). The advantage of this is that it requires little effort.

In case the information is missing, another way to compute values is through benchmarks similar to those performed in Section 3. Because the official documentation from NVIDIA does not specifically give values for the GF104, we also employ benchmarks to confirm values of hardware parameters. An example of this is global memory latency: the official documentation gives a value between 400 and 800 cycles, while our benchmark shows that it is 500 cycles (details in Section 3.2).

Kernel factors. The next category deals with factors which are influenced by the kernel code. The most important factor which falls in this category is instruction-level parallelism (ILP). ILP represents the number of independent instructions which can be executed in parallel. Instructions can be executed in parallel if there are no dependencies between them.

Another factor that influences ILP is instruction scheduling: by ordering instructions using the relation defined by dependencies we obtain only a partial order, i.e., independent instructions can be scheduled in any order. Because the hardware is optimized for certain types of instructions the order is important. In CUDA, instruction scheduling is the responsibility of the compiler.

Since the ILP value depends only on the instructions of the kernel we compute its value by examining the data-flow graph and reporting the available parallelism. Since there is no official documentation about the scheduling order, we use an *as-soon-as-possible* (ASAP) ordering of instructions using the latencies obtained through experimentation.

To showcase the importance of scheduling consider the example from Figure 5.1. We have to schedule

the provided basic block (Figure 5.1a) on two compute units where an addition takes two cycles and every other instruction takes one cycle. The difference in scheduling can be seen by comparing the total execution time of two possible schedules. The schedule from Figure 5.1c takes six cycles, while that from Figure 5.1d takes five cycles. The difference between the two schedules lies in the way ties are broken, e.g., scheduling first I_5 then I_3 or the other way around. In our case we use the heuristic proposed by Cooper et al. [7] which suggest scheduling the instruction which finishes first. In their paper, they empirically prove that for real world programs this method works well.

Runtime factors. The last category, represents the factors which are influenced by dynamic sources. That is, they depend on the actual values of the runtime parameters. The most important factor in this category is thread-level parallelism (TLP). As we saw in the previous sections, TLP is the key performance factor on the GPU, because it enables peak performance. The main factor which influences TLP is the kernel launch configuration, which specifies the number of available warps. As we seen, having more warps leads to better performance. Besides this, TLP can decrease because of dependencies: a warp has to wait for the result of an instruction or for a memory request in order to be able to execute.

Because of its dynamic nature, we compute TLP by simulating the GPU execution at a very coarse level. The main goal of the simulation is to keep track of the number of active warps which corresponds to TLP. By performing this simulation we also achieve another goal: we can identify hotspots and report them to the user as potential bottlenecks.

5.2 Instantiation workflow

In this section we present the general workflow for computing parameters values. For parameters which depend entirely on the hardware we run benchmarks or use specifications. The benchmarks and specifications are presented in detail in Section 2.2.2 and Section 3. Here we focus on parameters which are influenced by the GPU kernel and the launch configuration: parallelism (both instruction- and thread-level), coalescing, bank conflicts, and data transfer size.

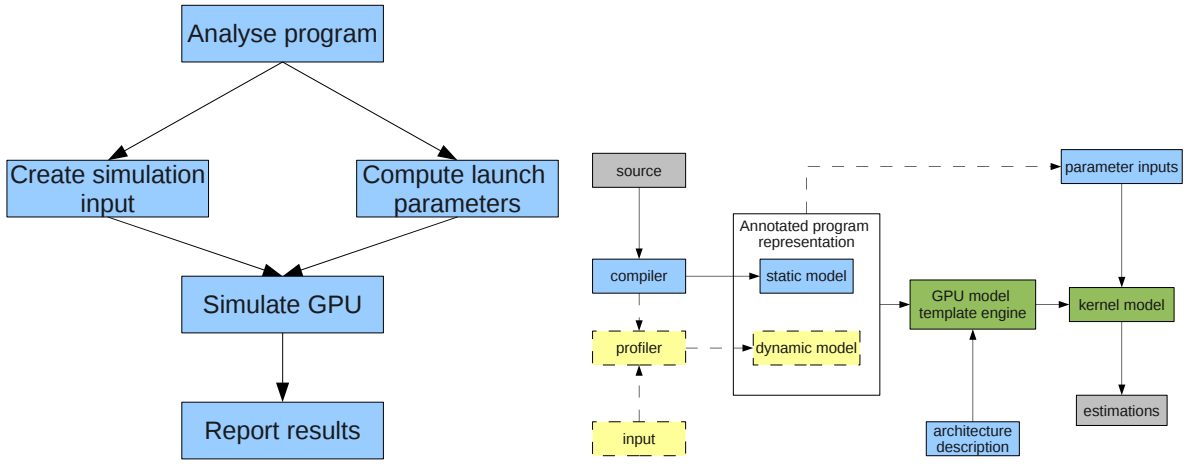
Figure 5.2 gives an overview of our performance prediction system. The main workflow is depicted in Figure 5.2a. We start by running program analyses to compute the majority of parameters (coalescing, bank conflicts, instruction-level parallelism).

The architecture of our system is presented in Figure 5.2b. The analyses represent the foundation upon which we run our simulation.

In our model we estimate thread-level parallelism by running a coarse level simulation. Since the current program representation is not suitable for such a task we use the analyses results to create a simplified version. To run the simulation we also require the size of the workload, which is determined by the kernel launch parameters. Since the C language does not have such a concept, we compute suitable values for the block and grid size.

After running the simulation we aggregate the results into a report which contains performance estimations.

In what follows we focus on explaining the GPU simulation since it computes the most important parameter of the performance model, TLP. Because the simulation relies on program analysis we give a brief overview of the system and defer the details to Section 6.



(a) workflow (b) architecture

Figure 5.2: Overview of performance prediction

Parameters computed using program analysis

Our primary technique for computing parameters values is program analysis. In the context of *vfEmbedded*– the tool in which we integrated our work, program analysis has a central role. *vfEmbedded* mainly relies on dynamic program analysis to compute data dependencies which are then used to parallelize programs.

In parallel with our efforts of performance prediction, a static analysis module has been developed and integrated which contains analyses designed to automatically transform programs to the GPU.

The module provides a report which contains the size of the transfered data, the size of requests for each memory access, and information regarding loads which can benefit from shared memory. The results are used as parameters in our model to estimate transfer overhead and compute the delay of memory instructions. Since the analysis only provides coalescing information, we adapt the technique to compute bank conflicts. Details about this procedure are provided in Section 6.1.

Another important parameter which is computed using program analysis is ILP. Since ILP is determined by the dependencies of the program it can be computed statically. To this end, we developed an analysis which estimates the amount of ILP for a program (Section 6.2).

In summary, the parameters computed using program analysis are:

- coalescing
- bank conflicts
- instruction-level parallelism
- size of transfered data

5.2.1 GPU simulation

The only parameter left to compute is thread-level parallelism (TLP). In contrast to instruction-level parallelism, TLP is influenced by dynamic variables, such as kernel parameters values, block and grid

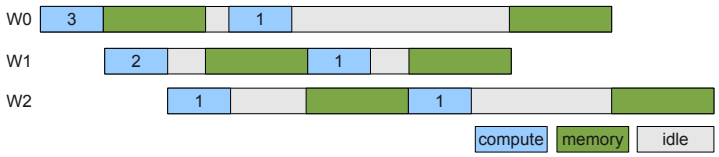


Figure 5.3: Interleaving of warps on the GPU. Numbers represent warps which are ready to execute.

size.

