Utrecht University

Reducing the overhead of data transfer in data-parallel programs

Master thesis

Paul Visschers

Supervisors:
Prof. Dr. S. Doaitse Swierstra (Utrecht University)
Ir. Maurice Kastelijn (Vector Fabrics)

March 28, 2012

Abstract

Graphics processing units (GPUs) have evolved to allow for general purpose many-core programming. Most GPUs have their own separate memory, requiring that input data be transferred to the GPU before running the program and transferring the results back to the CPU upon completion. This transfer of data imposes significant overhead that we would like to reduce. A possible solution is to split up a program into many smaller pieces, called tiles, and then setting up a pipeline that overlaps data transfers with program execution (on the GPU). This can reduce the overhead of data transfers significantly.

We examine the effectiveness of several variations of this tiling/pipelining transformation for a common class of programs. We introduce a model that predicts the run time performance of these transformations ahead of time, as well as a recipe that guides users in transforming their code. We show that one of these transformations provides a good speed-up, which for some problems is over 2 times faster than versions that do not overlap data transfer and program execution. Finally we show that our model accurately predicts the run time of programs using this transformation.

Contents

Chapter 1

Introduction

The days that graphics processing units (GPUs) were only used for graphics are over. These days they can be used for general purpose computation. As GPUs have hundreds of cores, this can lead to programs that are more than a hundred times faster than their CPU equivalents, that only have a handful of cores. Unfortunately the architecture of GPUs restrict what types of programs can actually benefit from running on the GPU.

Because many GPUs have their own memory that is separate from the memory of the CPU, we need to transfer data to and from the GPU if we are to do any calculations on it. The most common pattern is to transfer the input to the GPU, then perform the calculations there and finally transfer the output back to the CPU. These data transfers impose a significant overhead.

Luckily modern GPUs can transfer data and perform calculations concurrently, some can even transfer data upstream and downstream at the same time. This allows us to create a pipeline that does all three steps in parallel, which can effectively hide the execution time of the shorter ones. But in order for us to do this, we need to first split the program into smaller pieces called tiles. This thesis implements several variations of this tiling/pipelining method and shows how effective each is in reducing the overhead of data transfers for several instances of a common program class.

To increase the usefulness of the tiling/pipelining optimizations we create an accurate model of their execution. This allows us to predict how well each optimization will perform for a given program and targeted hardware. Users can use this to quickly do a cost benefit analysis before they invest time in implementing the optimization. It also enables users to get insight on run times on hardware they do not have access to. Note that this thesis only focuses on a single class of programs. The insights gained are transferable to other classes.

To further increase usability, we present a convenient recipe that guides users in converting their sequential programs to versions that use the tiling/pipelining optimizations. This recipe makes the conversion about as easy as converting the sequential program into a simple GPU-using implementation. To make the recipe as easy as possible, it depends on a custom-made library that performs

all the complex operations.

Chapter 2

Background

2.1 Vector Fabrics

This thesis is part of an internship at the company Vector Fabrics in Eindhoven. It is a start-up that is developing tools that help programmers analyze their programs and that make suggestions for parallelization. The tool predicts the performance of a suggested parallelization scheme and gives a recipe that the programmer can follow to implement the scheme. The tool is deployed online and continues to be improved. While until recently the focus was only on multi-core CPUs and embedded systems, the company has started to research ways of incorporating parallelization of programs to the many-core GPU.

2.2 GPGPU

Central processing units (CPUs) have never been well suited for graphics processing. Processing one pixel is often independent of processing another, and different phases can be pipelined so that work on the next image can begin before the current one is done. All this potential parallelism is lost on the CPU, and this led to the creation of specialized hardware called the graphics processing unit (GPU). Initially this hardware implemented each phase of the graphics pipeline directly, but as the complexity of rendering grew this changed. Operations to determine the color of a pixel became programmable, these programs were called pixel shaders. Later other types of shaders were introduced. As the years passed even more of the complexity was put into programmable software and the hardware became more and more general. In the last few years the hardware has become so general that it is now possible to do general purpose GPU (GPGPU) calculations.

Since graphics processing is highly parallelizable, GPUs have evolved to contain many cores (contemporary ones have around 500 cores) and can cheaply spawn millions of threads. These threads work in a single instruction, multiple data (SIMD) way, meaning that each thread is doing the same work, but on a

different part of the data. CPUs on the other hand have only a handful of cores and spawning threads is costly, but each core is truly independent.

In general it is not easy to determine what programs benefit from running on the GPU. It depends on many factors, including the parallelizability and complexity of the program itself and the performance of various pieces of hardware. Also running on the GPU introduces significant overhead.

Inherently sequential algorithms run slower on the GPU, as only a single core can be used; a single GPU core is generally slower than a single CPU core. Also problems that are parallelizable, but have a lot of control flow do not benefit as much from running on the GPU, since if some threads do some extra work because an if-statement was true for them, all other threads have to wait for those to finish their extra work before all of them can continue in unison. Problems that have lots of independent, computationally intensive and monotone work benefit fully from the GPU's architecture. But these problems are rare; in practice most problems have some limitations that make them harder to parallelize. Yet even with these limitations it is often possible to speed-up algorithms significantly by using a GPU.

2.3 Platforms

When it comes to GPU hardware, the currently most popular platform is a separate GPU that has its own dedicated memory and a CPU that controls the GPU. The CPU is often called the host, while the GPU is called the device. As the GPU has separate memory, programs are usually executed in an RPC style, which is explained in section 2.5. The CPU and GPU are connected by a PCIe bus. The currently standard PCIe 2.0 bus has a bandwidth of roughly 8 GB/s, so data transfer can be a real bottleneck. The highest performance GPUs currently all have dedicated memory, so data transfer is an important aspect in high performance computing. The most popular GPUs with dedicated memory at the time of writing are in the AMD Radeon HD 6xxx series and the NVIDIA GeForce 500 series. NVIDIA even has a GPU designed specifically for high performance computing, instead of graphics rendering, called the Tesla.

There are also GPUs that are integrated into the motherboard or even into the chip set. These usually do not have their own dedicated memory but are tied into the memory that normally belongs to just the CPU. Even though these GPUs physically use the same memory space as the CPU, they still use separate logical address spaces. This means that the RPC style can still be used, but these GPUs benefit especially from a technique called zero-copy, which is explained in section 2.5.3. Intel integrates GPUs onto its motherboards under the name Graphics Media Accelerator (GMA). GPUs that are integrated on the CPU chip set are very new and the pioneer is AMD with its Fusion.

At the moment the integrated GPUs are mostly useful because they are cheap and energy efficient. This makes them perfect for embedded systems and budget PCs. In the future the separate address spaces will probably be joined into a single address space, which should make them faster and easier to use

when it comes to memory operations, likely even faster than dedicated GPUs. However dedicated GPUs will probably remain much faster when it comes to computation. But even if GPUs end up becoming completely integrated with CPUs, the issue of data transfer may still be relevant in other fields, such as distributed computing.

2.4 Programming models

Programmers can write code for the GPU by using a suitable programming model. The Open Computing Language (OpenCL) [11] is one that can be used to run computations on general heterogeneous hardware platforms. It addresses a wider range of environments than just CPU/GPU environments. Because it is an open standard, it has been widely adopted by various hardware vendors.

NVIDIA supports OpenCL on their GPUs, but is also developing its own proprietary API called Compute Unified Device Architecture (CUDA) [15]. Unlike the more general OpenCL, it is focused solely on CPU/GPU platforms and it cannot be used for other types of heterogeneous platforms. Because it is proprietary, it works only with NVIDIA GPUs. This narrower focus has its perks though, as the programming model itself is much more convenient and generally tends to be ahead of OpenCL. It is worth noting that conceptually, both CUDA and OpenCL operate in roughly the same way when it comes to GPU programming.

Besides the programming models, both AMD and NVIDIA provide a software development kit (SDK). These kits provide libraries with common functionality, tools for debugging and profiling and other development help. AMD only supports OpenCL, so its Accelerated Application Processing SDK (APP) [1] is based on that. NVIDIA's GPU Computing SDK combines CUDA and OpenCL support. Both offer roughly the same functionality.

This thesis focuses on the CUDA programming model. Consider the following sequential program that for every element in a two-dimensional array (stored physically in a single one-dimensional array):

```
for{y = 0; y < size.y; y++) {
  for{x = 0; x < size.x; x++) {
    int a = in[idx2(x, y, size)];
    if(x < size.x − 1) {
      a += in[idx2(x + 1, y, size)];
    }
    out[idx2(x, y, size)] = a;
  }
}
```

If we convert this program into a CUDA version, we get the following code:

```
// The work that is performed on each element is extracted to a kernel.
__global__ void kernel(dim3 size, float *in, float *out) {
  // Each thread has its own x, y and z coordinates associated with it, these
```

```
  // special values are used to retrieve these coordinates.
  int x = blockDim.x * blockIdx.x + threadIdx.x;
  int y = blockDim.y * blockIdx.y + threadIdx.y;

  // The body of work is surrounded by an if-statement that makes sure that
  // if more threads than necessary are spawned, the extra threads do not
  // access memory that is out of bounds.
  if(x < size.x && y < size.y && z < size.z) {
    int a = in[idx2(x, y, size)];
    if(x < size.x - 1) {
      a += in[idx2(x + 1, y, size)];
    }
    out[idx2(x, y, size)] = a;
}

// Memory is allocated on the GPU (device).
float *devIn, *devOut;
cudaMalloc(&devIn, size * sizeof(float));
cudaMalloc(&devOutput, size * sizeof(float));

// Input data is copied from the CPU (host) to the GPU.
cudaMemcpy(devInput, input, size * sizeof(float),
  cudaMemcpyHostToDevice);

// The kernel is executed on the GPU. The gridSize and blockSize are set so
// that enough threads are spawned.
kernel<<<gridSize, blockSize>>>(size, devInput, devOutput);

// Output data is copied back from the GPU to the CPU.
cudaMemcpy(output, devOutput, size * sizeof(float),
  cudaMemcpyDeviceToHost);

// GPU memory is freed.
cudaFree(devInput);
cudaFree(devOutput);
```

In this code the body of the two nested loops is extracted and put into a function that is to be run on the GPU called a kernel. For every iteration of the original loop, a thread is spawned on the GPU that executes the kernel. The coordinates of the array element to work on are retrieved by special values provided by CUDA. Once the kernel is defined, the loops themselves have to be replaced with CUDA calls that allocate memory on the GPU, transfer input data to the GPU, send the kernel to the GPU and run it in many threads, transfer the output back to the CPU and finally free the memory on the GPU.

2.5 Data transfer

It is a necessary part of GPU computation to get the input data to the GPU beforehand and to get the output data back to the CPU memory afterwards for further use. The currently most common approach to running a computation on the GPU resembles a remote procedure call (RPC). One sends the input data to the GPU; send the kernel; runs it and send the output data back to the CPU. A CUDA example of this style is in the previous section. In normal RPC calls the calling system waits for the called procedure to return. While this is an option with GPUs, it is usually much faster to use asynchronous calls. This means that the CPU does not wait for a call to the GPU to finish, but instead continues immediately. The GPU queues the work it receives and special synchronization calls can be used to have the CPU wait for the GPU to finish a certain task before continuing. We refer to this style as asynchronous RPC (ARPC).

Normally, the operating system (OS) virtualizes the host memory. It maps this virtual memory to pages. Each page is a block of memory (usually 4 KiB) and may physically reside in the memory or on the hard drive. When data in pages on the hard drive are accessed, these pages are automatically put into the physical memory where they can be used. This virtualization and paging means that if we want to transfer data to the GPU, this data may reside on the hard drive instead of in memory. The OS must be involved for every data transfer to check for this and swap in any necessary data.

2.5.1 Direct memory access

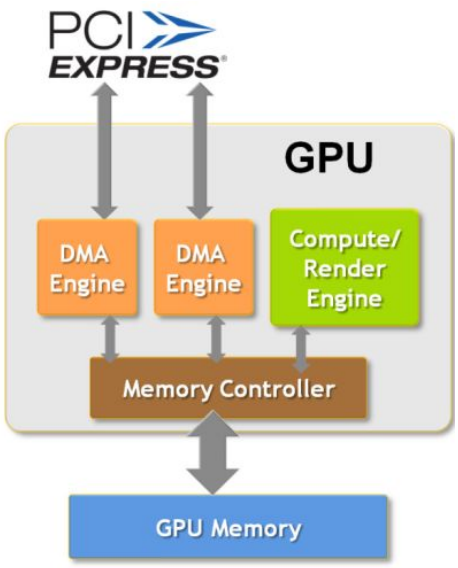

Figure 2.1: A GPU with two copy engines. (Image by NVIDIA [14].)

Most modern GPUs have a copy engine that uses direct memory access (DMA) to bypass the CPU when transferring data. This is much faster than normal data transfer and has the added benefit that the CPU is free to do something else. Furthermore, the copy engine is a separate unit from the kernel engine that runs the kernels on the GPU, which allows concurrent data transfer and kernel execution. Some GPU cards have two separate DMA engines, such as the NVIDIA Quadro that is modeled in figure 2.1. This architecture allows concurrent data transfer in both directions, as well as concurrent kernel execution.

2.5.2 Page-locking

In order to use the DMA feature it is required that the operating system locks the pages that contain the memory to transfer. This page lock ensures that the data is swapped in and remains at the same physical memory location, so that the memory can be accessed by a co-processor (in this case the GPU) without interference of the operating system. It is not a good idea to page-lock a lot of memory, as this severely limits the amount of physical memory available for swappable memory. This leads to a lot of swapping and decreases performance significantly.

2.5.3 Zero-copy

In addition to the ARPC style, there is another way to get data onto the GPU which is called the zero-copy method. In this method, the DMA engine is given direct access to a page-locked part of the host memory like before, but this data is not copied to the GPU memory explicitly. Instead the kernel is run and it uses the host memory directly. Data is transferred as needed by the kernel. This method is especially useful for integrated GPUs that run directly off of the host memory, but can also be effective for problems where each piece of data is only used once. If the GPU has dedicated memory and data items are used often, an explicit data transfer is usually faster. While zero-copy is an interesting topic, it falls outside the scope of this thesis.

2.6 Pipelining and tiling

Having one or two separate copy engines and a separate kernel engine allows us to set up a pipeline. Conceptually, a pipeline appears when a complex product is continuously created in several steps and each step is performed by an independent operator. To illustrate, consider the production of bread. The wheat farmer grows wheat, then the miller turns that wheat into flour and finally the baker bakes bread using the flour. Since the farmer, miller and baker are independent entities, the farmer can grow new wheat while the miller is still working on the old wheat and the baker is using flour created out of even older wheat. This means that even though the baker depends on the miller and the

miller depends on the farmer, all three can work continuously once they get going. Of course when starting up only the farmer is working, as the others are waiting for resources. The same happens when shutting down; the baker has work remaining while the other two have already finished up.

This situation is very similar to running kernels on the GPU. There are also several independent operators: the kernel engine and (ideally) two copy engines. Each does a single step in the process: copying the data to the GPU, running the kernel and copying the data back to the CPU. If there was a continuous stream of independent kernels to run, we could be copying back results from one kernel while the next kernel was being run and even copy in data that the second next kernel will be needing. Unfortunately, in most cases a program contains only a single kernel or multiple dependent ones. Luckily it is possible to get the benefits of pipelining on a single kernel with tiling.

When executing a kernel on the GPU many threads are spawned that all perform the same work, but on different data. We can actually divide up these threads into groups that we call *tiles*. We execute the kernel multiple times, each time for only the threads that are part of a single tile. These kernel executions are independent, so they can be pipelined. Dividing these threads into tiles, a process we call tiling, is non-trivial in general. The key to doing it is the access pattern of the kernel. For example, a thread with thread ID n takes the n-th element in an array, calculates a result based on that value and stores it in the n-th element of another array. This is a simple access pattern, many are more involved. When the threads in a kernel are divided up into tiles, each tile needs the union of data required by the threads that are in it. Figure 2.2 shows the different schedules for the RPC style, just tiling and both tiling and pipelining.

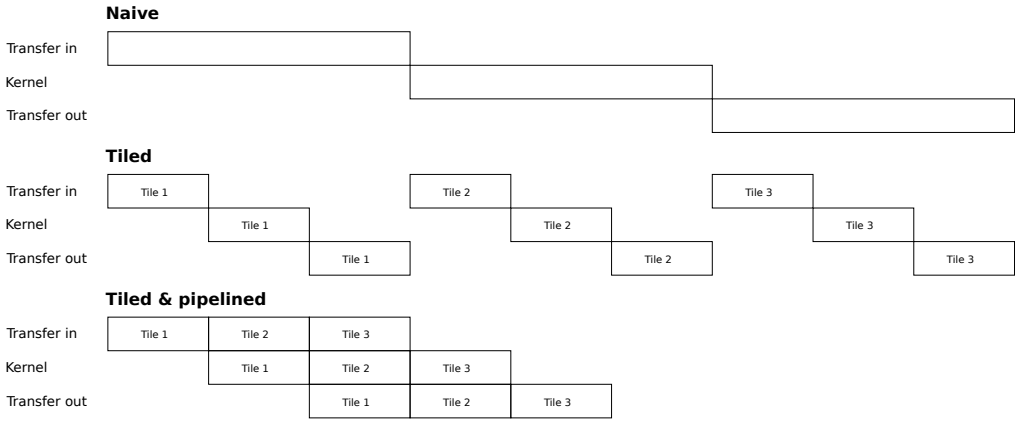

Figure 2.2: The data transfer and kernel execution schedules for the RPC style (labeled *naive*), employing only tiling and employing both tiling and pipelining.

2.7 Tiling of program classes

It can be difficult to determine what tile shape is most effective for any given program. To illustrate this the following sections describe several elementary programs and how they can be tiled.

2.7.1 Map and zip

Map functions are very easy to tile. They take an array and perform a function on each element individually, resulting in a new array. The n-th kernel thread only works on the n-th array element, so there is a direct correspondence. This means that if a tile for threads n through m is created, that tile needs the data of elements n though m.

Zip functions are slightly more complex, but essentially as easy to tile as map functions. Instead of having a single input array, zip functions have two or more input arrays. The direct correspondence between threads and array elements remains. An example of a zip function is vector addition, which adds up each number of the first array with the corresponding number in the second array. There are also unzip functions that have multiple outputs. The two can also be combined, where a function has multiple inputs and outputs.

2.7.2 Stenciled map and zip

Stenciled map functions are harder to tile than the regular map functions. A stencil of a certain element is the combination of that element together with some of its neighbors. So for these functions, every thread n accesses element n and its neighbors; what and how many neighbors are used depends on the algorithm. An example is the moving average, where each element in the output is the average of the corresponding input element and some number of neighbors. If we choose to take 2 neighbors on each side, a tile for threads n through m would need the array elements $(n-2)$ through $(m+2)$. This extra data is called *padding*. Padding leads to extra data transfer, which reduces the performance gained by pipelining. Padding also gives difficulties when creating tiles that are on the edge of the array, as some of the neighbors are missing in this case. It is possible to do a stenciled map on a two- or three-dimensional dataset. In this case padding in all directions is required, leading to even more complexity and transfer overhead. Many 2D and 3D image filters are instances of stenciled map. Stencils can also be included in zip/unzip functions.

2.7.3 Reduce and scan

Reduce functions convert an array of values to a single compounded value. For example the sum function adds up all numbers in an array. This function can be implemented on the GPU by launching a kernel that creates a thread for every even element. Each thread calculates the sum of that element and the one preceding it. Then another kernel is launched with threads for every fourth

element that adds up that element to the one two spots before it. Now every fourth element holds the sum of that element and the three preceding elements. Another kernel is launched for every eighth element, then for every sixteenth and so forth until the sum of all numbers ends up in the final element[1]. This process is shown in the top half of figure 2.3.

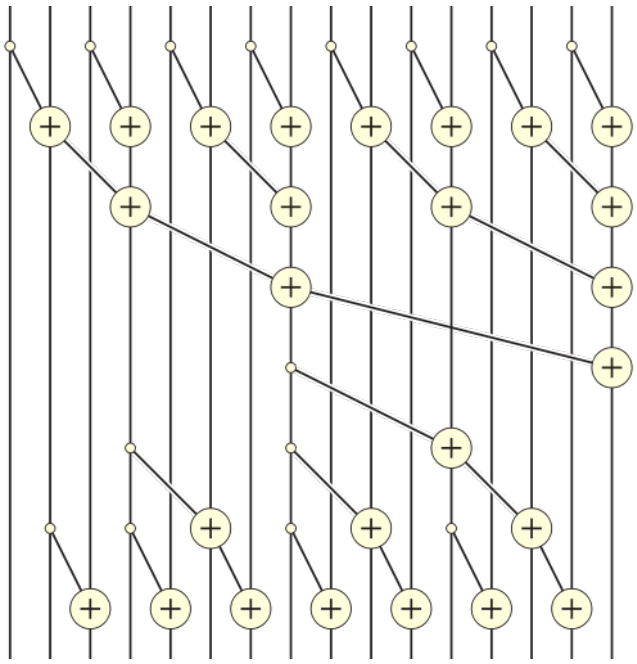

Figure 2.3: Parallel implementation of scan/prefix sum with 16 inputs. (Image by David Eppstein.)

Scan functions [6, 18] are like reduce functions, but instead of giving a single result (e.g. the sum of all elements) they return an array where each element holds the result up to that position. In the case of our sum example each element n holds the sum of elements 0 through n. This scan version of sum is called prefix sum. The first half of the parallel implementation of scan is the same as that of reduce. Once that is done, the values in the array are used to fill in the prefix sum for every element. This process is shown in the second half of figure 2.3. In the figure, adding the value of element 8 to the value of element 12 yields the prefix sum for the latter. After this is done, every 4th element holds their prefix sum and can be added to the element two spots after it. This pattern is repeated until all elements hold their prefix sum. It is easy to see how this process works for bigger arrays.

Tiling reduce and scan functions can be done by first splitting up the array as normal. But if we then run the function on each tile naively, we do not get

[1]For convenience and clarity, we assume that the length of the array is a power of 2.

the correct results. This is because each tile only calculates the local results, i.e. a tile for elements n through m contains the compounded value(s) of only those elements. We want them to contain those of elements 0 through m instead. By adding the compounded value of the previous tile to the current tile, this problem is solved.

While this tiling scheme works, it is not very efficient. This is because the middle part of scan functions (and the last part of reduce functions) has steps that only spawn a handful of threads. Contemporary GPUs have roughly 500 cores, so most of them are unused at this stage in the program (i.e. occupancy is low). By tiling, this middle part happens in every tile. This leads to more core inactivity than with a normal implementation. Whether the benefits of pipelining make up for this is hard to say just from code inspection. There may be more involved tiling schemes that eliminate this problem, but those will be even more specific to this small class of programs.

2.7.4 N-body simulation and matrix multiplication

An n-body simulation calculates the gravitational pull of particles; since every particle affects every other particle, this is a hard problem to tile. In one step of the simulation, for every particle a new position and velocity is calculated based on the location and mass of all the other particles. We break up the input into tiles as we did for the map/zip class and put those tiles in the pipeline. While working on a tile on the GPU, the data of every other tile is required. So those are presented to the GPU by pipeline as well. This leads to a lot of data transfers; for N tiles we get N^2 data transfers to get the data to the GPU as each tile is sent N times.

If the GPU has ample memory space it is possible to improve this behavior. Instead of copying over the old tile when transferring a new one to the GPU we allocate enough memory to hold the entire input. This way we can use the pipeline to fill in this data as we go. This means that calculation can still start right after the first tile is loaded into the GPU memory, but no tile is transferred more than once. If the GPU does not have enough memory to store the entire input, a hybrid can be used to minimize data transfers. As these tiling schemes are quite complex, it is difficult to predict whether they will actually speed up the program.

Matrix multiplication has an access pattern that makes it equally difficult to tile. For each element (x, y) all data in row x of the first matrix and all data in column y of the second matrix is required. While this pattern is more involved than that of the n-body simulation, it has the same basic problem that data in tiles is required more than once. The same optimization can be applied, given enough GPU memory.

2.8 Multiple kernels

Up until now we have only considered pipelining kernels by tiling them. But if we have multiple sequential kernels that each process the output of the preceding one, making a separate pipeline for each is inefficient. This is because for each kernel, we transfer data to the GPU and then transfer it back to the CPU. We can in fact leave the output of one kernel on the GPU and feed it straight into the next kernel. If the access patterns of the kernels are similar enough, we can merge the kernels themselves so that we transfer data to the GPU; run the first kernel; run the second kernel and finally transfer the output back. This can then also be tiled and pipelined. If the access patterns are very dissimilar, this is harder and may not even be viable. Note that although this is an important issue, it is outside the scope of this thesis.

Chapter 3

Research questions

3.1 Problem statement

When computing on the GPU, it is necessary to first transfer input data to it and transfer output data back to the CPU when the computation finishes. These data transfers impose a significant overhead. This thesis investigates how to use tiling and pipelining to overlap data transfer and computation in order to reduce/eliminate this overhead. Furthermore, this thesis is geared towards integrating this optimization into the tools developed at Vector Fabrics.

3.2 Challenges

One of the challenges in this thesis is to determine what optimizations actually lead to increased performance. We try various schemes to see what works best under what conditions. The second challenge is to determine the run time of an optimized program on a chosen set of hardware without actually needing to run on this hardware. We create a detailed model model that predicts these run times. These predictions help determine what the best optimization is and what speed-up to expect for any particular combination of program and platform. Once an optimization is chosen the input program needs to be transformed. A recipe will instruct how to do so by hand, in several easy steps. To simplify the recipe and to limit the amount of boilerplate code, a library is presented that implements most of the common code.

3.3 Program class

As was explained in section 2.6, there are many classes of programs that can be tiled. However it is not easy to make a general system that works for all of those classes. This thesis focuses exclusively on the stenciled map/zip class described in section 2.7.2. This class is basic enough to serve as a good starting

point and it is interesting as the stencils make tiling non-trivial. The following algorithms are instances of this class and serve as test cases:

Moving average A one-dimensional program that for each element, calculates the average of that element and any chosen number of its neighbors. This is often used in economics and other fields to show longer term trends.

Emboss A two-dimensional image filter that is used to convert images. The result indicates where in the image contrast is strong. For every element, it takes the sum of the direct neighbors on the bottom right and subtracts the sum of the neighbors on the top left. This image filter is ubiquitous in photo manipulation software.

Jacobi stencil A stencil algorithm that is used to solve Laplace's equation. For this thesis, a three-dimensional implementation is used to complement the other programs. For every element, this program takes the average of its direct neighbors minus the element itself. Laplace's equation is used to model electromagnetism, gravitation and fluid dynamics.

Chapter 4

Related work

4.1 Previous work at Vector Fabrics

Previous projects at Vector Fabrics have focused on converting programs to run on the GPU. One of those focused on an analysis that determines what programs can run on the GPU successfully [9], the other concerned a model that predicts performance of programs (kernels) running on the GPU [16]. The results obtained in these projects are orthogonal to those in this thesis.

4.2 GMAC

GMAC (Global Memory for ACcelerators) is a CUDA library that abstracts from the separate address spaces of the CPU and GPU and provides the programmer with a unified address space [5]. This removes the need to have two pointers (one for CPU memory and one for GPU memory) for essentially the same data. This is a very nice abstraction that removes a lot of the boilerplate code. While this abstraction is not necessary for this thesis, a similar abstraction could be created to deal with optimization-specific boilerplate. Not only does this increase user-friendliness but it may also make the optimization easier to reason about, aiding in expanding it to new problems.

4.3 Æcute

Æcute is a framework that decouples access and execute specifications [7]. By separating the two parts, it becomes much easier to reason about and the framework has enough data to optimize data transfer. The user provides an explicit description of the access patterns (i.e. how the program should iterate over the data). As this thesis only deals with a single program class with a relatively simple access pattern, there is little use for this. Still in future work it will be useful to see how this framework encodes those patterns and optimizes the code.

4.4 Thrust

Thrust is a C++ library that simplifies the development of a set of common problems in CUDA [2]. It has efficient implementations of four basic algorithms and also has algorithms that are derived from those four basic ones. The library does not seem to do any optimizations that concern data transfer, so it is likely that it can benefit from tiling/pipelining.

4.5 Memory access coalescing

Memory access coalescing deals with adapting the patterns of memory access in such a way as to optimize it [4, 17, 16]. In the context of GPUs it focuses on having a group of threads access the global memory in such a way that a single memory operation can read or write all the data at once, instead of having each thread have a separate memory operation that must be serialized with the others. While not being the same thing as tiling, both deal with memory access patterns and aim to adapt these so that the most optimal pattern is used. This means that if we want to do both, it is likely that the choices for one of them will affect the other. We may be able to kill two birds with one stone here, but the two may get in each other's way just as well.

4.6 Vectorization

Most modern CPUs have special hardware that can execute a single operation on multiple pieces of data (SIMD). The field of vectorization deals with analyzing programs to determine what data can be combined into vectors or superwords [12]. This can of course only be done if the same operation is going to be performed on those values and the operations are independent from each other. The hardware usually only allows for a handful of elements to be vectorized at once, while the tiles discussed in this thesis tend to consist of roughly a million elements. But with both vectorization and tiling, finding the places where they can be applied are very similar. This thesis does not do any program analysis and only deals with a single program class, so these difficulties are not encountered. Yet if the work in this thesis is going to be used in a full-fledged system, the ability to recognize opportunities for tiling will simply be invaluable.

4.7 Data Parallel Haskell and Regular Parallel Arrays

Data Parallel Haskell is an extension to the Haskell programming language that introduces parallel arrays [3]. These arrays behave mostly like normal arrays or lists, but many basic operations on them have parallel implementations. It uses some fusion techniques so that more complex operations derived from these basic ones are more efficient. It can deal with user-defined element types and parallel

arrays can even be nested, while being stored as flat arrays in memory. It is an elegant approach as the user can use functions that they are already familiar with, so that programming with the parallel arrays is no harder than using lists or normal arrays. Unfortunately not all list functions have an equivalent operation on parallel arrays. It also seems that the use of the stencils in this thesis are not easily expressed in this library, in fact those may not be viable at all in Data Parallel Haskell.

Regular Parallel Arrays (Repa) were introduced as a continuation of this work [10]. These arrays are implemented without needing to introduce new extensions to the Haskell language and provides a pure interface that allows array operations to be expressed naturally with functions such as maps, folds and permutations. For many operations the performance rivals more complex implementations in C.

This style can even be applied to stencils [13]. This makes it a very interesting library as it can do the same as the work in this thesis, but does so in a much more elegant way. However Repa does not employ GPUs, which is the primary focus of this thesis.

4.8 CPU-GPU Communication Manager

The CPU-GPU Communication Manager is a run-time CUDA library and a compiler that combine into a system that performs all the data transfers automatically [8]. The library has its own calls for allocating memory on both the CPU and the GPU, the data transfers are completely implicit. Besides hiding the data transfers it also performs optimizations that turn cyclic communication into acyclic communication. Basically this means that it uses calls that are asynchronous, i.e. that return immediately and then perform their work. This way the CPU can continue with the rest of the program immediately instead of having to wait for the GPU to finish the command. It also tries to move some operations out of loops so that those can be performed earlier on and less often. We think that this is a good way to improve user friendliness. There seems to be little overlap with this thesis though and the work in this thesis will not benefit from it. This is because the Communication Manager takes away control over the data transfers, and that control is needed by the approach in this thesis.

Chapter 5

Proposed solution

This chapter introduces an overview of the proposed solution. Aiding programmers in converting their sequential program to a parallel version is a complicated process involving several distinct operations. A high-level overview of this process is given in figure 5.1. It contains a program analysis, a platform analysis, a performance model, a scheme selection algorithm, a recipe generator and a library. These parts make up the full set of operations needed to analyze an input program, see if it can be optimized, predict performance and help the user modify their code. The following sections explain the purpose of each operation.

5.1 Program analysis

It is important to determine what programs, or parts of programs, can be run on the GPU. This can be done by using a combination of static and dynamic source code analysis. To a large extent, this has been addressed in a previous internship at Vector Fabrics [9]. This work analyzes a given loop[1] and checks that several conditions are met. The first condition states that one loop iteration may not depend on other iterations, i.e. there may be no loop-carried dependencies. A loop must be affine, which basically means that the number of iterations can be statically determined. Also no static or global variables may be used and only a limited set of library and system calls. A loop may contain any number of nested loops, as long as those loops adhere to the same restrictions and all operations are performed in the inner-most loop.