In our model, TLP represents how well threads can be interleaved and corresponds to the number of ready to execute warps. This number varies depending on the computation which are executed. To illustrate this, consider an SM which allows only one warp to execute computations, and only one warp to access memory. Figure 5.6 shows a possible interleaving of a kernel with 2 compute (coloured in blue) and memory (coloured green) instructions using 3 warps. Note that warps execute in parallel only if enough resources are available. The numbers attached to compute instructions represent the available warps. Even if we assume a fixed scheduling, it is infeasible to capture this behaviour with a formula.

Since we cannot use a formula, we turn our attention to another performance prediction technique – simulation. Although more costly than formulas, simulations are designed to model dynamic behaviour with high accuracy.

To keep the computation cost manageable we make some simplifications to the simulation. That is, we focus on keeping track of available warps and do not model the full GPU hardware. This has the added benefit that it makes our model more generally suitable for a range of GPUs.

We now proceed to explain the simulation workflow. We start by presenting the inputs to the simulation: the representation and the kernel launch parameters. After this, we show the simulation algorithm and explain its features and design choices.

Simulation input representation. The current program representation is designed for program analysis. This makes it unsuitable for simulation since some information is missing (e.g., delays of instructions) or there is information which the simulation abstracts from (e.g., data-flow). Thus we perform a conversion which adds the missing information and abstracts unnecessary details. Since our representation is flat we call it a *trace*. A trace represents the computations of a warp.

As we saw in the previous example we only need to keep track of the active warps. To do this we need to model the execution units and the warps which execute on them. From this point of view, we distinguish between instructions which target the execution units of the SM and instructions which target the memory subsystem. The former we name *compute* and the latter *memory* instructions.

From a performance prediction point of view we are only interested in the amount of time it takes an instruction to execute. Thus we add a single attribute to the each operation which represents its delay.

To increase the performance of our simulation we abstract control-flow. That is we create a flat trace of the control-flow graph of the program. Sequential control-flow is not affected, but branches and loops are. For branches we serialize all the alternatives. This corresponds to branch divergence as explained in Section 2.2.3. In the loop case, we make the assumption they are synchronous and do not cause divergence. Thus, we flatten them by multiplying the delay of the instructions which constitute the body by the loop count.

In summary a trace of a GPU kernel is a flattening of the programs control-flow graph, where each node is

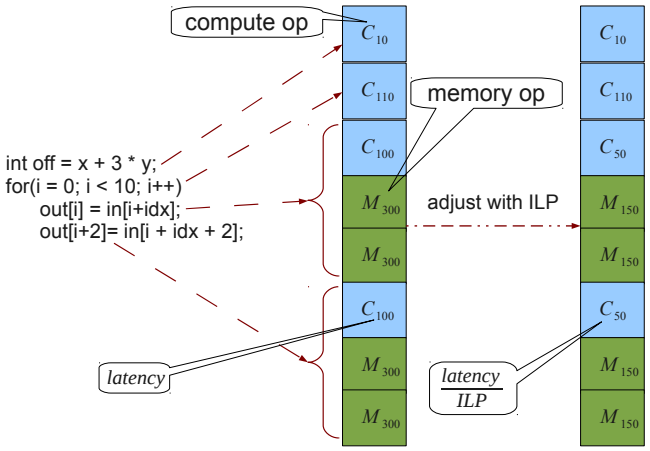


Figure 5.4: Creating a GPU trace.

either compute or memory and is decorated with the delay of the instruction. Because we use the average number of iterations when flattening loops, we obtain a trace which models the average behaviour of a kernel warp.

Figure 5.4 shows an example of the creation of a trace. For ease of reading, we use a high-level language as the initial representation. For delays we assume that each compute instruction takes 10 cycles, while a memory instruction takes 30 cycles. Notice that the delay of instructions which correspond to the loop is adjusted using the loop count, i.e., it is higher by a factor of 10.

The second part of the figure shows how we prepare the trace for the simulation. Since ILP can be computed statically, we adjust the delays of instructions using its value. This is the first step in computing the delay formulas (4.5) and (4.6). In the example kernel, we have an ILP of 2 for the body of the loop, and an ILP of 1 everywhere else.

Computing kernel launch parameters. The other prerequisite for the simulation are the kernel launch parameters: the grid and block size. The current infrastructure on which we work supports converting loops to GPU kernels. In order to be able to compute a valid mapping between the loop iteration space and the GPU thread grid several constraints must be satisfied. One of the constraints requires the loops to be normalized. This amounts to having all the induction variables start at zero and incrementing them by one at every iteration.

Because we have the loop counts we need to focus only on computing a suitable block size. As mentioned in Section 2.2.3, block size is influenced by the number of used registers, the amount of allocated shared memory, and by the hardware constraints of the GPU.

Since the current program representation makes use of virtual registers we must use register allocation techniques to estimate the number of used registers. Details about this analysis are available in Section 6.4. To compute the amount of shared memory we process the information reported by the program analysis infrastructure. Details are given in Section 6.1.

To abstract loop dimensionality we compute the number of warps per block (n_w) and then convert that number into a block size. This also ensures that the block size is a multiple of the warp size, thus avoiding spawning threads which do no useful computation. If the loop is one dimensional, then the conversion is just a multiplication between the number of warps and the warp size. In the two dimensional case, we

fix the y component to the warp size and set the x component to the number of warps per block. This is captured by the following formula:

$$blocks(n_w) = \begin{cases} n_w \cdot warp_{size} & \text{if loop is 1D} \\ (n_w, warp_{size}) & \text{if loop is 2D} \end{cases}$$

The problem now becomes to compute n_w given the total number of threads n_t (which corresponds to the loop counts), the number of registers per thread n_r, and the amount of shared memory per block n_s.

We use three strategies to compute n_w: evenly divide work amongst SMs, achieve high utilization, and ensure enough warps to hide latency of instructions. Using these strategies we obtain several values for n_w from which we choose the minimum. This corresponds to:

$$n_w = \min\left(n_w^1, n_w^2, n_w^3\right)$$

To evenly divide work we divide n_t by the number of SMs, that is:

$$n_w^1 = \frac{n_t}{n_{SM} \cdot warp_{size}}$$

To achieve high utilization we compute the maximum number of warps with respect to registers, shared memory, and hardware limits:

$$n_w^2 = \min\left(max_w, \frac{max_r}{warp_{size} \cdot n_r}, \frac{max_s}{warp_size \cdot n_s}\right)$$

where max_w is the maximum number of resident warps, max_r is the total number of registers per SM, and max_s is the total amount of shared memory.