The tiling optimization proposed in this thesis introduces several additional conditions that must be met to enable proper tiling. As we only consider the stenciled map/zip class, these conditions essentially check that the given loop is in this class. The first condition is that input and output is stored in separate arrays. Because multiple iterations may depend on the same inputs, in-place updating would introduce loop-carried dependencies. There may only be up

[1] Parallelization optimizations exploit repetition, making loops the obvious choice for analysis. Recursion is not considered in this thesis.

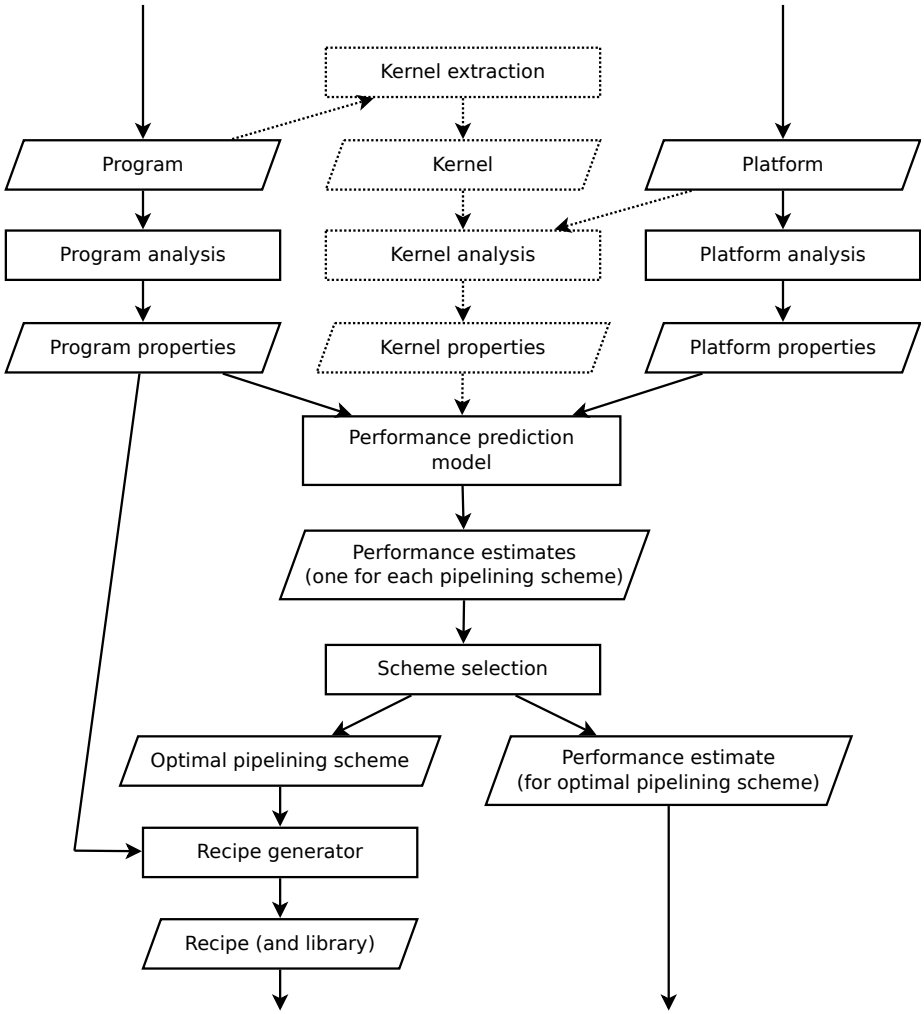

Figure 5.1: High-level overview of the operations that are involved in guided optimization.

to three nested loops, as well as up to three logical dimensions in arrays (this may be implemented as nested arrays or as a single contiguous array). This is because GPUs only supports logic for three dimensions. Furthermore each iteration must map to a single element in each array. In input arrays neighboring elements may be accessed, which make up the stencil.

Once the program analysis decides that a loop is an instance of the stenciled map/zip class, it extracts the following data[2]:

type loopProperties = {

[2]These and subsequent algebraic data types are described in OCaml.

```
  lowerBound: int;
  upperBound: int;
  stride: int
}

type arrayProperties = {
  bytesPerElement: int;
  size: vector;
  mapping: matrix
}

type inputArrayProperties = {
  general: arrayProperties;
  stencil: vector list
}

type programProperties = {
  dimensionality: int;
  iterationSpace: loopProperties list;
  inputArrays: inputArrayProperties list;
  outputArrays: arrayProperties list
}
```

These data types model the structure of the analyzed loop as far as is relevant for the performance prediction model and the recipe generator[3]. To illustrate how these data types work, we give an instance for the two-dimensional Emboss program:

```
let emboss = {
  dimensionality = 2;
  iterationSpace = [
    {lowerBound = 0; upperBound = 8000; stride = 1};
    {lowerBound = 0; upperBound = 8000; stride = 1}
  ];
  inputArrays = [
    {general = {bytesPerElement = 4; size = [8000; 8000]; mapping = [[1;1]]};
    stencil = [[−1;−1];[0;−1];[−1;0];[1;0];[0;1];[1;1]]}
  ];
  outputArrays = [bytesPerElement = 4; size = [8000; 8000]; mapping = [[1;1]]}]
}
```

Here the *dimensionality* indicates the number of nested loops, while *iterationSpace* shows that each of these loops runs from 0 up to (but excluding) 8000 with a stride of 1. We have one input array and one output array, each of them is 8000x8000 elements. These elements are floats and thus use 4 bytes

[3]The quality of generated recipes can be improved by having the program analysis provide more detail, e.g. on variable names or line numbers. But this is not done in this thesis.

each. The *mapping* value indicates that each array is accessed with the iteration variables directly. We encode this explicitly to allow one loop to iterate over multi-dimensional arrays and multiple nested loops to iterate over a single array. The *stencil* value indicates what values in the input array are used. In this case six neighbors are used and the value itself is not.

5.2 Platform analysis

The choice for the most optimal pipelining scheme does not only depend on the program, but also on the platform this program is intended to run on. Therefore a platform analysis is included that extracts relevant performance metrics. These are stored in the following data type:

```
type platformProperties = {
  copyEngines: int;
  pinnedHostToPinnedHost: measurements;
  pinnedHostToDevice: measurements;
  deviceToPinnedHost: measurements;
  mallocFreeHost: measurements;
  mallocFreeDevice: measurements;
}
```

This data type contains the number of copy engines (see section 2.5.1), run times of data transfer operations that are used by the tilings schemes and run times of memory allocation and deallocation. To illustrate we show the instance for the Tesla C2050:

```
let tesla = {
  copyEngines = 2;
  pinnedHostToPinnedHost = [13.774600; 20.085802; ... 1849985.625000];
  pinnedHostToDevice = [3184.277100; 3259.644043; ... 1697684.750000];
  deviceToPinnedHost = [3253.160400; 3516.952637; ... 1644188.375000];
  mallocFreeHost = [779003.500000; 778444.375000; ... 7295391.000000];
  mallocFreeDevice = [101616.515625; 101620.953125; ... 224929.796875]
}
```

The Tesla has dual copy engines as is indicated. Furthermore the relevant metrics are added that show how many nanoseconds each operation takes when done for 10, 100, 1000 etc. up to 10.000.000 bytes at a time (i.e. for powers of 10). The performance prediction model uses these values to estimate various data transfer and memory allocation/deallocation times. Host refers to the CPU, device to the GPU and pinned to the fact that the memory is page-locked.

5.3 Kernel analysis

The difficult task of predicting the run time of kernels is outside the scope of this thesis, but we still need accurate kernel metrics to generate an accurate

prediction. If we were to actually predict it, the program analysis would have to extract additional data that tells what operations are used in the kernel. The platform analysis would also have to provide metrics on individual kernel operations. The performance prediction model would then predict the kernel run time by using that data. The performance of kernels is often non-linear, so predicting it is a complex subject in itself. Previous work at Vector Fabrics has already focused on modeling kernel performance [16].

In this thesis we simply create the kernel from the given input program (by hand), then measure the run time in the same way as the platform analysis measures the run time of data transfers and memory allocations. This provides us with a reliable run time for kernels, that can be used directly in performance predictions of the full program. In figure 5.1, the blocks associated with kernel analysis are dotted, to indicate that these disappear in a full implementation.

5.4 Performance prediction model

Once the program properties, kernel properties and platform properties are extracted, this data is fed into the performance prediction model. There are several ways of implementing the tiling/pipelining method, which we call pipelining schemes. Each of these schemes is modeled and an estimate of the total run time is generated. As the run time depends on the chosen tile size, each result is returned as a function:

```
type estimates = {
  basic: tileSize -> float;
  scalable: tileSize -> float;
  buffered: tileSize -> float;
}
```

The concrete pipelining schemes referred to here are explained in the next chapter (section 6.2).

5.5 Scheme selection

The output of the performance prediction model is used by the scheme selection algorithm to select the optimal combination of pipelining scheme and tile size. Once this has been selected, the performance prediction is returned, so that it can be used to give the user an expected speed-up. The selection itself is also returned and used by the recipe generator.

5.6 Recipe generator

After a pipelining scheme has been chosen, a recipe is generated that explains how the input program is to be transformed. This recipe is presented to the user/programmer so that they can make the necessary changes. To cut down

on boilerplate and to simplify the recipe as much as possible, much of the implementation details are provided in a library. The recipe does not only use knowledge of the chosen tiling scheme, but also data extracted by the program analysis such as stencil size. An advanced version can even include information such as line numbers and variable names to make the recipe more concrete and informative, but this is omitted in this thesis.

Chapter 6

Implementation

In this chapter we explain in detail what has been implemented during this thesis. We explain how tiles are chosen; what the various approaches to pipelining entail; how small data transfers affect performance; how the platform and kernel analyses are done; how the performance model works in detail and finally how the library can be used. The program analysis and the scheme selection as described in the previous chapter are not implemented in this thesis and so are not explained. The recipe generator also is not implemented, but a prototype recipe is given.

6.1 Tiles

In this thesis, tiles are always created as either a range, a rectangle or a cuboid for one-, two- or three-dimensional programs respectively. The sizes in each dimension can be freely chosen and these affect performance in several ways.

Choosing smaller tiles has its advantages. As mentioned in section 2.6, at the start and end of a pipeline not all engines are working. When the first tile starts, only one (copy) engine is working, when the second tile starts two engines are working (copy and kernel engine). GPUs with only a single copy engine are maxed out at that point, while those with a second one are maxed out when the third tile starts. At the end of the program, the same occurs in reverse. This happens only for the first and last few tiles, so having smaller tiles reduces this effect. Smaller tiles can also save on both total memory used and memory allocation/deallocation time (for the scalable and buffering schemes described in section 6.2).

Larger tiles can also pay off. Since fewer of them are needed, the number of data transfer operations and kernel launches is reduced, which decreases the overhead incurred by these operations. If there is padded data on tiles, larger ones have fewer padded elements relative to their size. This reduces the total amount of data that is transferred.

In two and three dimensions not just the size of the tile matters, but also

its shape. Having square or cubic tiles helps reduce the impact of padding. For example a 25x1 tile with one element of padding on each side has 54 elements of padding, while a 5x5 tile has only 22, even though both tiles have the same amount of elements. On the other hand, having tiles that are longer on the x-axis reduces the number of data transfer operations (this is explained in section 6.2.3).

Based on the program and the platform, these effects each occur in some degree. This means that somewhere between one tile for the whole program and one tile for every element there is a sweet spot. There is no single sweet spot for all programs. This is why we created the performance prediction model, so that we can determine this quickly and reliably based on properties of the program, the platform it is to run on and details of the optimization used. Section 7.1 shows the effects of tile size on the run time of some typical programs.

6.2 Pipelines

For this thesis, we implemented several pipeline schemes. While the basic notion of the pipeline is present in each of them, there are some variations in the implementation details. These variations concern how memory is allocated on the GPU and how data is transferred to it. This section first explains what happens with the naive approach, then it introduces two simple pipelining schemes and discusses their relative strengths and weaknesses. It also explains a problem that arises with small data transfers and introduces both another scheme and a small modification of the existing schemes that aim to circumvent this problem.

6.2.1 Naive approach

Figure 6.1 shows how for a single input array, memory is allocated on the GPU and how data is transferred to it. It also shows a representation of the pipeline, where each engine is scheduled to perform an operation. Because in the naive approach there is no actual pipelining, it shows only the RPC model where first data is transferred to the GPU, the kernel is performed and finally the results are sent back to the CPU (as discussed in section 2.5).

6.2.2 Initial schemes

Basic scheme

The first scheme is rather straightforward, so we have called it the basic scheme (see figure 6.2). For each input and output array, it allocates an array of the same size on the GPU. The data associated with each tile is then copied from the CPU input array to the same location in the GPU input array. Then for that tile, a kernel is run that reads the data in the GPU input array and writes to the GPU output array. Then from the GPU output array the data is transferred back to the CPU output array. As there are multiple tiles, work can be overlapped as is shown in the figure.

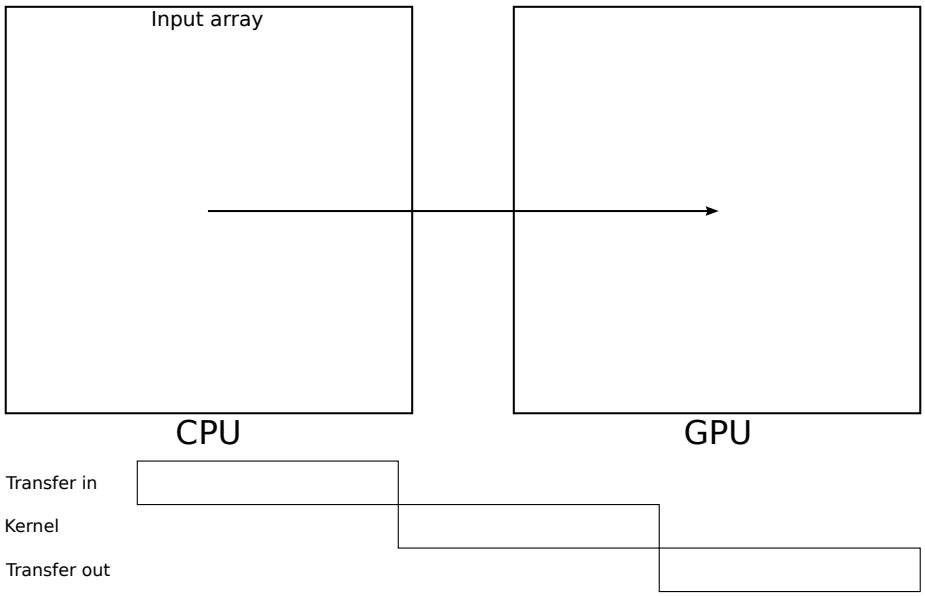

Figure 6.1: Data transfer of an input array and the pipeline schedule for the naive approach.

Scalable scheme

The scalable scheme (see figure 6.3) is similar to the basic scheme. However instead of allocating a full-sized array on the GPU, a tile-sized array is allocated for every input and output array. This isolates the data for each tile from that of the other tiles. As there are only three copy/kernel engines, only three tiles are worked on at a time. This means that when we start work on the fourth tile, the memory used for the first tile is free. This allows us to reuse this memory for the fourth tile. This saves time, as allocation and deallocation of arrays is not needed beyond the first three. In this scheme, each kernel operates on just one of these tiles and a corresponding output array that is also tile-sized. Then from that output array it is transferred back to the CPU and put into the correct location in the final output array. The arrays that are allocated for the input tiles must be large enough to include all necessary padding. For the output arrays this is not necessary as neighbors can only be read, not written to. Note that the pipeline schedule is the same as for the basic scheme.

Trade-offs

We cannot clearly say that one of these schemes is better than the other. But each does have its relative strengths and weaknesses. The basic scheme is the easiest to implement, as elements can be indexed the same way on the GPU as they can be on the CPU. This makes it far easier to implement. In the library

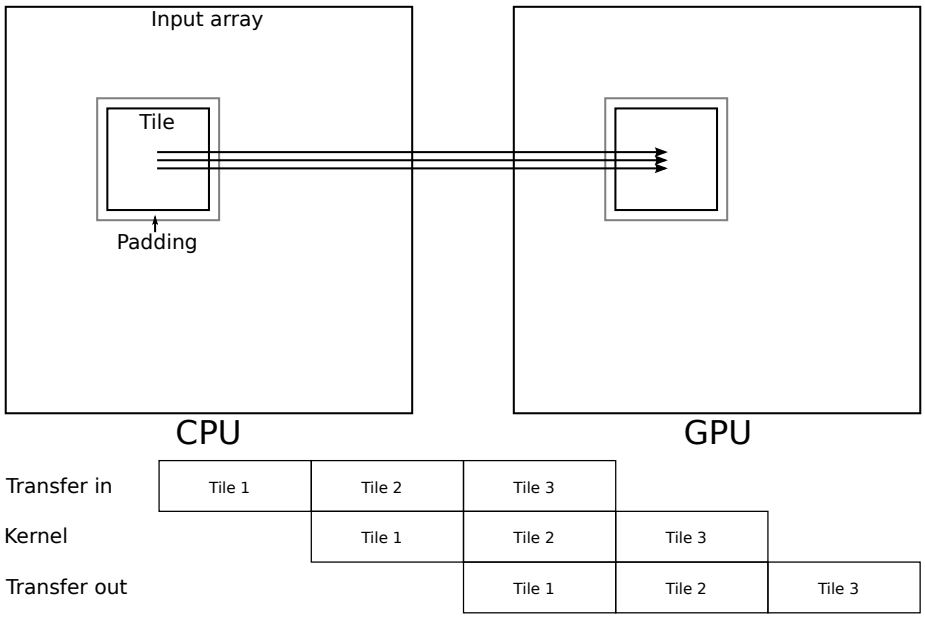

Figure 6.2: Data transfer of a tile of an input array and the pipeline schedule for the basic scheme.

implemented in this thesis, the user has to manually call an indexing function for each array access. These calls do the actual conversion, but the user has to add them by hand. It may be possible to remove the need to add them manually through the pre-compiler.

On the other hand, the scalable scheme needs far less memory than the basic one. This means that less time is needed to allocate and deallocate this memory on the GPU. In most situations the scalable scheme is faster because of this (see section 7.1.1). Also because the size of memory on the GPU is based on the tile size and not the array size, it can easily run programs that use more memory than is available on the GPU.

At this point the schemes do not account for the fact that with heavy padding, there is a lot of overlap in tiles. Both just copy this overlapping data multiple times. For the basic scheme, it would be easy enough to eliminate these redundant data transfers. For the scalable scheme this is not the case, as the overlapping data is actually needed in different places in memory on the GPU. It may be possible to use extra kernels to move this data around, but this adds much complexity and may introduce so much overhead that is not worth it. In practice, we did not run into any instances of the stenciled map/zip class with a lot of padding though, which is why this was not a priority. In section 7.1.3 we discuss the implications of this decision.

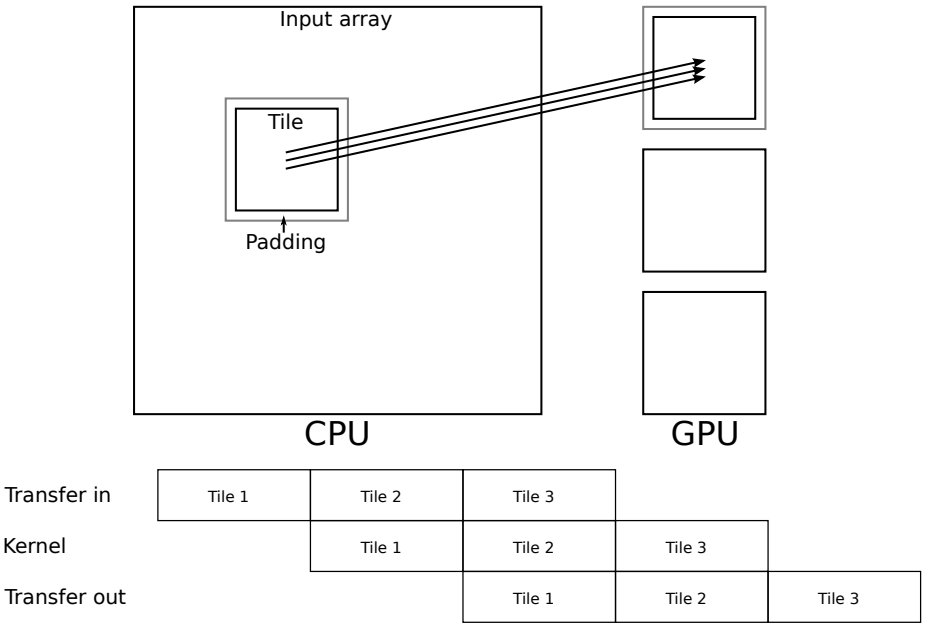

Figure 6.3: Data transfer of a tile of an input array and the pipeline schedule for the scalable scheme.

6.2.3 Small data transfers

Non-contiguity

When using the basic and scalable scheme on two- or three-dimensional programs, run times became very large. After some analysis, it turned out this was caused by the way data was transferred. To illustrate, look at figure 6.4. Because two- and three-dimensional data is physically stored in a single contiguous one-dimensional array, tiles are usually not contiguous in memory. To transfer this data, many smaller data transfers are needed.

This does not have to be a problem in itself, but unfortunately small data transfers to and from the GPU are extremely expensive. To illustrate, the graph in figure 6.5 shows the transfer time per byte for data transfers of various sizes. Transferring 10 bytes at a time is in fact more than 1600 times slower than doing 100,000 at once (326 vs. 0.20 ns/B). Transferring data on the CPU itself (i.e. CPU to CPU) has the same problem, but it is far less pronounced (see figure 6.6 for a better view). Here 10 bytes at a time is roughly 10 times slower than 100,000 at once (0.67 vs. 0.07 ns/B). Oddly enough the efficiency is lowered with data transfers that are larger than 100,000 bytes; we are not sure what causes this, possibly caching behavior.

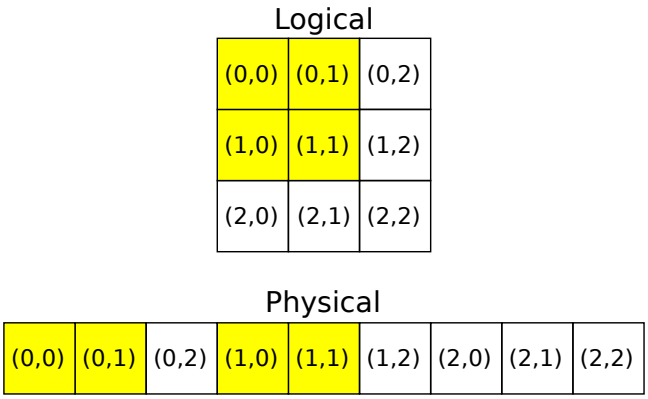

Figure 6.4: Logical versus physical views on a two-dimensional array. The selected tile (in yellow) is not contiguous in physical memory.

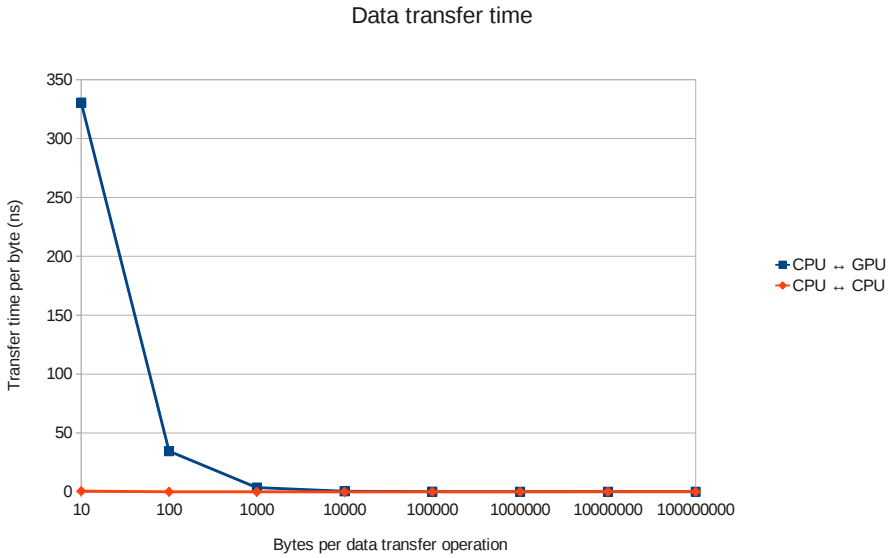

Figure 6.5: The clear difference in data transfer times for smaller vs. larger data transfers. These values are for an NVIDIA Tesla C2050 and an Intel i5.

Buffered scheme

Because the overhead of smaller data transfers is so much less of an issue on CPU transfers, we introduce the buffered scheme (see figure 6.7). This adds an intermediate buffering step on the CPU to the scalable scheme. Instead of sending data from the input array to a GPU tile array, this data is sent to a tile array on the CPU. From this array it is then transferred to the GPU.

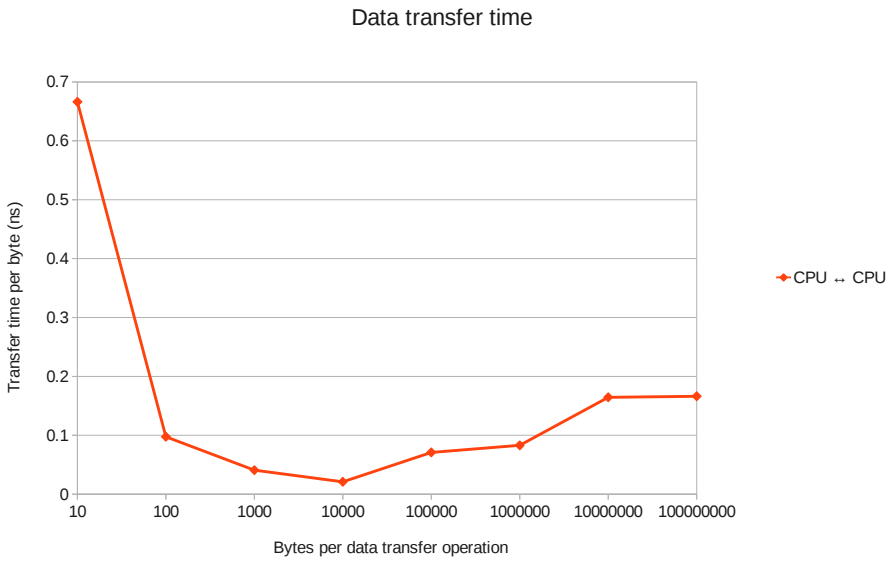

Figure 6.6: A more informative view on CPU to CPU transfers. Again these values are for an NVIDIA Tesla C2050 and an Intel i5.

For the output, there is an analogous buffering step. The CPU to/from GPU transfers are all contiguous, eliminating the effect that small data transfers have on transfer speed.

Unfortunately the performance of this scheme is quite poor. While it does much better than the basic and scalable ones in cases that have many small data transfers, the buffered scheme transfers each piece of data twice and this incurs its own overhead. So much so that its performance circles around that of the naive approach. Section 7.1 shows the performance of this scheme for some typical programs.

Luckily the work that was done on this scheme is not wasted effort. In this thesis we only focus on programs that store data in physical memory as a single contiguous array. Yet if a program were to use nested arrays (i.e. arrays of pointers to arrays) the data cannot be transferred in a single copy to begin with. This means that naively the data is either transferred with many small transfers, or the data is flattened beforehand. At that point the buffered scheme definitely outperforms the naive approach. It is of course best to use flattened arrays to begin with, but that may not always be an option.

Making data contiguous

As the buffered scheme was inadequate in resolving the problem of non-contiguous memory, another approach is needed. Recall figure 6.4 that shows how data in a tile is not necessarily contiguous in physical memory. We can in fact make a

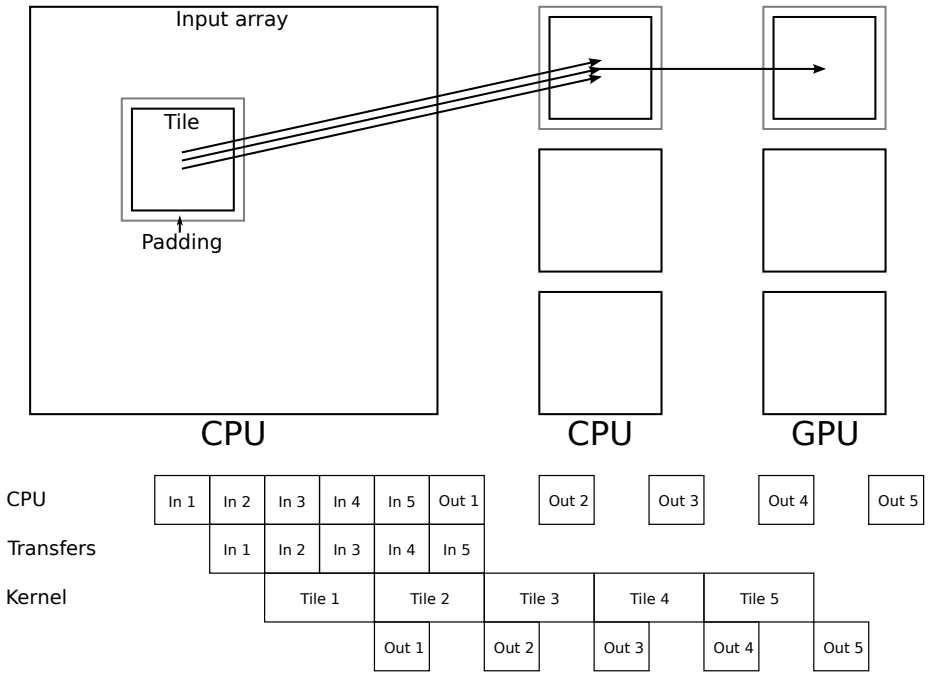

Figure 6.7: Data transfer of a tile of an input array and the pipeline schedule for the buffered scheme.

two-dimensional tile contiguous by stretching it in the X-axis until it is as large as the array itself (see figure 6.8).

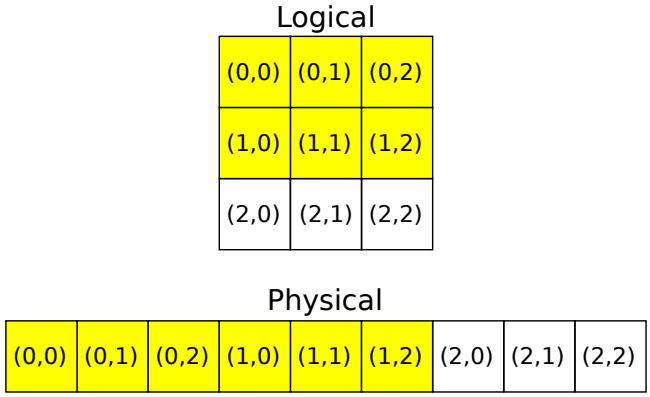

Figure 6.8: Logical versus physical views on a two-dimensional array. This time the selected tile (in yellow) is contiguous in physical memory.

We adapt the three previous schemes (basic, scalable and buffered) to recog-

nize when the tile size in the X-axis is as large as that of the input and output arrays on the CPU. If this is the case, the small data transfers are replaced by a single larger transfer. As the small data transfers are removed, their overhead also disappears. This adaptation works very well, resulting in multi-dimensional programs performing roughly as well as one-dimensional ones.

This optimization only works with very specific tile sizes, as tiles must be as large as the full array on one or more axes for the data to become contiguous in memory. This means that we must make a trade-off between the loss of performance by choosing this particular tile size (as discussed in section 6.1 and the gain in performance of using the optimization. For example, programs with very large arrays use data transfers that are large enough to begin with. The effectiveness of this adaptation is shown in section 7.1.

6.3 Platform and kernel analysis

Both the platform analysis and the kernel analysis are implemented as a single executable C program. Running this program on a system provides a list of measurements of the run times of each of the various data transfer and memory allocation/deallocation operations. It also provides these results for the moving average, emboss and Jacobi stencil kernels.

Each list holds the run time of an operation performed on powers of 10 bytes, so 10, 100, 1000 etc. This gives enough data points for (linear) interpolation. The program repeats measurements and calculates the average, so only a single run is required for accurate results. The output is in OCaml code that can be used directly in the code for the prediction model. Figure 6.5 has already illustrated the values retrieved for both CPU to/from GPU and CPU to CPU data transfers.

The choice to use powers of ten and linear interpolation is a practical one. It is accurate enough for our purposes, easy to implement and the analysis program terminates within a few minutes.

6.4 Performance model

In this section, we explain how we have modeled the performance of the various tiling/pipelining schemes. First we discuss memory allocation and deallocation, kernel execution and data transfer individually and then how the performances of each is combined into a pipeline.