The third strategy computes the number of warps needed to hide the latency of instructions. To do this we adopt the strategy from [26, Sec. 5.2.3], which indicates the following formula:

$$warps_l(i) = \begin{cases} c \cdot L & \text{for compute instructions} \\ \frac{c \cdot L}{ai} & \text{for memory instructions} \end{cases}$$

where c is the compute factor specific to each CUDA generation (for CUDA 2.1 $c = 4$), L is the latency of the instruction, and ai is the arithmetic intensity. For more details about how we compute arithmetic intensity we refer the reader to Section 6.3. Since the kernel has multiple instructions we take the maximum:

$$n_w^3 = \max_{i \in insns} warps_l(i)$$

Simulation algorithm. Having the trace and the launch parameters we can perform our simulation. As mentioned to estimate TLP we need to keep track of the ready to execute warps. We do this by simulating the execution units of the GPU. Our target is to model the execution described in Section 2.2.3.

Our approach contains two steps: distribution of workload, and simulation. The goal of the first step is to create the input for simulation. That is, using a trace and the launch parameters, create warps which can then be simulated. The goal of the second step is to keep track of the active warps by simulating GPU execution. In what follows we present the two steps and the trade-offs involved.

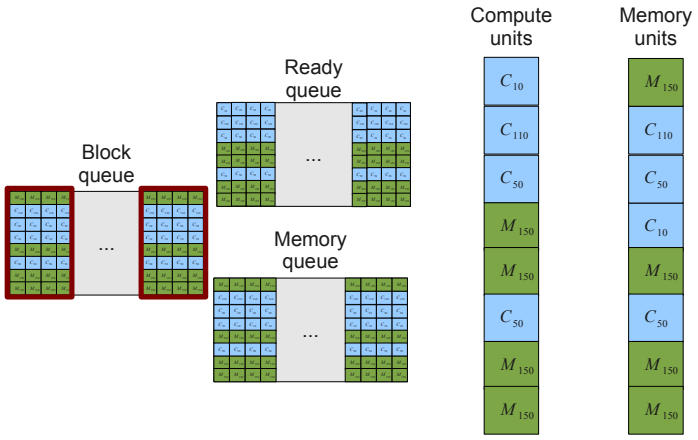


Figure 5.5: GPU simulation architecture overview.

As mentioned earlier, the first step deals with creating warps which will be simulated. It is infeasible to simulate the full workload of the GPU because it can contain an arbitrary amount of warps.

To make the workload manageable, we assume that blocks have roughly the same behaviour. This means that they take roughly an equal amount of time to execute. This assumption is reasonable because blocks contain warps which model average behaviour. Furthermore, this fits with the general GPU model which is optimized for data-parallel algorithms.

This allows us to simulate only a fraction of the workload and extrapolate the results. That is, if we have a workload of n warps, we only simulate W warps and obtain a runtime of t cycles. We extrapolate the results as follows:

$$total = \frac{t \cdot n}{W}$$

Another assumption that we make is that the workload is equally distributed. This means that each SM receives an equal amount of blocks, which allows us to simulate only a single SM. Since SMs execute in parallel, the total execution time is the execution time of a single SM.

In summary, if we have a workload of n warps, and a GPU with SM multiprocessors, we perform the simulation on:

$$\min\left(\frac{n}{SM}, W\right)$$

We now proceed to explain the simulation algorithm. Figure 5.5 contains an overview of the simulation architecture. We use two types of execution units which correspond to the types of trace instructions. The execution units are modelled as queues which contain warps. We model contention on them by using queues. That is, when an execution unit is busy and a warp requests access we enqueue the warp in order to process it later. To respect the hardware limit regarding maximum resident warps res_w, we use a block queue and allow the execution of up to res_w warps.

The value of TLP is the number of warps which are ready to execute. When an instruction is executed we adjust its delay by dividing with the TLP factor. By doing this we obtain the final formulas of our model: (4.5) and (4.6).

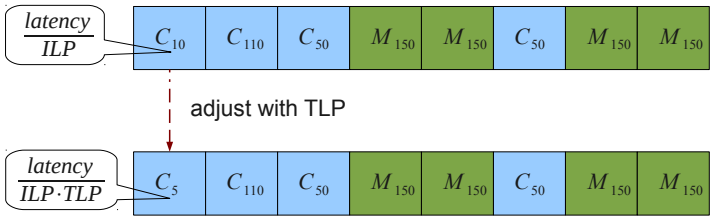


Figure 5.6: Example of adjusting delay using TLP.

As we saw in those formulas, the division by TLP models the fact that the instructions latencies are hidden using other warps. In our simulation the TLP factor can change for the warps in the ready queue by the time they get executed. If this happens then the total execution time changes and the simulation is not faithful to the model. To account for this we also adjust the delays of instructions from the ready queue and mark them as processed. When they are executed we only divide by TLP if we have not already done so.

Figure 5.6 shows an example of adjusting the delay of a warp using $TLP = 2$. The scenario assumes that there are 2 warps, one in the queue and one which is executing. Notice that only the first instruction is affected. The other instructions will be adjusted when they will be scheduled for execution.

We keep track of the total time by performing a lock-step simulation of the execution units. The simulation looks at the delay of the current instruction from the warps which are executing and selects the minimum value. To simulate the passage of time a global clock is incremented with the selected delay. The next step is adjusting the delays of warps instructions by subtracting the chosen delay. Instructions whose delay is lower than 0 are removed from the warp.

To identify hotspots we keep track of the time when only memory instructions are executed. We do this by associating to each memory instruction a value which represents the percentage of its delay in which the GPUs compute units where idle. This corresponds to a scenario when the ready queue and the compute units are empty and only memory instructions are executed.

Listing 5.1 contains the pseudo-code of the algorithm. To make the simulation more general we parametrize it on:

- number of compute units which corresponds to GPUs that can execute warps in parallel
- number of memory units which corresponds to the number of outstanding memory requests
- maximum number of resident warps and blocks
- maximum number of warps to simulate which balances precision with efficiency
- the queuing and dequeuing strategy used for execution units and queues which allows us to see the effects of scheduling policies on performance

5.3 Reporting performance

After instantiating the model and running the simulation we need to give the user information that will help him decide if GPU parallelization is beneficial for his program. Our report contains three types of information: values for metrics which characterize the performance of the kernel, memory hotspots, and recommendations for parameters which influence performance.

Algorithm 5.1 Simulation algorithm

```
procedure SIMULATE(blocks)
    time ← 0
    queue_r, queue_b ← split blocks according to number of SMs and res_w
    queue_m, unit_c, unit_m ← ∅
    while queue_r ≠ ∅ or queue_b ≠ ∅ do
        if queue_r has space for another block then
            add blocks of warps from queue_b to queue_r
        end if
        unit_c, unit_m ← distribute warps from queue_r
        TLP ← |queue_r| + |unit_c|                    ▷ TLP is the number of executing warps
        unit_c, unit_m ← adjust delay using TLP
        delay ← min_delay (unit_c, unit_m)            ▷ instruction with the minimum delay
        if unit_c = ∅ then
            increase idle time of instructions from unit_m by delay
        end if
        unit_c, unit_m ← subtract delay from unit_c and unit_m
        time ← time + delay
    end while
end procedure
```

The first category contains metrics discussed in Section 2.3.2: speedup, and effective bandwidth. Furthermore we also compute arithmetic intensity and number of floating point instructions per second (FLOPS). Besides arithmetic intensity, all the others are derived from execution time.

Speedup is computed by dividing the sequential time by the sum of GPU kernel execution and the overhead. We obtain the sequential time from the available infrastructure:

$$speedup = \frac{sequential}{overhead + parallel}$$

Effective bandwidth is computed by dividing the size of the accessed data by the execution time. We compute the size by combining the size of each memory instruction with the number of times it was executed.

$$bandwidth = \frac{\text{accessed size}}{overhead + parallel}$$

FLOPS is a widely used metric for reporting performance of computer systems. It represents the ratio between the number of floating point instructions and the total execution time. The number of floating point instructions is computed using the dynamic instructions counters available in *vfEmbedded*.

$$FLOPS = \frac{\text{number of floating point instructions}}{overhead + parallel}$$

Arithmetic intensity is a GPU specific metric which measures the ratio between compute and memory instructions. This can be viewed as a metric which computes the data-parallelism of a kernel. Having high arithmetic intensity means that there are more computations performed per data element. Details about how we compute this metric are presented in Section 6.3.

$$AI = \frac{\text{compute instructions}}{\text{memory instructions}}$$

Besides these metrics, we also report the memory hotspots of the kernel as computed in our simulation.

As part of running our simulation we compute values for parameters which influence the performance of the kernel. We use heuristics to compute values for block size. Furthermore, for shared memory optimizations we choose the ones which are the most beneficial. We do this by preprocessing the shared memory opportunities presented by the program analysis infrastructure. Because shared memory size is limited, there can be scenarios where all the opportunities do not fit. When this occurs, we choose the ones which are the most beneficial from an execution time point of view. Details about this are presented in Section 6.1.

6 Supporting analyses

In this section we present the analyses which we use to compute values for parameters. As mentioned in the previous section, this represents the foundation of our analytical model because it computes values for the majority of parameters. In what follows we present the available infrastructure (Section 6.1) and then focus on the developed analyses: instruction-level parallelism (Section 6.2), arithmetic intensity (Section 6.3), and spill/reload analysis (Section 6.4).

6.1 Available infrastructure

In order to implement our program analyses we mainly rely on the existing infrastructure of *vfEmbedded*. We were able to reuse and build upon it in order to compute values for the parameters of our model. We work on an annotated program representation which contains information suitable for both static and dynamic analyses.

Since our analyses work with language agnostic concepts (e.g., data-flow) we do not clutter the presentation with details regarding the representation. For our purposes, it suffices to say that the program is represented using a typed control- and data-flow graph (CDFG) which is in static single assignment form (SSA) [8]. The nodes of the control-flow graph represent basic blocks which correspond to sequences of instructions through which control flows sequentially. Edges between basic blocks represent jumps in the program. At the basic block level, nodes correspond to instructions and edges correspond to data dependencies between instructions. Other details regarding the representation will be given when needed.

Available information

From the static analysis module we make use of the generated report. This contains the size of the transferred data, the size of requests for each memory access, and information regarding loads which can benefit from shared memory. An important property of the report is that it is parametrized. This is useful from a users point of view to generalize to other instances, but in our case this can hinder our goal of performance prediction because the variables can correspond to any value. The report can be instantiated using values which correspond to loop counts. To obtain those counts, we use the dynamic analysis infrastructure and evaluate the report, but even after this step variables can remain.

To illustrate this consider the kernel from Listing 6.1. Note that the value of `j` depends on the inbound variable `s` which is not constrained by the loop count `N`. In this case, the coalescing report for the `a[j]` access will be parametrized by `s`. Since we don't know anything about the actual value of `s`, we assume bad coalescing and consider the maximum number of requests (32).

```
1 for(int i = 0; i < N; i++){
2   int j = s * i;
3   a[j] = j + i;
4 }
```

Listing 6.1: Kernel for which the instantiated report contains variables

As a general rule, when interpreting values which contain variables we take a pessimistic approach. As the static analysis package is not yet published, we cannot provide any references, and restrict ourselves to explaining only the parts which we use.

The number of memory requests represents the coalescing parameter in formula (4.8), while the size of the transferred data is used to compute parallelization overhead (4.2). When the number of requests is a variable we assume bad coalescing using the maximum number of requests, i.e., 32. If we encounter a parametrized transfer we cannot provide a maximum value, and ignore the transfer. Although it sounds unreasonable, in practice this choice has little impact on the quality of our predictions because most of the values in GPU programs are dependent on loop counts, for which we already have values.

Processing shared memory information

The existing infrastructure produces information about which loads can benefit from shared memory. Without this information all loads would be to global memory because the concept of shared memory is not available in the C language.

In order to give the user a real estimation about the actual benefits of GPU parallelization, we suggest which sharing opportunities give the best results and compute the number of bank conflicts.

Computing bank conflicts. From a performance prediction point of view, having shared memory implies that coalescing effects are replaced with bank conflicts. In order to compute the number of bank conflicts for a memory instruction we adapt the technique developed for computing coalescing.

The static analysis infrastructure computes coalescing requests by analysing memory expressions. The expressions symbolically represent the address of the instruction and are parametrized with program variables. Algorithm 6.1 gives an overview of the algorithm. We receive as inputs the access expression f which corresponds to a generic warp, the number of banks n_b and the width of a bank b_w. To compute bank conflicts we instantiate the expression for all the threads of a generic warp. The number of bank conflicts is detected by looking at how many of these offsets fall within the same bank. If we cannot fully evaluate the expressions we pessimistically return the maximum number of bank conflicts.

Instantiating shared memory opportunities. As mentioned the output of the static analysis is a list of sharing opportunities. It can be the case that not all of these fit into shared memory, which leads to the need to choose between them.

We can view this a knapsack problem: we have a knapsack which corresponds to shared memory capacity, and the opportunities are objects which we must pick. The *weight* of an object represents the size that it occupies in the knapsack, which in our case is the size of the opportunity. The value of an opportunity depends on the number of times that instruction was executed.

As with other results obtained from static analysis, the size of an opportunity is parametrized using the

Algorithm 6.1 Bank conflicts algorithm

procedure GET_BANK_CONFLICTS(f,n_b,b_w)
  offsets $\leftarrow [f(x), f(x+1), \ldots, f(x+warp_{size})]$
  $banks \leftarrow map\ (\lambda\ o \rightarrow (o\ /\ b_w)\ \mod n_b)$ offsets ▷ compute the bank accessed by each thread
  if *banks* contains variables **then**
    return $max_{\text{conflicts}}$
  else
    $bank_{map} \leftarrow$ create mapping from banks to threads
    return max($bank_{map}, max_{\text{conflicts}}$)
  end if
end procedure

dimensions of the block. To obtain numeric values, we instantiate the variables which correspond to the block size. To do this we use the same methodology as that used for computing block size in the simulation. The only difference is that we ignore shared memory constraints since our goal is to compute them.

Since in the general case knapsack problems are NP-hard, we use a greedy heuristic to solve our problem.

Algorithm 6.2 Greedy algorithm for choosing shared memory opportunities

procedure GET_SHARED_MEMORY(*oportunities*,*block*,$freq$,mem_{sh})
  $sizes \leftarrow$ evaluate *oportunities* using *block*
  opportunities $\leftarrow$ remove *opportunites* which have parametric *size*
  $benefits \leftarrow map\ (\lambda s_i \rightarrow freq_i/s_i)\ sizes$
  sort *opportunities* in decreasing order of $benefits$
  shared $\leftarrow$ choose first n entries from *opportunities* which fit in mem_{sh}
  return *shared*
end procedure

Algorithm 6.2 contains the main steps of the algorithm. We start by evaluating the opportunities using the given block size. We compute the benefit of each opportunity by dividing its execution frequency to its size. Thus opportunities which are often executed are preferred. We sort in decreasing order the opportunities based on their benefit, and choose the ones which fit into the available shared memory.