6.4.1 Memory allocation/deallocation prediction

For the tiling schemes to operate, memory must be allocated on the GPU. For the buffered scheme, some must even be allocated on the CPU. This memory must also be freed after use. First we have to determine how much memory every scheme needs:

Basic scheme For every input and output array, an array of the same size is allocated on the GPU.

Scalable scheme For every input array, three arrays are allocated on the GPU. These arrays are as big as the chosen tile size, plus padding. If the size of the tile in the X-axis is the same as that of the input array, the padding in that direction is omitted[1]. If this is the case, this may also happen for the Y-axis. The implementation of the scheme does not remove the padding in the Z-axis if possible, as at that point there is only one tile and pipelining has become redundant. For every output array, three tiles are allocated on the GPU as well. These also have the size of a tile, but do not have padding.

Buffered scheme For the buffered scheme the same memory is used as for the scalable scheme. In addition to this, the same memory is duplicated on the CPU for the buffers.

Now that we know how many allocation/deallocation operations are needed and how many bytes each of them needs, we can get a run time for these operations by (linearly) interpolating on the values provided by the platform analysis.

6.4.2 Kernel prediction

In a real system, the run time for a kernel is predicted by first making a detailed analysis of the work the kernel does and the hardware it is running on. There are many optimizations available to speed up a kernel and not all of these are very obvious. It can matter how data is stored and accessed for example, and this can contribute to total run time much more than just the amount of data that is used. It is an exciting and difficult subject in its own right.

As this thesis focuses on the data transfer from and to the GPU and the overlap between these and kernel execution, predicting the run time of the kernel itself is out of scope. Instead we use the values provided by the platform/kernel analysis directly, to interpolate how much time a kernel takes for a single tile of the chosen tile size.

6.4.3 Data transfer prediction

To predict the data transfer time per tile we need to determine both how many data transfers there are and how many bytes each of them transfers. First we determine this for the transfer of output data, as this is easiest. Then we do so for the input data where padding makes things trickier.

Output transfer

For every tile that is processed, we transfer a tile-sized chunk of data from the GPU to the CPU for each output array. In the three-dimensional case, this

[1] If we did not omit this padding, data that is contiguous on the CPU would not be contiguous on the GPU, defeating the purpose of choosing that specific tile size.

data is in the shape of a cuboid. If this tile is of size (X, Y, Z), it is clear that we have to transfer a total of $X * Y * Z$ elements. But as mentioned in section 6.2.3, this data is not contiguous in physical memory on the CPU. It is part of a bigger array that is stored in a single flattened array.

We have already stated that we assume that all arrays are flattened. This means that a single contiguous array is used to stored multi-dimensional data. For this prediction, we also assume that the program analysis describes the program in such a way that the logical index (x, y, z) corresponds to the physical index $x + X * (y + Y * z)$[2]. That means that normally only the elements that have the same y and z coordinates are contiguous in memory. In this case we have $Y * Z$ data transfers, each X elements long.

As is explained in section 6.2.3, if on the x-axis the tile is as big as the output array itself we can make do with fewer, longer data transfers. When this is the case, all data that has the same Z-coordinate is contiguous in physical memory. We end up with Z data transfers, each $X * Y$ elements long. If in this case the same condition holds in the y-axis, we get only a single data transfer that is $X * Y * Z$ elements long.

Once we know how long each of the data transfers is, we can use the data provided by the platform analysis to determine how long each of these transfers takes. We then multiply by the number of transfers to get the total duration of output data transfers for a single tile. Note that for the buffered scheme, we need to use different measurements than for the basic and scalable schemes (CPU to CPU instead of GPU to GPU). We must also add a single GPU to CPU transfer of $X * Y * Z$ elements.

Input transfer

For transferring input, the same model is used as for transferring output, but we must first modify the tile size we work with to account for padding. The initial conversion is quite straightforward, we simply add the tile size (X, Y, Z) to the padding size (PX, PY, PZ).

However if the tile size on the x-axis is as big as the input array, we need to remove this padding on that axis. It is not just that this extra memory is never used, but keeping it causes the data that is contiguous on the CPU to not be contiguous on the GPU. Therefore we must remove it or we run into the same inefficiencies we are trying to avoid. The same goes for the y-axis, but only if it is already done for the x-axis. In other words, based on contiguity of data of the tile we get a tile size of either $(X + PX, Y + PY, Z + PZ)$, $(X, Y + PY, Z + PZ)$ or $(X, Y, Z + PZ)$. With this modified tile size, we do the same as for the output transfers.

While the preceding model is precise for the input tiles that are in the middle of the array, it is not quite accurate for the tiles that are at the edges. This is because at the edges of the array, some of the padding is necessarily removed as those elements do not actually exist (and accessing them would lead to a

[2] Uppercase letters indicate tile size on one of the axes. Lowercase letters indicate indices.

segmentation fault). For programs with a lot of padding, this may cause the prediction to be less accurate. It is not a significant problem in practice as the schemes we have implemented are not suited to programs with very large stencils anyway.

If we were to implement more involved schemes that take into account that the padding on one tile is also used in one or more other tiles, modeling how much data is sent and when would become more involved as well.

6.4.4 Pipeline prediction

The transfers of input and output and the execution of the kernels are scheduled in a pipeline. It is important to understand how the pipeline works so we can properly model when operations overlap and when they do not. This section shows what various pipelines look like and how they can be properly modeled.

Dual copy engines

Depending on the program either the data transfer to the GPU, the execution of the kernel, or the data transfer back to the CPU takes the longest; we call this the dominant operation. Which of these operations is dominant affects how the pipeline behaves. Figure 6.9 shows what happens when each of the operations is dominant.

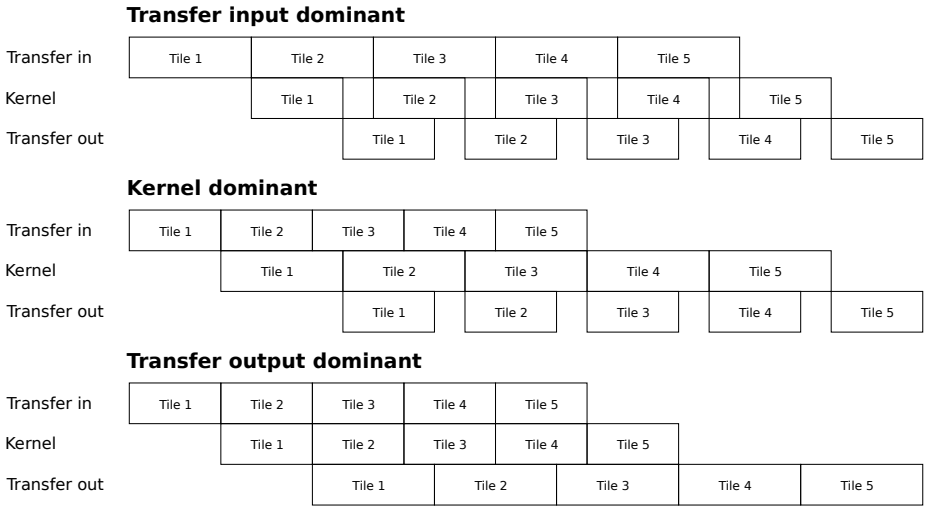

Figure 6.9: The behavior of a five-tile pipeline on a GPU with dual copy engines, for various dominant operations.

It is interesting to see that in each case, once the engine performing the dominant operation starts working, it processes all tiles back-to-back. We call the part where the dominant operation is being performed the inner part. The parts where the dominant operation is not being performed is called the outer part(s).

As the duration of the inner part was defined as the sum of the durations of the dominant operation for each tile, we can express this as $5 * dominant\_operation$. If we abstract from the number of tiles and make the meaning of the term dominant operation more concrete, we get

$$\#tiles * \max(transferIn, kernel, transferOut).$$

For the outer part, we can clearly see that its duration is always equal to the sum of the durations of the non-dominant operations. This can be expressed as

$$transferIn + kernel + transferOut - \max(transferIn, kernel, transferOut).$$

We can simply combine the inner and outer parts to get the run time of the entire pipeline, we get

$$transferIn + kernel + transferOut$$
$$+(\#tiles - 1) * \max(transferIn, kernel, transferOut).$$

If we interpret this formula, we can look at it as us having to pay full price for the first tile, but each additional tile only adding the cost of the dominant operation. Unfortunately, this interpretation does not help us if we try to model the behavior when there is only a single copy engine and/or when buffering is involved. Another interpretation of the pipeline is needed.

Single copy engine

The pipelines for a single copy engine are pictured in figure 6.10. To illustrate that the previous view does not work well in this case, we try to interpret it in the same way. Since both copy operations are now performed on the same engine, we need to add them up instead of taking the maximum. We get this slightly modified formula:

$$transferIn + kernel + transferOut$$
$$+(\#tiles - 1) * \max(transferIn + transferOut, kernel).$$

When we check this formula against the pipeline in the figure, we see that it does not work. This is because even though both transfers now operate on the same engine, they are not a single operation. Each still has a different set of interdependencies with the kernels and each other.

The problem is actually that we have not correctly identified the inner and outer parts. Recall the description of pipelines in section 2.6. Here we have explained that when the pipeline is started, only one engine is active as the rest is waiting for input. Then the next step two engines are working and then all three are working. The reverse is true when the pipeline finishes up, first decreasing to two engines and then only one. Our formula does not reflect these start-up and shutdown steps though. But we can adapt it so that it models them properly.

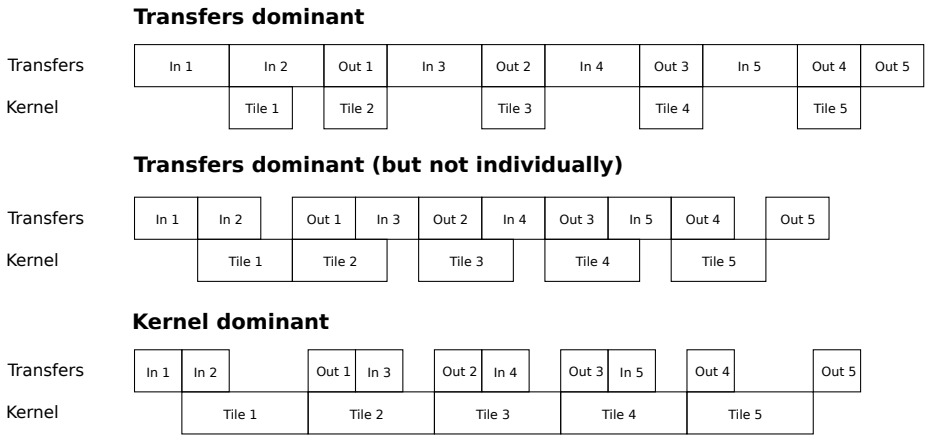

Figure 6.10: The behavior of a five-tile pipeline on a GPU with a single copy engine, for various dominant operations.

We can model the first step as *transferIn*, as this step only transfers input. The next step has both an input transfer and a kernel, so its length can be defined as $\max(transferIn, kernel)$. Then we get the step where all three operations are performed, that we can model with $\max(transferIn + transferOut, kernel$. This step is actually repeated often, depending on the total number of tiles in the pipeline. We get similar formulas for the shutdown steps, leading to the following compounded formula:

$$\begin{aligned} &transferIn + \max(transferIn, kernel) \\ &+(\#tiles - 2) * \max(transferIn + transferOut, kernel) \\ &+ \max(kernel, transferOut) + transferOut. \end{aligned}$$

We can actually identify all of these steps in the pipelines we have drawn; For the single copy engine pipeline this is done in figure 6.11.

Dual copy engines revisited

The start-up and shutdown steps that model the behavior with a single copy engine can also be used to model the behavior with dual copy engines. When we take another look at figure 6.9, at first glance it does not seem as though this view is applicable. This is because we have scheduled those pipelines so that all operations start as soon as they possibly can. But we can safely postpone various non-dominant operations without affecting the total length of the pipeline. We have done so in figure 6.12 and postponing these operations reveals the same steps as we had in the single copy engine case.

So now we can actually model the pipeline as it should look. Regrettably measurements on the Tesla C2050 have shown that when both copy engines are working, the total bandwidth is not doubled. In fact it only in-

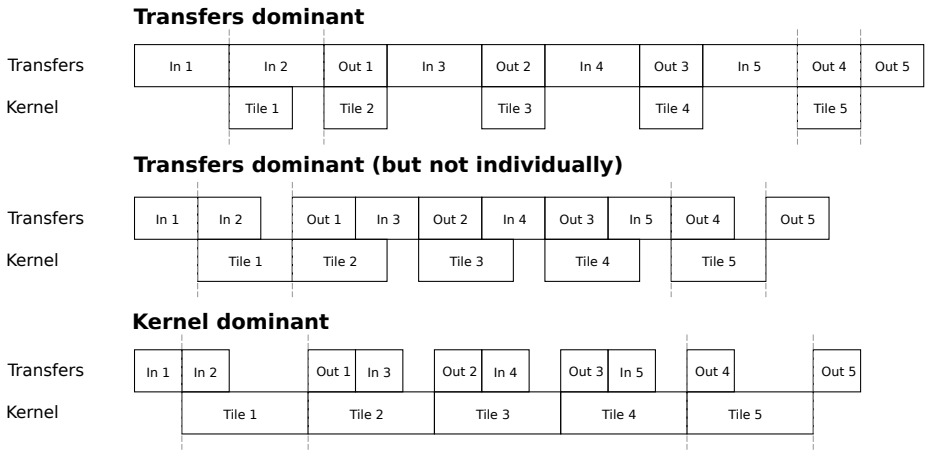

Figure 6.11: The behavior of a five-tile pipeline on a GPU with a single copy engine with the start-up and shutdown steps marked.

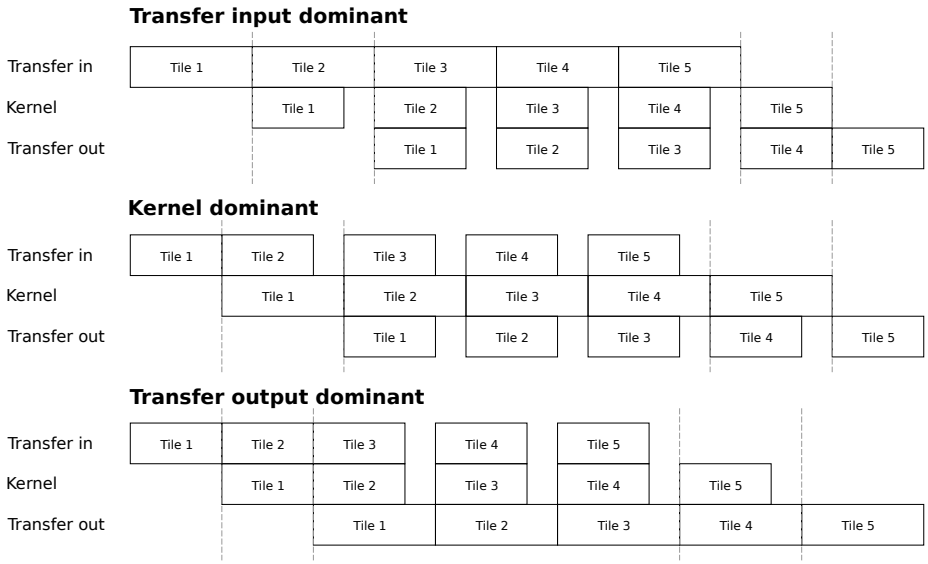

Figure 6.12: The behavior of a five-tile pipeline on a GPU with dual copy engines with the start-up and shutdown steps marked.

creases by 26%, which is far from advertised. We modeled the duration of both transfers as $\max(transferIn, transferOut)$, but this is not correct in practice. If both transfers take roughly the same time, a bandwidth of 126% means that the duration is 79.5% of the sum of the two operations, or 157% of one of them. One of the operations may be longer than the other (e.g. when there are multiple input arrays), meaning that these operations only

overlap during the duration of the shorter one. This overlapping part can be expressed as $1.57 * \min(transferIn, transferOut)$. What is left is the part where the longer operation does not overlap and goes at full speed, modeled as $\max(transferIn, transferOut) - \min(transferIn, transferOut)$. If we combine these parts and simplify, we get the formula:

$$\max(transferIn, transferOut) + 0.57 * \min(transferIn, transferOut).$$

It is unclear what exactly causes the bandwidth to only increase so little. It may be the maximum bandwidth of the GPU or the CPU memory is just not large enough. It may be that the GPU has to do a lot of extra synchronization. It may be the connection between GPU and CPU, although the PCI/e bus should have separate bandwidth for each direction. As we do not really know why it is happening, it is hard to predict what happens on other GPUs with dual copy engines.