6.2 Instruction-level parallelism approximation

As we saw in Section 4 one of the most important parameters of our model is the ILP factor. ILP represents the number of instructions which can run in parallel. To illustrate this concept consider the following example:

```
1 e = a + b
2 f = c + d
3 g = e * f
```

The instruction on line 3 depends on the previous two instructions, thus it cannot be computed until both of them have completed. On the other hand, the two additions do not depend on other instructions,

so they can be computed in parallel. If we assume that an instruction takes 1 cycle to complete, then the three instructions will finish in 2 cycles, giving an ILP of 3/2.

In order to estimate the amount of ILP we analyse the data-flow graph (DFG). Our formula for computing ILP is to take the ratio between the number of instructions in the graph, and the longest dependency chain. That is:

$$ILP = \frac{n_{insns}}{p} \tag{6.1}$$

where ILP is the instruction-level parallelism, n_{insns} represents the instructions of the DFG, and p represents the longest path in the DFG.

The longest path corresponds to instructions which are sequentially executed. If we assume that each instruction takes the same amount of time t, then the execution time will be the sum of the instructions on the longest path. This is because other instructions have less dependencies and will finish sooner. Using this analogy, ILP can be viewed as a form of speedup:

$$ILP = \frac{\text{sequential execution}}{\text{parallel execution}} = \frac{n_{insns} \cdot t}{p \cdot t} = \frac{n_{insns}}{p}$$

In general computing the longest path in a graph is an NP-hard problem. In the case of acyclic graphs, the problem can be solved using a topological sort in $O(|V| + |E|)$, where V is the vertex set and E is the edge set of the graph. In order to use this algorithm, we take advantage that our DFG is in SSA form. A property of SSA is that each variable has exactly one definition. This means that the graph does not contain any cycles.

If we apply the formula to the previous example we obtain $ILP = 3/2$ which is the exact value. Note that our parameter is an approximation for the real ILP value. This is because in reality, ILP also depends on the ability of the hardware to extract it. That is if we have a program with infinite ILP, the hardware is not able to process it. In our model we explicitly model this behaviour by adding serialization penalties, as explained in Section 4, formulas (4.5) and (4.6).

6.3 Arithmetic intensity analysis

As described in Section 2.2.3, an SM relies on thread-level parallelism to maximize utilization of its functional units. Utilization is therefore directly linked to the number of resident warps. At every instruction issue time, a warp scheduler selects a warp that is ready to execute its next instruction, if any, and issues the instruction to the active threads of the warp.

In Section 5.2.1 one of our strategies for computing block size is to ensure there are enough warps to hide latency. In that case, arithmetic intensity was used to scale the latency of memory instructions. This corresponds to the empirical observation that more warps are needed to hide latency if the ratio between on-chip and off-chip instructions is high.

In order to compute this value, we analyse the kernel and compute the number of on-chip and off-chip instructions. An instruction is considered on-chip if all of its arguments are either registers of other compute instructions or come from shared memory. In all other cases we consider the instruction off-chip.

Algorithm 6.3 gives an overview of the algorithm. We have as input: the data-flow graph DFG, the variables which are stored in shared memory sh, and the input and output variables of the kernel $inouts$. Using sets `on` and `off` we keep track of the variables which are either on- or off-chip. Variable n denotes the number of on-chip instructions, while variable m denotes the number of off-chip. The algorithm also

Algorithm 6.3 Arithmetic intensity algorithm

```
procedure ARITHMETIC_INTENSITY(DFG,sh,inouts)
    on ← sh
    off ← inouts
    n, m ← 0
    for all nodes v ∈ V(DFG) in BFS order do
        if uses(v) ∩ off = ∅ then
            n ← n + 1
        else
            m ← m + 1
        end if
        if v is ld/st and uses(v) ∩ off = ∅ then
            on ← defs(v) ∪ on
        else
            off ← defs(v) ∪ off
        end if
    end for
    return n/m
end procedure
```

makes use of two functions: `uses` which returns the operands of an instruction, and `defs` which returns the result variable of the instruction.

As per our definition, we initialize *on* with variables stored in shared memory. We consider the input and output variables to be off-chip because they are stored in global memory. The algorithm examines the nodes of the DFG in a breadth-first manner in order to ensure that all of the current nodes operands are already processed. We distinguish between two types of instructions: memory (represented by loads and stores) and everything else. For all instructions we check if their operands are on- or off-chip and increase the corresponding counters. In the case of memory instructions we also add the resulting variable to the corresponding set. This is done so that depending instructions treat them accordingly.

6.4 Spill/reload analysis

The program representation of our infrastructure uses virtual registers. Usually the number of live variables at an instruction (i.e., register pressure) is much higher than the number of hardware registers. Even though GPUs have thousands of registers, they are shared between thousands of threads. To solve this problem the compiler introduces *spill* code by saving some of the registers to memory and restoring them when they are needed.

From a performance prediction point of view, we are interested in this behaviour because spill code introduces memory accesses which are costly. If we know that an instruction generates a spill, then we modify the trace such that it contains an additional memory instructions. We know from [26, Sec. 5.3.2.2] that on the Fermi architecture spilled variables are cached. In accordance to this we use the shared memory timings.

To detect spills we use the well known *MIN* [3] algorithm which spills the value which has the furthest use. The algorithm is given in Algorithm 6.4. It operates on the data-flow graph of a basic block.

To compute the instruction which has the furthest use, we define the **nextUse** function. This function

computes the next-use distance of a variable v at an instruction I by counting the number of instructions between I and the next use of v in the block. If I is the next use, the distance is 0, and if there is no other use in the block, the distance is ∞.

The variables which are currently in registers are denoted in the set W. Initially W is empty. The algorithm traverses the basic block from entry to exit, updating W according to the effects of each instruction. On the GPU, an instruction I can be viewed as:

$$I : (\underbrace{y_1, \ldots, y_m}_{\mathbf{defs}(I)}) \leftarrow \tau(\underbrace{x_1, \ldots, x_n}_{\mathbf{uses}(I)})$$

x_i are the operands which must be available in registers, and y_i are the results which are written to registers. At any program point, the register file cannot contain more than k (the number of available registers) elements. The effects of an instruction I on W are:

1. All variables in $\mathbf{uses}(I) - W$ have to be reloaded before I. Thus, they have to be added to W. If W does not have enough room, $|W| + |\mathbf{uses}(I) - W| - k$ variables in W have to be spilled.
2. None of the variables in $|\mathbf{defs}(I)$ can be in W directly in front of I since all of these variables are dead there. Hence, we need $\mathbf{defs}(I)|$ free registers. If there is not enough room in W, $|W| + |\mathbf{defs}(I) - k|$ variables have to be evicted from W.