Still, now that we know how to correctly model both the pipeline and concurrent data transfers, we can create the final formula for dual copy engines, which is:

$$\begin{gathered} transferIn + \max(transferIn, kernel) + (\#tiles - 2)* \\ \max(\max(transferIn, transferOut) + 0.57 * \min(transferIn, transferOut), kernel) \\ + \max(kernel, transferOut) + transferOut. \end{gathered}$$

Buffered scheme

The buffered scheme adds extra buffering steps to the start and end of the work on a tile. This makes the pipeline more complex. Figure 6.13 shows the pipelines for both single and dual copy engines and marks the start-up and shutdown steps, of which there are now four on each end. The analyses in the previous sections work for the buffered scheme too. The following formula models the behavior when using a single copy engine:

$$\begin{gathered} bufferIn+ \\ \max(bufferIn, transferIn)+ \\ \max(bufferIn, transferIn, kernel)+ \\ \max(bufferIn, transferIn + transferOut, kernel)+ \\ (\#tiles - 4) * \max(bufferIn + bufferOut, transferIn + transferOut, kernel)+ \\ \max(bufferOut, transferIn + transferOut, kernel)+ \\ \max(bufferOut, transferOut, kernel)+ \\ \max(bufferOut, transferOut)+ \\ bufferOut \end{gathered}$$

This formula can be adapted to dual copy engines by substituting each instance of $bufferIn+bufferOut$ with $\max(transferIn, transferOut)+0.57*\min(transferIn, transferOut)$ just as we did with the basic and scalable schemes.

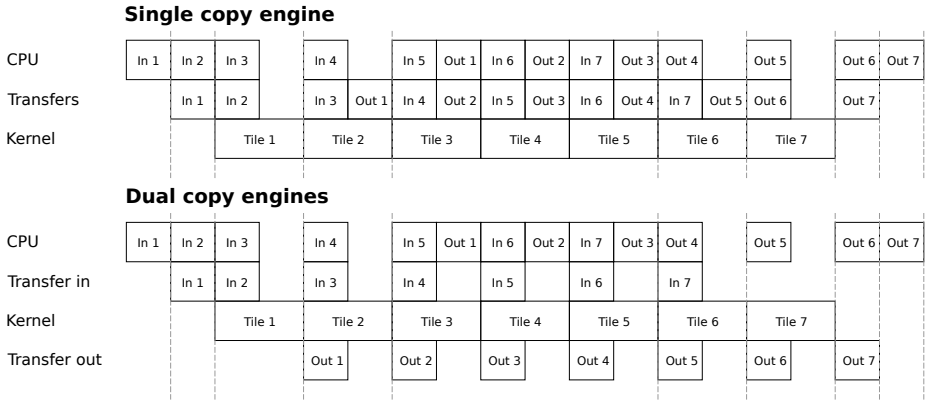

Figure 6.13: The behavior of a seven-tile pipeline on a GPU with both single and dual copy engines with the start-up and shutdown steps marked.

6.4.5 Final compounded prediction

The final compounded prediction is the prediction for the pipeline added to the prediction of the allocation and deallocation of the memory. The predictions for the basic and scalable schemes are implemented in OCaml. The buffered scheme is not implemented, but using this thesis and the other implementations as a reference, that should be easy enough. In section 7.2 the accuracy of the predictions of this model are determined by comparing them to the actual run times.

6.5 Library and recipe

The three schemes described in section 6.2 are implemented in a CUDA library. This library can be used to easily map sequential programs that are in the stenciled map/zip class to versions that use tiling and pipelining on the GPU. In addition to the constraints mentioned in 5.1, this library has the following: array elements must be floats, all arrays must be the same size, arrays must be flattened in such a way that for each of them the physical position of an element with logical index (x, y, z) is $x + X * (y + Y * z)$ (where X and Y are the array size in the x and y-axis respectively). These constraints were added to ease implementation and we intended to remove them eventually, but never did. We reckon that removing the restrictions will not be too difficult, but it does mean that more configuration parameters are needed which makes the library less convenient. However instead of lifting just these constraints, it may be better to also generalize to other programs classes.

To give users instructions on how to (manually) convert their programs, we provide them with a recipe. The following prototype recipe explains succinctly how to convert the sequential version of a program to a version that uses this

tiling/pipelining library:

1. Create a kernel that performs the computation for a single element.
   (a) Extract the body from the loop(s) and put it in a new *__global__ void* function we will refer to as *kernel*.
   (b) The *kernel* function must have 6 parameters:
      i. A *vector3* called *size* that holds the size of the length in each direction of the arrays.
      ii. A *grid3* called *inP* that holds information that is needed for the *idx1/2/3* functions when accessing input arrays.
      iii. A *float *** called *in* which is an array containing all input arrays.
      iv. A *grid3* called *outP* that holds information that is needed for the *idx1/2/3* functions when accessing output arrays.
      v. A *float *** called *out* which is an array containing all output arrays.
      vi. A *vector3* called *offset* that holds offsets for the tile and is to be used to get the correct coordinates.
   (c) Add CUDA-specific code to the *kernel* function that determines the x, y and z coordinates for the current element.
   (d) Surround the extracted computation code with an if-statement so that it performs the body only if the x, y and z coordinates are each smaller than those of the 'vector3' parameter that indicates the dimensions of the arrays.
   (e) Make sure to access the elements of each array by using the *idx1/2/3* function.
2. Replace the original code with a scheme that uses CUDA calls to concurrently copy and calculate on the GPU.
   (a) Include the CUDA tiling library.
   (b) Create an array and put each input array in it. Do the same for the output arrays.
   (c) Add a call to *runScalable*; as arguments use the size of the arrays, the number of input arrays, the array of input arrays, the number of output arrays, the array of output arrays and the kernel that was created in the previous steps. (In an actual recipe created by the recipe generator, this section would also indicate the values of the arguments for tile size, block size, number of streams and padding size.)
3. Compile with CUDA.
   (a) Change the extension of the file from '.c' to '.cu'.
   (b) Compile with 'nvcc' instead of 'gcc'.

As mentioned before we did not implement a recipe generator. Such a generator basically outputs the given recipe and add values for relevant parameters such as tile size and block size. A more involved generator can even include line numbers and variable names to make the recipe more concrete, but then the recipe will look different. The given recipe is similar in style to those used by Vector Fabrics. However in their recipes each step has a small link to a more detailed explanation and examples. This recipe does not have this expanded documentation yet.

Section 2.4 has already shown how a sequential program can be converted to a CUDA program. Now we show the result of using the recipe on that same sequential program:

```
// The user still extracts a kernel, but with the following parameters.
__global__ void kernel(vector3 size, grid3 inP, float **in,
    grid3 outP, float **out, vector3 tileOffset) {
  // Tile information is used to correctly retrieve thread coordinates.
  int x = tileOffset.x + blockDim.x * blockIdx.x + threadIdx.x;
  int y = tileOffset.y + blockDim.y * blockIdx.y + threadIdx.y;

  // The body is only changed slightly, a new idx2 function is used that corrects
  // for different sizes of arrays on the GPU. Also note that there could be
  // multiple arrays, therefore in this case we take the first element from
  // the array of input arrays and of the array of each output arrays.
  if(x < size.x && y < size.y && z < size.z) {
    int a = in[0][idx2(x, y, inP)];
    if(x < size.x - 1) {
      a += in[0][idx2(x + 1, y, inP)];
    }
    out[0][idx2(x, y, outP)] = a;
}

// Instead of allocating memory, transferring data and calling the kernel, a
// single call to the library function is made. This does all the work. Note
// that the last four arguments will be provided by the program analysis and
// the performance model.
runScalable(size, inputCount, inputs, outputCount, outputs, kernel,
  tileSize, blockSize, streamCount, padding);
```

When compared to the naive CUDA implementation, there is very little added complexity in the creation of the kernel while the rest of the program has become much simpler. The library takes care of (and hides for the user):

- Allocating and deallocating GPU memory.
- Setting up and using streams that are used in CUDA to manage concurrency.
- Setting up and using events, that can allow the CPU to wait until the GPU has finished an operation.

- Determining the number of tiles.
- Setting up a loop that creates the pipeline that:
  - Calls the kernel with address corrections.
  - Orders the operations sent to the GPU so that overlap is maximized, yet no tiles interfere with each other.
  - Handles the start-up and shutdown phases properly.
- Sends the data associated with each tile by:
  - Determining the location of each tile.
  - Adding padding for input and removing padding in corner cases to avoid segmentation faults.
  - Determining if there is contiguity in data and exploit that if possible.
  - Copying the correct data in several chunks to and from the GPU.

Note that the library can easily be used with the basic and buffered schemes by calling *runBasic* or *runBuffered* instead of *runScalable*. It is likely that when generalizing the library to different program classes, its usage will become more complex.

Chapter 7

Results

This chapter discusses both the performance of the various tiling/pipelining schemes, as well as the accuracy of the performance model. We do so by showing the results of using the library and the model on the prototypical example programs introduced in section 3.3 and discussing them. However we limit discussion to the one-dimensional moving average and the three-dimensional Jacobi stencil programs. While we have performed measurements on the two-dimensional emboss program as well, the results fall right in between the other programs and thus do not provide any further insights. The raw data for all three programs is included in appendix A.

All measurements were performed on an Intel i5 (2.80 GHz quad core) CPU with either a Tesla C2050 (1.15 GHz per each of 448 cores, dual copy engines) or a GeForce GTX 460 (1.35 GHz per each of 336 cores, single copy engine) GPU. The CUDA environment was initialized beforehand, all input and output data was page-locked and registered with CUDA beforehand and all of this data was fully allocated (malloc is normally lazy). Measurements were taken by using a single CPU-based timer (*clock_gettime* in the C standard library). Results were only accurate down to milliseconds, so all results are rounded to them.

7.1 Performance of optimizations

In this section the performance of the various tiling/pipelining schemes described in section 6.2 is shown. We vary tile size, kernel execution times and padding size. Other parameters are kept constant, such as array size which is not interesting to vary and are always 64 million elements total. We also keep thread block size constant[1] (1000 for moving average and 400x1x1 for Jacobi stencil).

[1] The GPU groups its threads in blocks. This is an important parameter when optimizing kernels, but not interesting for our purposes.

7.1.1 Tile size

The results of the performance model are used to both determine the best scheme and the optimal tile size for that scheme. In this section we discuss the effects of tile size on performance.

Moving average

The graph in figure 7.1 shows the run times based on chosen tile size for the one-dimensional moving average program when run on the Tesla. Note that the x-axis of this graph uses a logarithmic scale (base 10). The line that is labeled *sequential* indicates how long it takes to run the program on the CPU without any optimizations. The line labeled *naive* shows the run time for the naive approach, where we simply move all the input data to the GPU, run the kernel and then move all the output data back. Neither actually uses tiles, which is why they are both horizontal lines. We want the optimizations to perform better than the naive implementation.

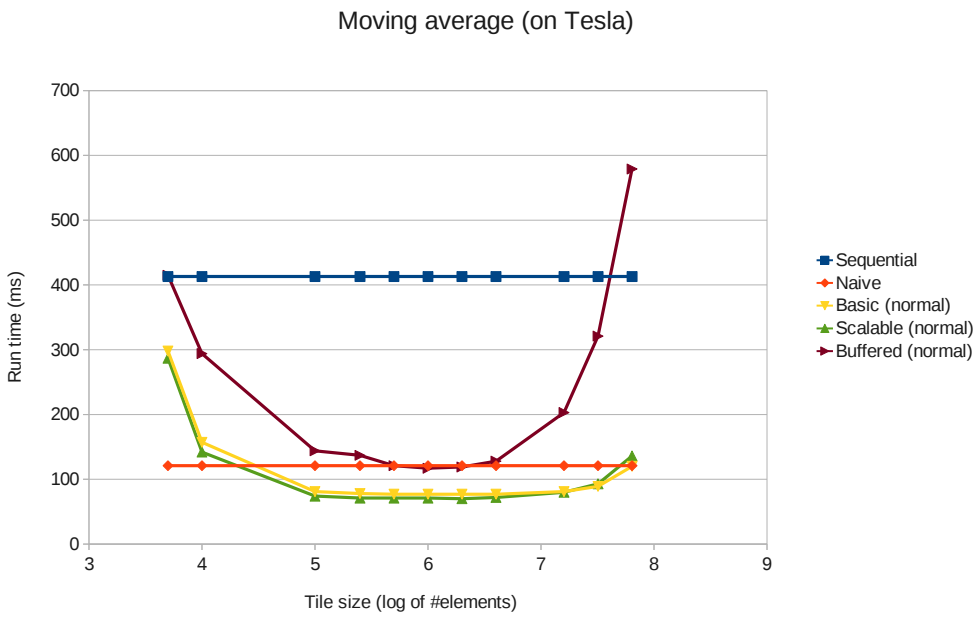

Figure 7.1: The effects of tile size on run time. The array has 64 million elements; padding is 4 elements on each side; blocks are 1000 elements wide.

The basic and scalable schemes perform well and almost identical. The scalable one tends to be a bit faster, as per array this scheme allocates three times the tile size instead of one time the full array size. At the last point the tile size is equal to the full array size, which is why the basic scheme performs

better here. The smallest tile size measured is 5000 elements; with smaller tiles the performance drops significantly as the data transfers become too small. The buffered scheme performs poorly as it performs more data transfers and has a more complex pipeline. This scheme was created to remedy problems in multi-dimensional programs, so there is no actual benefit to using it in this case.

Of the available schemes, the scalable one performs best for this program. The optimal tile size is one million elements, at which point it takes 71 ms to run the program. The naive implementation takes 121 ms, so this is a speed-up of 1.70x. The graph in figure 7.2 shows the results for the same program run on the GTX; these are very similar. In this case the scalable version is also the best and the optimal tile size is again one million elements. With a naive run time of 138 ms and a run time of 85 ms for the scalable scheme, the speed-up is 1.58x. The slightly lower relative speed-up was expected, as the GTX has only a single copy engine, while the Tesla has dual copy engines.

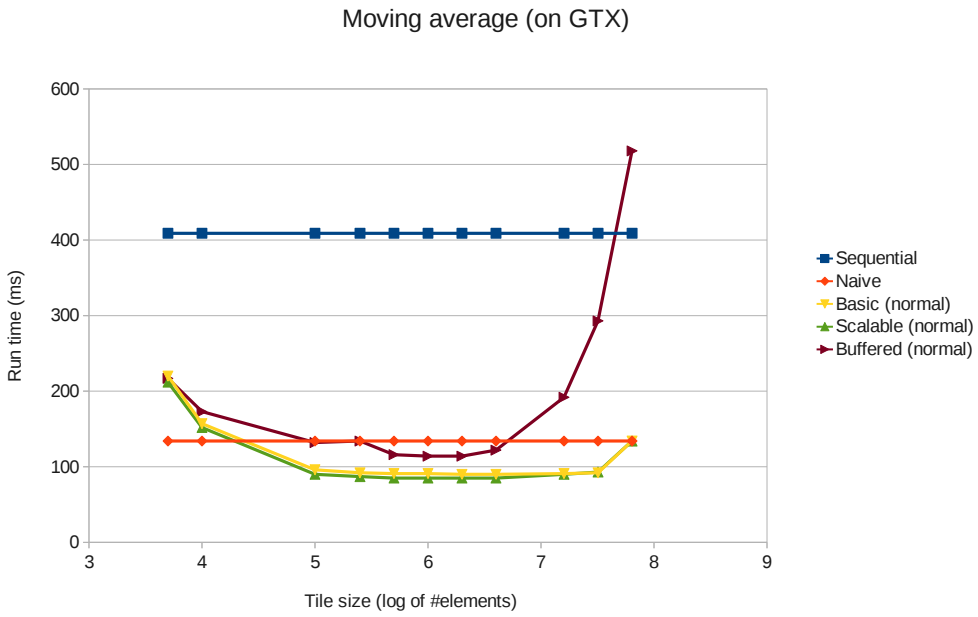

Figure 7.2: The same graph as in figure 7.1, but for the GTX instead of the Tesla.