These steps are performed by the algorithm on each instruction using the helper function **LIMIT**. The function takes the current register set W and sorts it according to the next-use distance from I, and evicts all but the first m variables. The first call to **LIMIT** makes room for the operands and the second provides room for the results. In the latter case, the next-use distance is measured from the next instruction because the uses of I do not matter when I writes its results.

The algorithm also utilizes the fact that the program is in SSA form. A property of SSA is that each variable has exactly one definition and thus needs to be spilled at most once. If a variable is evicted multiple times, a spill is placed only at the first eviction. We keep track of evicted variables using the set S which corresponds to the variables spilled from W. We modify S in two scenarios: when a variable is reloaded, and when it is spilled. In the first scenario, since the variable is reloaded it must have been spilled previously. In the second scenario we only spill a variable if it was not already spilled and its next use is not ∞.

As a starting value for k we use the maximum number of registers per thread, which in the Fermi architecture is 64.

Summary

In this section we have seen how we employ the current program analysis infrastructure to compute values for the parameters of the model. We show how we compute values for coalescing and bank conflicts, instruction-level parallelism, and arithmetic intensity.

To improve the precision of our model we also take into consideration register spills. The *MIN* algorithm works well for code which has long basic blocks as shown in [10]. Since it only looks at the basic block level, it can insert spills and reloads inside loops which leads to unnecessary memory instructions. An improvement would be to generate spills at the function level as done by current compilers.

Algorithm 6.4 MIN algorithm

```
procedure MIN_ALGORITHM(DFG, k)
    W, S ← ∅
    for I ∈ DFG do
        R ← uses(I) − W
        W ← W ∪ R
        S ← S ∪ R
        LIMIT(W, S, I, k)
        l ← k − |defs(I)|
        LIMIT(W, S, next(I), l)
        W ← W ∪ defs(I)
        reload variables from R
    end for
end procedure
```

```
procedure LIMIT(W, S, I, m)
    sort(W, I)
    W, W' ← split(W, m)
    for w ∈ W' do
        if w ∉ S ∧ nextUse(I, m) ≠ ∞ then
            add a spill for w before I
        end if
        S ← S − w
    end for
end procedure
```

7 Validation

In this section we show how well the execution times predicted by the proposed performance model comply with the actual measured times.

In our experiments we used the NVIDIA GTX 460 GPU. Since this is the same card on which we performed experiments in Section 3 we use those values for instantiating hardware parameters. For examples of values for instructions we refer to Table 3.1. For global memory we use a latency of 500 cycles (Figure 3.2) and a bandwidth of 86.4 GB/sec (Figure 3.3). Since we are performing the simulation on a single SM, we scale the bandwidth accordingly. That is, we use a value of $86.4/7 = 12.34$ GB/sec. For shared memory we use a latency of 36 cycles (Figure 3.2) and for bandwidth we use the theoretical value of 4 bytes/cycle. The behaviour of the obtained formulas can be seen throughout the figures in Section 4.

Because we do not model caches, when performing measurements on the GPU we disable them. We do this by compiling with the flags `-Xptxas -dlcm=cs` which enables streaming behaviour. That is, at every cache query the results will be invalidated.

We perform several experiments which validate different aspects of our model. The experiments can be grouped into two categories. The first category tests fundamental properties of our model in isolation, while the second is designed to test the interaction between them. We compute precision by dividing the predicted time to the measured time. In this setup, good precision equates to a value close to one, while optimistic predictions are smaller than one and pessimistic predictions are greater.

As a general remark, input parameters are chosen such that the dynamic analyses performed by the underlying analysis system finish in reasonable time.

7.1 Fundamental benchmarks

The first experiment is designed to test the accuracy of the instruction delay formula (4.5). We do this by using a kernel similar to the one from Listing 3.4 which we used to benchmark the GPU. To simulate the same workload we wrap the kernel in a loop with 1024 iterations. Our tool reports an ILP of 1, and suggests a block size of 160 which corresponds to an evenly distributed workload across the SMs. Since there are no memory instructions we do not report any hotspots. We performed this benchmark using `mul` instructions and achieved a precision of 0.93 which validates our model.

The next experiment is designed to test a deficiency in our system: because we do not analyse binaries compiled by GPUs, instructions can differ. To illustrate this we consider the same kernel, but we use `madd` instructions encoded as `a = a * b + c`. The GPU compiler uses a single instruction to encode

this , while our compiler uses a `mul` followed by an `add`. In this case the predictions are pessimistic, as indicated by the 2.12 precision value. This is expected, because the program which we analyse contains roughly two times more instructions.

The last fundamental experiment is designed to measure the global memory effects. Again we use a similar kernel as the one for benchmarking (see Listing 3.1). The system identifies the two loads as the main memory hotspots. When comparing execution time, we obtain a precision of 2.04. The main culprit behind this is our assumption about equally distributed workload which translates to equally dividing the bandwidth between SMs. In practice, a single SM could reach the full bandwidth of the GPU.

7.2 Complex benchmarks

We tested our system with two other experiments which test the interaction between performance factors. In both cases, because of the large workload, the suggested block size was $(32, 32)$ which corresponds to the hardware limits.

To test coalescing effects we define a matrix transposition kernel. The matrices are linearised and each thread transposes a tile of the whole matrix. Listing 7.1 illustrates the main portion of the kernel. Only line 3 represents the actual kernel computation which is also executed on the GPU. Note that the access to `odata` is column wise, which implies that consecutive iterations do not access contiguous memory portions. On the GPU, this translates to bad coalescing. In this case our simulation suggests that the access to `odata` is more costly than that to `idata`. Regarding precision we obtain a value of 1.64 which further validates our approach.

```
1 for( y = 0; y < size_y; ++y){
2   for(x = 0; x < size_x; ++x){
3     odata[x * size_y + y] = idata[y * size_x + x];
4   }
5 }
```

Listing 7.1: Transpose kernel

Our last experiment is designed to test the interaction of all performance effects, including shared memory. To this end we use a more complex program – a 2D convolution filter. The algorithm process the pixels of a 2D image by combining them with a filter. Listing 7.2 shows the kernel code. We see that for each input pixel the filter is overlaid over the neighbourhood and an average value is computed. Because of space constraints we omit the out-of-bounds check performed along the image edges.

In this case the static analysis reports that the whole `filter` array and a portion of the `in_image` can be stored into shared memory. These shared opportunities fit into the capacity of the shared memory, thus no selection is involved.

Again notice that we have bad coalescing for `in_row` and `out_row`, because both are accessed column-wise. Because we have put `in_row` in shared memory, coalescing is not a problem but bank conflicts may appear. In this case, the results of our bank conflict analysis indicate that the access to `in_row` does not suffer any conflicts.

The precision which we obtained for this benchmark is 1.89 which again shows that our approach works. Regarding the predicted hotspots, our system indicates that the access to `out_row` is the most costly. This shows the benefits of shared memory because even though the accesses to `in_row` and `filter` are

Experiment	Precision	General remarks
mul	0.93	Good accuracy because of throughput
madd	2.12	Analysed code differs
memcpy	2.04	Memory bandwidth estimated for 1 SM
transpose	1.64	–
convolution	1.89	–

Table 7.1: Summary of validation experiments.

more frequent, the access to `out_row` is to global memory and is uncoalesced.

```
1  for (y = 0; y < image_height; y += 1){
2    for (x = 0; x < image_width * 3; x += 3){
3      unsigned char *out_row = out_image[y * image_width * 3];
4      int fx, fy;
5      int red,grn,blu;
6      for (fy = 0; fy < filter_height; fy += 1){
7        int py = y + fy - (filter_height / 2);
8        unsigned char *in_row = in_image[py * image_width * 3];
9        for (fx = 0; fx < filter_width; fx += 1){
10         int px = x + 3*(fx - filter_width / 2);
11         int coeff = filter[fx + fy * filter_width];
12         red += in_row[px + 0] * coeff;
13         grn += in_row[px + 1] * coeff;
14         blu += in_row[px + 2] * coeff;
15       }
16     }
17     out_row[x + 0] = red * filter_gain;
18     out_row[x + 1] = grn * filter_gain;
19     out_row[x + 2] = blu * filter_gain;
20   }
21 }
```

Listing 7.2: Convolution kernel

Conclusions

The experiments performed in this section validate our model. As mentioned, our goal was not cycle-accurate execution but rather obtaining relative speedup and identifying bottlenecks. A summary of our experiments is given in Table 7.1. In general we obtain good results for calibration benchmarks, because our model is derived from their behaviour. In the memory case, we lose precision because of the assumptions which we made. Furthermore, we validated our model on complex kernels. Again when memory instructions are introduced we can see that precision decreases, and we become more pessimistic.

The quality of our results needs to be judged also in the context of *vfEmbedded*– the system in which we integrated our work. Since it employs an interactive compilation workflow, the error of our model is

acceptable. Furthermore, data-parallel kernels usually achieve a speedup of 100× on the GPU. Even if we are 10 times more pessimistic, our tool would still report a 10× speedup which will determine the user to parallelize the kernel.

Regarding bottlenecks, our system correctly classified the instructions which caused the GPU to idle. Since the kernels which we tested already achieved the maximum performance no real bottlenecks were found. We believe this feature will be useful for more complicated kernels, which employ synchronization or have *map-reduce* patterns.

8 Related work

Performance prediction has been extensively studied in the context of parallel and sequential systems [5, 20, 30, 34] for the past 30 years. Since GPUs have only recently emerged as a viable parallel platform there is little work about predicting performance for them [1, 2, 12, 19].

Most of these models target the previous generation of GPUs: the GF80 and GF200. Although the same performance factors are present, their impact is different than in our architecture. For example, on the GF80 instruction-level parallelism was not exploited and memory requests did not pass through a cache hierarchy.

Furthermore, all the models use as input native CUDA kernels and do not deal at all with parallelization overhead. In our case, as input we have a sequential program which has to be transformed to run on the GPU. Since the program is compiled for sequential execution, instructions can differ to the point where the two programs cannot be compared.

These factors make the comparison of results impossible. Thus we settle on presenting the other approaches and compare the techniques. In what follows we group related work into two categories: analytical models and GPU simulators.

8.1 GPU analytical models

As we have seen, an important part of our research consist of a GPU analytical model. The current GPU analytical models [12, 19, 1, 35] target the CUDA architecture and propose techniques which address similar issues as those described in Section 3.

One of the first approaches to define a GPU analytical model is presented in [19]. The authors propose a solution which combines several known models of parallel computation: PRAM, BSP, and QRQW. Their model is simple, thus efficient to compute. The main drawback is that they do not model factors like shared memory bank conflicts, and global memory coalescing. Also, they assume that warps are always ready to execute, which corresponds to achieving maximum performance.

In [12] Hong and Kim develop a technique which models the number of memory requests by taking into account memory bandwidth and the number of parallel threads. They perform validation using a set of micro-benchmarks and report an absolute error between 5.4% and 13.3%. Their main contribution is the development of two metrics, *memory warp parallelism (MWP)* and *computation warp parallelism (CWP)*, which characterize the data-, respectively instruction-, level parallelism of a whole GPU kernel.

Baghsorkhi et al. [1] propose the first model which takes into account branch divergence and bank conflicts. The model uses techniques from program analysis and symbolic evaluation to make accurate

predictions. The main contribution is the fact that the model can statically determine loop bounds, data access patterns and control flow patterns for a restricted class of kernels. Another contribution is the characterization of GPU parallelism in the context of branch divergence.

We see several major differences between these studies and our work. That is, instead of trying to build an analytical model based on an abstraction of GPU architecture and then verifying the model by benchmarks, we adopt the reverse strategy. We first design benchmarks (Section 3), observe the benchmark results, and then derive a corresponding analytical model respectively for instruction pipeline, shared memory, and global memory (Section 4). This approach allows us to observe and consider only the architecture and programming factors that are most relevant to performance.

Zhang and Owens [35] use an approach that is similar to ours. They design a set of micro-benchmarks from which they derive the simple throughput models. In contrast to our simple simulation, to estimate time they gather statistics using an actual GPU simulator, Barra [6]. This makes their approach more accurate but more resource demanding. Furthermore, another difference is the fact that they do not model bank conflicts.

8.2 GPU simulators

Another successful approach to predicting GPU performance is by using simulators [6, 17, 16, 2]. The majority of simulators target the CUDA platform and perform simulation either of PTX or native GPU code. NVIDIA's *Parallel Thread Execution* (PTX) [27] is a virtual instruction set architecture with explicit data-parallel semantics which is suitable for the CUDA architecture.

We present the results in order to compare techniques which could be used to improve our simulation algorithm. We mainly focus on aspects related to the GPU architecture and the execution model.

Collange et al. [6] implement Barra, a GPU functional simulator for NVIDIA's Tesla GPUs. Barra is implemented using UNISM, a modular simulation framework, and uses as input CUDA binary kernels which are usually executed by the GPU. The advantage of this approach is that it provides cycle-accurate performance estimations and allows the user to monitor all the GPU activities. Because it uses emulation the running times of the simulation can be quite long. Since the binary instruction set can be modified from generation to generation, this approach is not very flexible. Calibration is done in a similar manner with our benchmarks.

Kerr et al. [16, 17] introduce another GPU simulator framework called Ocelot. The framework provides an emulation and compilation infrastructure that implements the CUDA runtime API. At the core of the simulator lies a virtual machine which emulates PTX instructions. By taking this approach the authors allow not only the emulation of CUDA kernels, but also their translation to different architectures. Besides predicting GPU performance, Ocelot computes control and data dependencies by gathering instruction and memory traces.

The most complex GPU simulator to date, *GPGPU-Sim*, is developed by Bakhoda et al. in [2]. Their main goal is to provide a tool which allows users to experiment with different GPU architectures and easily explore the design space. GPGPU-Sim achieves this by emulating the PTX instruction set and by closely following the CUDA architecture. Also, the simulator features cycle-accurate performance predictions and allows the changing of several architectural details such as the interconnection network between the SMs.

9 Conclusions

Today, programmers are unable to effectively exploit all the computational power provided by the GPU. This is mainly because the programming models are not mature enough, thus the programmers must carefully reason about their programs and manually optimize them. Having to always try optimizations and run benchmarks to see their effect incurs a high development cost and requires intimate knowledge of the underlying architecture.