Jacobi stencil

As we mentioned in section 6.2.3, multi-dimensional programs can suffer from not having tiles that are contiguous in physical memory. The Jacobi stencil is such a program as it is three-dimensional. We wanted to run this on 64 million-element arrays, if we make a cube out of those we get a 400x400x400 element

array. We are using C floats so 400 elements take 1600 bytes; transferring only those 400 elements at a time is not a viable option.

The graph in figure 7.3 shows that for the Jacobi stencil program, the normal scalable scheme (i.e. the one that copies data in small chunks, even though this data is contiguous due to the chosen shape of the tile) performs extremely poorly. The buffered scheme performs much better, but never performs better than the naive implementation. Finally the adapted scalable scheme that uses contiguity of data to merge data transfers works very well. We do not show the basic scheme as it performs almost identically to the scalable one. At the optimal tile size of 400x400x25 it takes only 74 ms, which is a speed-up of 1.49x compared to the 110 ms the naive version takes. Again the GTX has a very similar graph, so we do not show it here; it has the same optimal tile size and the speed-up is 1.36x (120 vs. 88 ms).

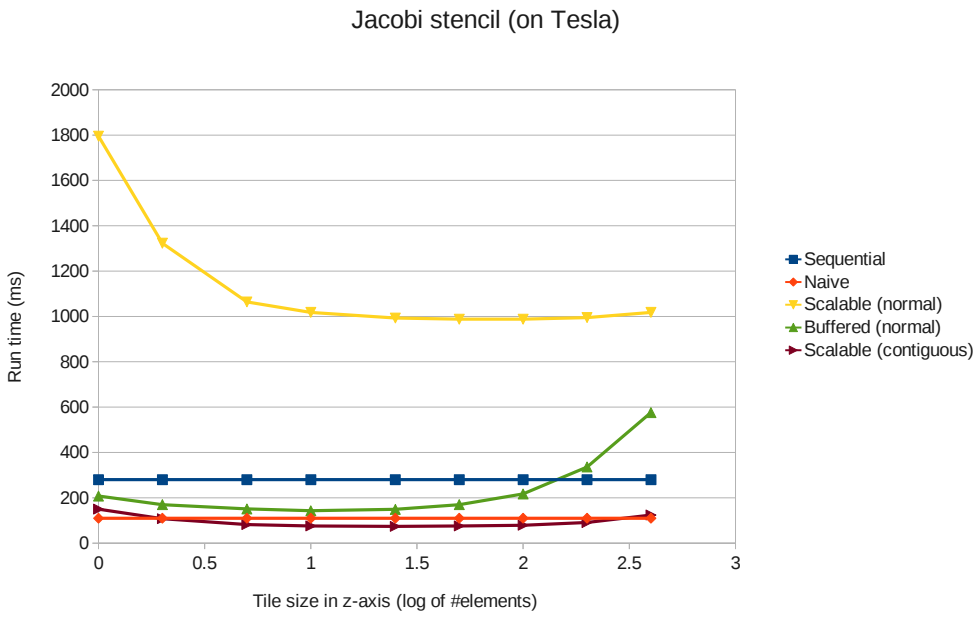

Figure 7.3: Not every scheme deals with three dimensions well. The array has 400x400x400 elements while the tile has 400x400xZ elements (z-axis is variable); padding is 1 element on every side; blocks are 400x1x1 elements wide.

Note that this program has a lower relative speed-up than the moving average program. This is because this program has to transfer much more padding per tile. At the optimal tile size it uses 400x400x2 elements of padding per tile (the padding in the X and Y-axes falls away because the tile is as large as the array in those axes). There are 16 tiles in total making for a total of 5,120,000 elements of padding, which is significant (8% of the total size). The moving

average uses only 8 elements of padding per tile and has a total of 64 tiles at the optimal tile size. This means a total of 512 elements of padding, which is negligible. The large amount of padding for the Jacobi stencil program (which only has a single element of padding in each direction) indicates that reducing or eliminating the redundant data transfers of padding data is a good candidate for future work.

7.1.2 Kernel size

Not all programs in the stenciled map/zip class benefit equally from using the tiling/pipelining schemes. By running variations of our prototype examples where the kernel is executed a variable number of times, we show the effects of the ratio between data transfer times and kernel execution times on performance and relative speed-up.

Moving average

The graph in figure 7.4 shows the run time for the moving average program with tiles of one million elements. The number of times the kernel is repeated is varied, giving a good indication of both the absolute performance of the schemes as well as the relative performance compared to the naive implementation. Again the X-axis is a logarithmic scale with base 10.

It is interesting to see with more kernel iterations the three schemes perform almost completely identically. This is because at that point, the execution of the kernel takes so long that it effectively hides the overlapping data transfers. Conversely with few kernel iterations the data transfers completely hide the kernel executions and because the data transfer time is not varied, this results in a horizontal line. This horizontal line is higher for the buffered scheme as this scheme does additional data transfers.

For very small kernels, their activity is hidden because they are overlapped with the larger data transfers. But as those kernels are so small, there is little speed-up gained from overlapping them. For very large kernels, the data transfers become fully hidden and we always get the same amount of speed-up in absolute terms. But as those data transfers are relatively small compared to the kernel, the relative speed-up is also quite low. The sweet spot in terms of relative speed-up is where the data transfers (with overlap between them) and the kernels take roughly the same amount of time.

For the moving average program, the relative speed-up is largest at three kernel iterations; it is 2.06x (171 vs. 83 ms). This is actually quite close to the theoretical limit. While the naive theoretical limit is 3x when all three engines are working at peak occupancy, we have already shown in section 6.4.4 that this is not realistic. As the dual copy engines only give a speed-up of 1.26x instead of 2x, we can say that a more accurate theoretical limit is 2.26x.

It is a bit hard to conclude from the measurements taken how close to this theoretical limit the scalable scheme can get, as measurements were only taken for integral numbers of iterations. At two iterations the relative speed-up is

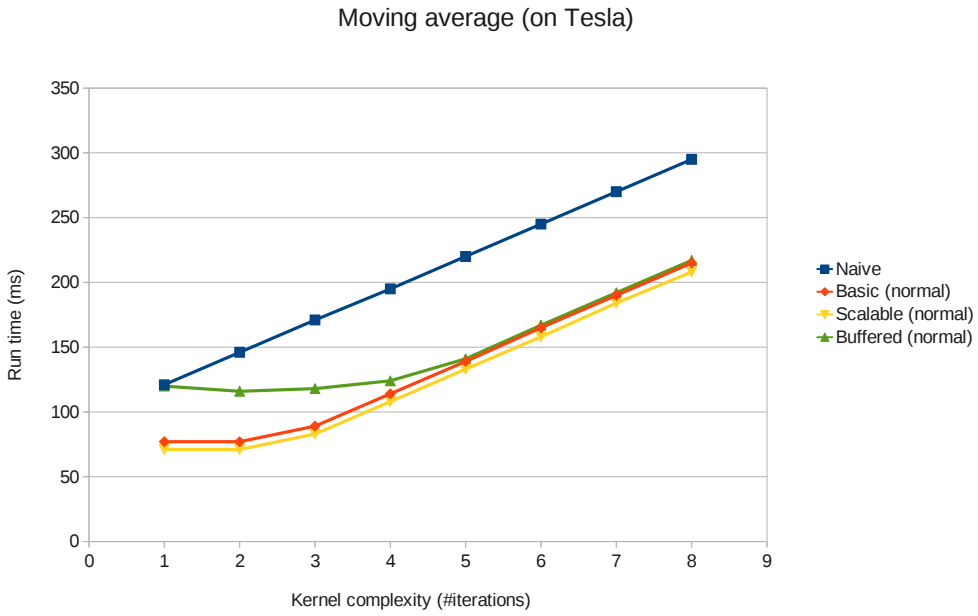

Figure 7.4: The effects of kernel run time on total run time. The array has 64 million elements; tile size is 1 million elements; padding is 4 elements on each side; blocks are 1000 elements wide.

2.056x; at three iterations it is 2.060x; at four iterations it is 1.806x. Still it is clear that the optimization is very close to the theoretically expected speed-up.

For the GTX the best number of iterations was two, which gave a speed-up of 1.99x. As the theoretical speed-up with a single copy engine is 2x, this is very good.

Jacobi stencil

The results for the Jacobi stencil are interesting as well. Figure 7.5 shows the graph for the most interesting cases. Here we once again see that the normal scalable scheme performs very poorly. The data transfers are so inefficient that it takes a very large kernel for the kernel to outweigh the data transfers. Even when it does, the performance is much poorer compared to the other schemes. This is likely due to synchronization issues that occur with very small data transfers, which means that those data transfers are never truly overlapped with kernel execution and thus never truly hidden. Figure 7.6 shows the same graph for lower iteration numbers without the normal scalable scheme; it again shows the horizontal lines for few iterations, and diagonals for many iterations. The optimal number of iterations for the Tesla is five, giving a 2.02x speed-up. For the GTX the best speed-up is 1.81x at 4 iterations.

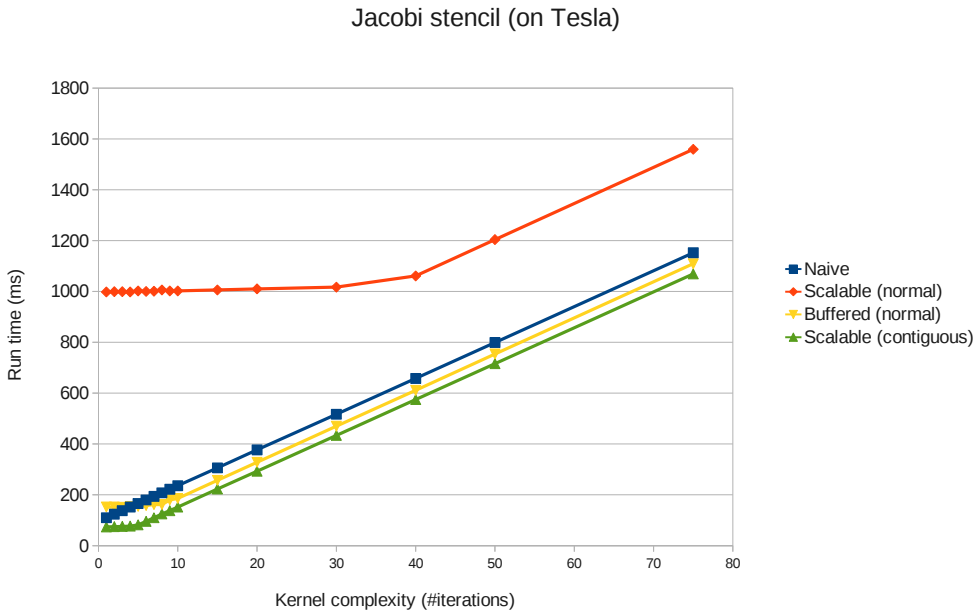

Figure 7.5: The effects of kernel run time on total run time. The array has 400x400x400 elements while the tile has 400x400x25 elements; padding is 1 element on every side; blocks are 400x1x1 elements wide.

7.1.3 Padding/stencil size

As we have already discussed in section 6.2.2, larger amounts of padding lead to extra data transfer in the schemes we have implemented. To illustrate we have made a variation of the three-dimensional Jacobi stencil program that allowed us to easily change the amount of padding. The results are shown in figure 7.7. It shows that the effects are linear and at roughly 13 elements of padding on each side using tiling and pipelining becomes useless as the naive version is faster. It will definitely pay off to remove the redundant data transfers, even with small padding sizes if this does not introduce significant overhead.

7.1.4 Insights

It is clear that tiling and pipelining GPU-based programs pays off. Speed-ups of over 2x can be expected for some programs. For the test programs we have used, the scalable scheme that merges transfers of contiguous data works best. Removing redundant data transfers for padded data seems to be a good next step, as this can reduce overhead of programs, even with low amounts of padding in multi-dimensional cases.

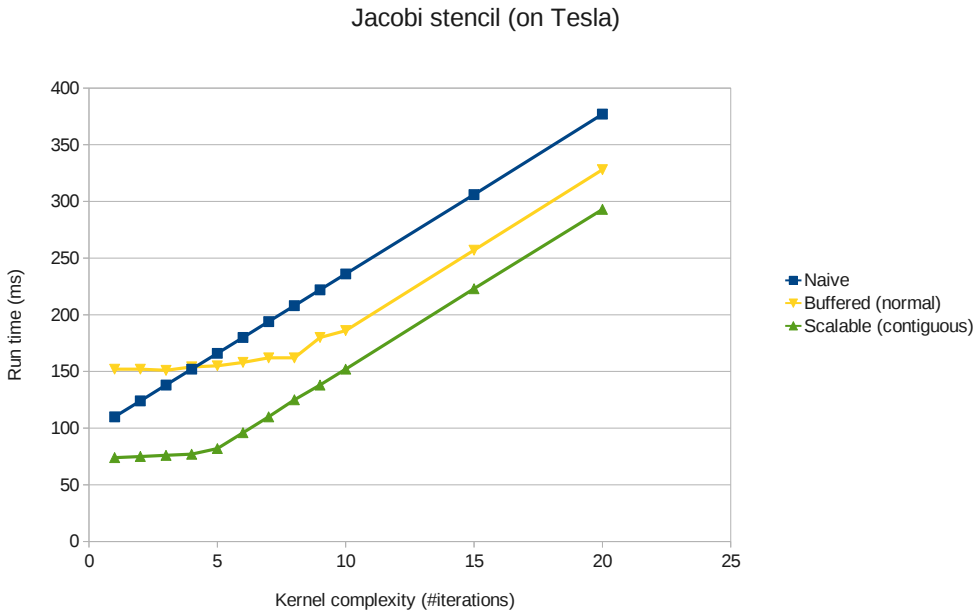

Figure 7.6: The effects of kernel run time on total run time. The array has 400x400x400 elements while the tile has 400x400x25 elements; padding is 1 element on every side; blocks are 400x1x1 elements wide.

7.2 Accuracy of the performance model

This section shows the accuracy of the performance prediction model with respect to the run time of programs that are using the scalable scheme that merges transfers for contiguous data. The raw data in appendix A holds the results for the basic scheme too, as well as for variations that do not merge transfers if possible. The buffered scheme is not implemented, so there are no results for it.

7.2.1 Moving average

Figure 7.8 shows the graphs for the moving average program for both the Tesla and the GTX. These graphs vary the tile size, so the measured results are the same as in in section 7.1.1. The predictions made by the model are not always very accurate. That is fine though, as long as the predictions both correctly reveal the optimal tile size and are accurate for this optimal tile size.

We can see that the model does both very well. For the Tesla, the model predicts that the optimal tile size is any of 250,000, 500,000 or 1,000,000 elements that all have a run time of 69 ms. The measured values for these samples are all 71 ms, which is in fact the lowest measured run time. The prediction is only 2.82% off, so it is very accurate. For the GTX it predicts that 500,000,

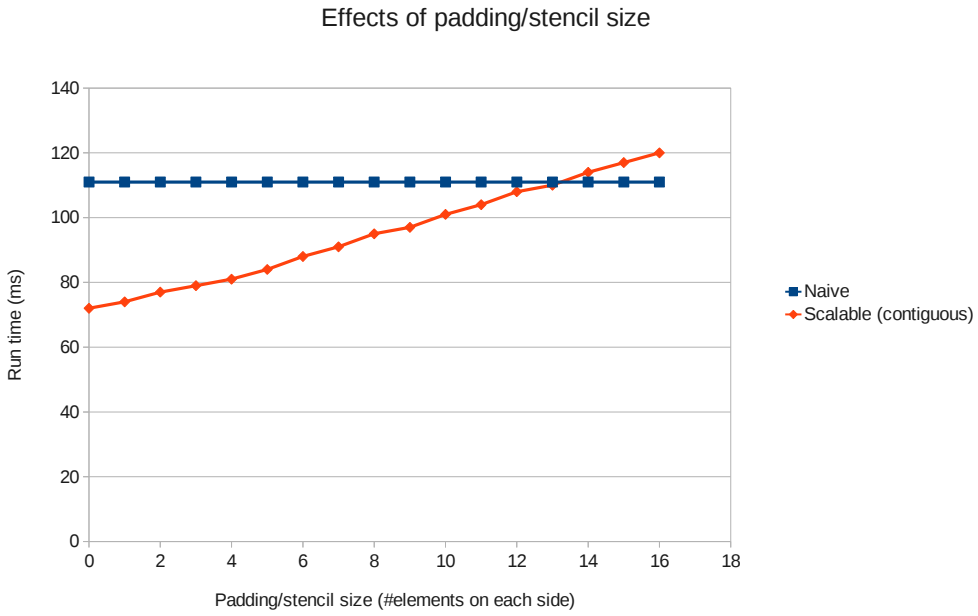

Figure 7.7: The run time of a three-dimensional program based on padding/stencil size on the Tesla. The program is a variation of Jacobi Stencil. The array has 400x400x400 elements while the tile size is 400x400x25 elements; padding is 1 element on every side; blocks are 400x1x1 elements wide.