In this thesis we addressed this problem by creating a parametrized analytical model which allows programmers to estimate execution time and identify potential bottlenecks in their programs. Besides prediction, our tool also suggests values for parameters that influence performance.

Our system has as input a data-parallel sequential program written in ANSI C99 and is able to provide performance insights which allow the user to quantitatively assess if the program is worth parallelizing to the GPU. Our tool answers three main questions: If I convert this program to the GPU what performance should I expect? What values should I use for parameters which influence performance? If the program runs slow, where should I start optimizing.

The system is integrated with an existing production ready analysis tool *vfEmbedded* [33] which parallelizes sequential programs. Our system builds upon its existing GPU mapping capabilities by adding a performance estimation module.

At the heart of our system lies a parametrized GPU model. The model is developed by characterizing the GPU behaviour through benchmarks and deriving formulas which measure performance. We target three major GPU components: the instruction pipeline, the shared memory and the global memory systems. In order to compute values for the models parameters we employ three techniques. Benchmarks are used to derive values for hardware parameters such as throughput and latency. For static parameters like instruction-level parallelism we employ program analysis. Lastly, for dynamic parameters like thread-level parallelism we use a coarse-level simulation.

By using simulation we are able to pinpoint potential bottlenecks such as memory hotspots. Another major part of the model deals with predicting the overhead introduced by parallelization.

To our knowledge this is the only analytical model developed for the Fermi architecture. Although we only calibrated our model for a specific GPU, we believe that our techniques and model apply to other GPUs. To this day, the performance factors of GPUs are more or less the same. Because we employ benchmarks to create the model, a similar methodology could be applied to other GPUs in order to capture their behaviour.

Future work

This thesis represents the first step towards performance prediction for GPUs. Our work has several limitations which we hope to address in future work.

First, an important GPU component that is not modelled by our system is the cache hierarchy. Usually caches improve performance of memory operations which leads to increased bandwidth. Because we employ simulation, we can extend the infrastructure to include a cache subsystem. In our current implementation we have already done so but we where unable to correctly calibrate them.

Another important subject relates to instructions which require synchronization. Because the infrastructure deals with pure data-parallel programs, synchronization was not necessary. Another important class of GPU kernels are those which contain *map-reduce* patterns. The reduce part of those kernels requires synchronization. To this end, in our implementation we have added a *synchronization* type of operation which models such instructions. In our simulation, warps with such instructions get synchronized at the block level. Because this is not currently used by any program, we chose to omit it.

Future work includes also calibration and validation for other CUDA or AMD GPUs. The GPU which we use for benchmarking and validation, the GTX 460, is a mid level product which is not designed specifically for GPGPU programming. To this end, we plan to test on GPUs which specifically target GPGPU programming such as the Tesla C2050. Since NVIDIA is not the only vendor which targets GPGPU, we also plan to see how well we can adapt our model to other GPUs, such as those from AMD.

Bibliography

[1] S.S. Baghsorkhi, M. Delahaye, S.J. Patel, W.D. Gropp, and W.W. Hwu. An adaptive performance modeling tool for GPU architectures. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel computing*, pages 105–114. ACM, 2010.

[2] 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. IEEE, 2009.

[3] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5:78–101, June 1966.

[4] E.A. Brewer, C.N. Dellarocas, A. Colbrook, and W.E. Weihl. Proteus: A high-performance parallel-architecture simulator. In *Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, page 248. ACM, 1992.

[5] M.J. Clement and M.J. Quinn. Analytical performance prediction on multicomputers. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, page 894. ACM, 1993.

[6] S. Collange, D. Defour, and D. Parello. Barra, a modular functional GPU simulator for GPGPU. 2009.

[7] K.D. Cooper, P.J. Schielke, and D. Subramanian. An experimental evaluation of list scheduling. *TR98*, 326.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, 1991.

[9] M.D. Ernst, J.H. Perkins, P.J. Guo, S. McCamant, C. Pacheco, M.S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69:35–45, 2007.

[10] J. Guo, M. J. Garzaran, and D. Padua. The power of belady's algorithm in register allocation for long basic blocks. In *The 16th International Workshop on Languages and Compilers for Parallel Computing*, 2003.

[11] S.D. Hammond, G.R. Mudalige, J.A. Smith, S.A. Jarvis, J.A. Herdman, and A. Vadgama. WARPP: a toolkit for simulating high-performance parallel scientific codes. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, pages 1–10. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009.

[12] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. *ACM SIGARCH Computer Architecture News*, 37(3):152–163, 2009.

[13] E. Ipek, B.R. De Supinski, M. Schulz, and S.A. McKee. An approach to performance prediction for parallel applications. *Euro-Par 2005 Parallel Processing*, pages 196–205, 2005.

[14] I.K. Isaev and D.V. Sidorov. The use of dynamic analysis for generation of input data that demonstrates critical bugs and vulnerabilities in programs. *Programming and Computer Software*, 36(4):225–236, 2010.

[15] S.A. Jarvis, D.P. Spooner, H.N. Lim Choi Keung, J. Cao, S. Saini, and G.R. Nudd. Performance prediction and its use in parallel and distributed computing systems. *Future Generation Computer Systems*, 22(7):745–754, 2006.

[16] A. Kerr, G. Diamos, and S. Yalamanchili. A characterization and analysis of ptx kernels. 2009.

[17] A. Kerr, G. Diamos, and S. Yalamanchili. Modeling gpu-cpu workloads and systems. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 31–42. ACM, 2010.

[18] M. Kim, H. Kim, and C.K. Luk. Prospector: A Dynamic Data-Dependence Profiler To Help Parallel Programming.

[19] K. Kothapalli, R. Mukherjee, M.S. Rehman, S. Patidar, P.J. Narayanan, and K. Srinathan. A performance prediction model for the CUDA GPGPU platform. In *High Performance Computing (HiPC), 2009 International Conference on*, pages 463–472. IEEE, 2010.

[20] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 2–13. ACM, 2004.

[21] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pages 23–23. Usenix Association, 1996.

[22] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[23] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, corrected edition, 2004.

[24] Nvidia. *Fermi Compute Architecture Whitepaper*.

[25] Nvidia. *CUDA C Best Practices Guide*, 2010.

[26] Nvidia. *CUDA C Programming Guide*, 2010.

[27] Nvidia. *PTX: Parallel Thread Execution ISA Version 2.1*, 2010.

[28] Nvidia. *cuobjdump – CUDA dissasembler*, 2011.

[29] D. Ofelt and J.L. Hennessy. Efficient performance prediction for modern microprocessors. In *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 229–239. ACM, 2000.

[30] Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*, page 12. IEEE, 2006.

[31] H.G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[32] S. Sahni and V. Thanvantri. Performance metrics: Keeping the focus on runtime. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 4(1):43–56, 2002.

[33] VectorFabrics. vfEmbedded. http://www.vectorfabrics.com/products/vfembedded.

[34] R.C. Whaley and D.B. Whalley. Timing high performance kernels through empirical compilation. In *Parallel Processing, 2005. ICPP 2005. International Conference on*, pages 89–98. IEEE, 2005.

[35] Y. Zhang and J.D. Owens. A quantitative performance analysis model for gpu architectures. In *Proceedings of the 17th IEEE International Symposium on High-Performance Computer Architecture (HPCA 17)*, 2011.