1,000,000 and 2,000,000 elements will all result in the minimal run time of 86 ms. The measurements show that this is exactly what happens. So this is even more accurate.

7.2.2 Jacobi stencil

For the Jacobi stencil, the results are shown in figure 7.9. Again accuracy is very high. For the Tesla, it predicts that 400x400x25 is the optimal tile size with a run time of 75 ms. It actually takes 74 ms, so it is only 1.35% off. For the GTX it predicts that either 400x400x25 or 400x400x50 is the optimal tile size, taking 89 ms. The actual result show that both of these tile sizes run for 88 ms. This is as little as 1.14% off. So again the results are very accurate.

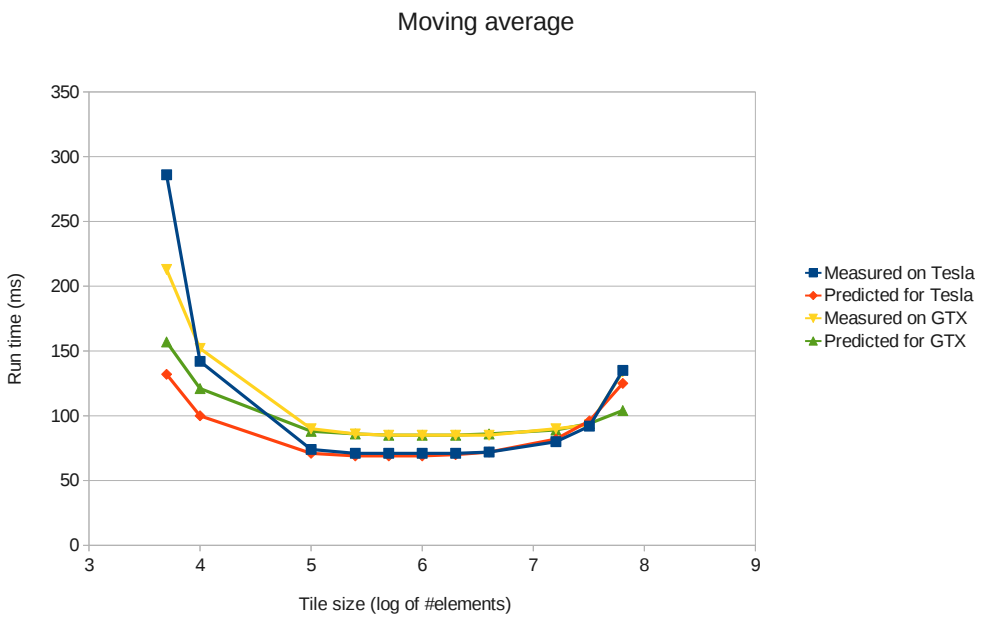

Figure 7.8: The accuracy of the model for moving average for various tile sizes. The array has 64 million elements; padding is 4 elements on each side; blocks are 1000 elements wide.

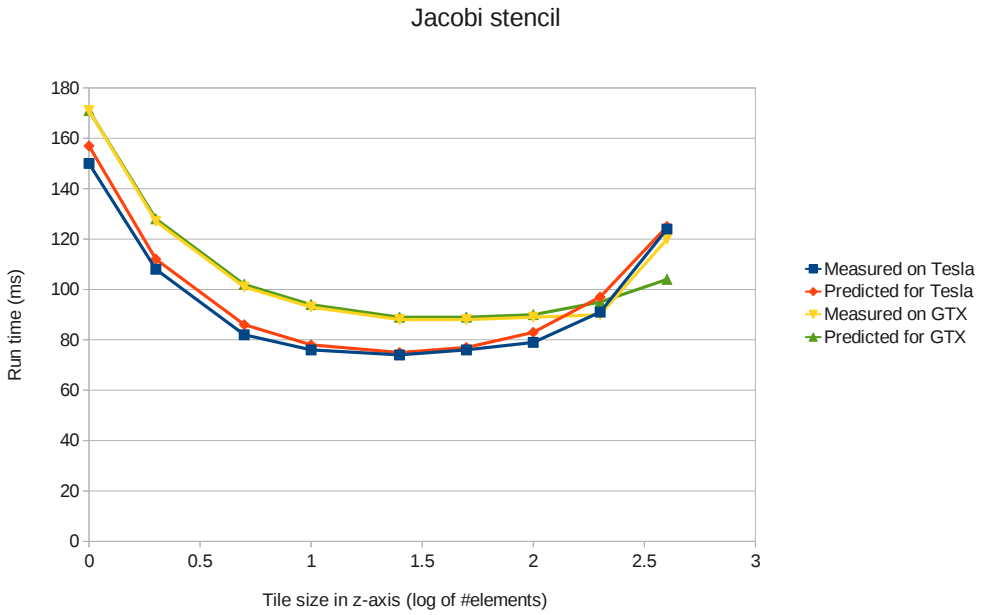

Figure 7.9: The accuracy of the model for Jacobi stencil for various tile sizes. The array has 400x400x400 elements while the tile has 400x400xZ elements (z-axis is variable); padding is 1 element on every side; blocks are 400x1x1 elements wide.

Chapter 8

Conclusion

From the results obtained it is very clear that in the context of GPU programming, tiling and pipelining is a very useful optimization. Speed-ups of over 2x are possible for some programs, while for typical programs speed-ups of over 1.5x can be expected. This is very useful, especially given the fact that for many programs that one wants to run on the GPU, the transfer of data takes more time than the actual calculations.

The performance of the optimizations can be predicted accurately The performance prediction model has, for the typical programs that were tested, predicted the optimal tile sizes perfectly. Furthermore, the values that are predicted at the optimal tile sizes are never more than three percent off.

For the stenciled map/zip class, the implementation details can be hidden from the end user. A recipe can be given to ease the transformation of the original sequential source code to the tiling/pipelining GPU version.

8.1 Future work

While this thesis features some interesting work, there is more to be done. Expanding up this work can be done by doing one of the following:

- The optimizations can be extended to new program classes that are more difficult to tile. This is not trivial at all, but much work has already been done on this in other contexts [12, 13].
- The library can be expanded so that it is less restrictive, e.g. using any combination of data types instead of only floats and processing input and output arrays of different sizes.
- The work can be integrated with the tools created by Vector Fabrics, so that the users of those tools can actually benefit from it.
- Fusing multiple kernels into a single pipeline can make more complex input programs much faster (as discussed in section 2.8).

- Current implementations transfer the data that is part of the padding of a tile multiple times. The basic scheme can be modified to eliminate these extra transfers. The scalable and buffered schemes may also be adapted to eliminate these transfers, but this will be more difficult.
- The naive approach does not work very well if the arrays used are nested (i.e. arrays of pointers to arrays instead of a single flat array). We expect that in general it is better to use flat arrays instead, but that may not always be an option. The buffered scheme will likely outperform the naive approach in this case. It will be interesting to see by how much.
- The performance prediction model has a small gap in it when it comes to accuracy. It does not take into account that at the edges and corners of the input arrays, the padding falls outside of the boundaries of the arrays and is thus not transfered. The model can be made more accurate by taking this into account as well. The effects on the typical examples used in this thesis will be minimal, but for programs that have a lot of padding, this may make a significant difference in accuracy.

Bibliography

[1] AMD. *Accelerated Parallel Processing OpenCL programming guide*, August 2011.

[2] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for cuda. In Wen-mei W. Hwu, editor, *GPU Computing Gems, Jade Edition*. Morgan Kaufman Publishers, October 2011.

[3] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *Proceedings of the POPL 2007 Workshop on Declarative Aspects of Multicore Programming, DAMP 2007, Nice, France, January 16, 2007*, 2007.

[4] Jack W. Davidson and Sanjay Jinturkar. Memory access coalescing: a technique for eliminating redundant memory accesses. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 186–195, New York, NY, USA, 1994. ACM.

[5] Isaac Gelado, Javier Cabezas, Nacho Navarro, John E. Stone, Sanjay Patel, and Wen-mei W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2010)*, March 2010.

[6] Mark Harris. *Parallel Prefix Sum (Scan) with CUDA*. NVIDIA, April 2007.

[7] Lee W. Howes, Anton Lokhmotov, Alastair F. Donaldson, and Paul H.J. Kelly. Deriving efficient data movement from decoupled access/execute specifications. In *Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, volume 5409 of *Lecture Notes in Computer Science*, pages 168–182. Springer, 2009.

[8] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic cpu-gpu communication management and optimization. In *PLDI'11 Proceedings of the 2011 ACM Conference on Programming Language Design and Implementation*.

[9] Călin Juravle. Automatic program analysis for data parallel kernels. Master's thesis, Utrecht University, July 2011.

[10] Gabriele Keller, Manual M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. 2010.

[11] Khronos Group. *The OpenCL specification*, June 2011.

[12] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 145–156, New York, NY, USA, 2000. ACM.

[13] Ben Lippmeier, Gabriele Keller, and Simon Peyton Jones. Efficient parallel stencil convolution in haskell. 2011.

[14] NVIDIA. *Quadro dual copy engines*, October 2010.

[15] NVIDIA. *CUDA C programming guide*, May 2011.

[16] Andreas Resios. GPU performance prediction using parametrized models. Master's thesis, Utrecht University, July 2011.

[17] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[18] Shubhabrata Sengupta, Mark Harris, and Michael Garland. *Efficient Parallel Scan Algorithms for GPUs.* NVIDIA, December 2008.

Appendix A

Raw results

This appendix has all the raw data that was used to get the results in chapter 7. All measurements and predictions are in milliseconds.

			Normal			Contiguous		
Tile size	Sequential	Naive	Basic	Scalable	Buffered	Basic	Scalable	Buffered
5k	413	121	298	287	415	305	286	414
10k	413	121	157	142	294	151	142	294
100k	413	121	81	74	144	81	74	143
250k	413	121	78	71	137	78	71	136
500k	413	121	77	71	121	77	71	121
1M	413	121	77	71	117	77	71	117
2M	413	121	77	70	119	77	71	119
4M	413	121	77	72	128	77	72	128
16M	413	121	81	80	203	81	80	202
32M	413	121	89	93	321	89	92	321
64M	413	121	120	136	579	120	135	472

Table A.1: Performance of moving average on the Tesla based on tile size. The array has 64 million elements; padding is 4 elements on each side; blocks are 1000 elements wide.

			Normal			Contiguous		
Tile size	Sequential	Naive	Basic	Scalable	Buffered	Basic	Scalable	Buffered
5k	409	134	220	212	217	220	213	216
10k	409	134	157	152	173	157	152	175
100k	409	134	96	90	132	96	90	133
250k	409	134	92	87	134	92	86	133
500k	409	134	91	85	116	91	85	116
1M	409	134	91	85	114	91	85	114
2M	409	134	90	85	114	90	85	113
4M	409	134	90	85	122	90	85	121
16M	409	134	91	90	192	92	90	192
32M	409	134	92	93	293	93	93	293
64M	409	134	134	134	518	134	134	428

Table A.2: Performance of moving average on the GTX based on tile size. The array has 64 million elements; padding is 4 elements on each side; blocks are 1000 elements wide.

	Tesla				GTX			
	Normal		Contiguous		Normal		Contiguous	
Tile size	Basic	Scalable	Basic	Scalable	Basic	Scalable	Basic	Scalable
5k	139	132	139	132	164	157	164	157
10k	106	100	106	100	127	121	127	121
100k	78	71	78	71	94	88	94	88
250k	76	69	76	69	92	86	92	86
500k	75	69	75	69	91	85	91	85
1M	75	69	75	69	91	85	91	85
2M	76	70	76	70	91	85	91	85
4M	77	72	77	72	91	86	91	86
16M	84	82	84	82	91	89	91	89
32M	93	96	93	96	91	94	91	94
64M	111	125	111	125	91	104	91	104

Table A.3: Predictions of moving average based on tile size. The array has 64 million elements; padding is 4 elements on each side; blocks are 1000 elements wide.

			Normal			Contiguous		
Iterations	Sequential	Naive	Basic	Scalable	Buffered	Basic	Scalable	Buffered
1	413	121	77	71	120	77	71	117
2	822	146	77	71	116	77	71	117
3	1233	171	89	83	118	89	83	117
4	1647	195	114	108	124	114	108	124
5	2058	220	139	133	141	139	133	140
6	2466	245	165	158	167	164	158	167
7	2898	270	190	184	192	190	184	194
8	3280	295	215	208	217	215	209	216
9	3699	319	240	234	242	240	234	242
10	4116	344	264	257	265	264	257	265
15	6174	468	389	382	390	389	382	390
20	8223	592	514	507	515	514	507	515
30	12307	840	764	757	765	764	757	765
40	16457	1088	1014	1007	1015	1014	1007	1015
50	20573	1336	1264	1257	1266	1264	1257	1265
75	30820	1956	1889	1882	1890	1889	1882	1890
100	41113	2576	2514	2507	2515	2514	2507	2515
150	61775	3817	3764	3757	3765	3763	3757	3765
200	82191	5057	5013	5007	5015	5013	5006	5014

Table A.4: Performance of kernel-iterated variant of moving average on the Tesla. The array has 64 million elements; tile size is 1 million elements; padding is 4 elements on each side; blocks are 1000 elements wide.

			Normal			Contiguous		
Iterations	Sequential	Naive	Basic	Scalable	Buffered	Basic	Scalable	Buffered
1	409	135	91	85	114	91	85	114
2	818	171	92	86	114	92	86	114
3	1227	206	124	118	125	124	118	125
4	1636	242	160	154	161	160	154	162
5	2045	278	196	190	197	196	190	197
6	2454	314	232	226	233	232	226	233
7	2864	350	268	262	269	268	262	269
8	3273	386	304	298	305	304	298	305
9	3682	422	340	334	341	340	334	341
10	4091	458	376	370	377	376	370	377
15	6137	638	556	549	557	556	549	557
20	8183	817	735	729	737	735	729	736
30	12274	1177	1095	1089	1097	1095	1089	1096
40	16366	1537	1455	1449	1456	1455	1449	1456
50	20458	1896	1815	1809	1817	1815	1809	1816
75	30685	2795	2715	2709	2716	2715	2709	2716
100	40918	3694	3616	3609	3616	3615	3609	3616
150	61375	5493	5415	5409	5416	5415	5409	5416
200	81829	7291	7215	7209	7216	7215	7209	7216

Table A.5: Performance of kernel-iterated variant of moving average on the GTX. The array has 64 million elements; tile size is 1 million elements; padding is 4 elements on each side; blocks are 1000 elements wide.

			Normal			Contiguous		
Tile size	Sequential	Naive	Basic	Scalable	Buffered	Basic	Scalable	Buffered
1	237	117	274	260	410	192	185	421
2	237	117	188	180	260	137	129	270
5	237	117	132	126	175	99	91	183
10	237	117	123	117	145	90	83	155
25	237	117	115	109	128	81	74	138
50	237	117	112	107	126	79	72	125
100	237	117	113	107	132	77	71	118
200	237	117	102	96	133	77	71	116
400	237	117	103	97	142	78	72	124
800	237	117	106	101	168	79	74	141
2k	237	117	113	111	217	82	80	200
4k	237	117	135	140	335	88	91	315
8k	237	117	161	177	582	116	131	457

Table A.6: Performance of emboss on the Tesla with tiles of 8000xY elements (the tile size in the table indicates the size of the tile on the Y-axis). The array has 8000x8000 elements; padding is 1 element on every side; blocks are 1000x1 elements wide.

			Normal			Contiguous		
Tile size	Sequential	Naive	Basic	Scalable	Buffered	Basic	Scalable	Buffered
1	235	129	298	292	268	259	251	275
2	235	129	214	209	197	175	168	207
5	235	129	168	162	147	124	117	157
10	235	129	161	154	132	107	100	142
25	235	129	148	142	123	96	90	134
50	235	129	137	132	121	93	87	122
100	235	129	134	128	132	91	85	114
200	235	129	133	128	133	91	85	114
400	235	129	133	128	140	90	85	119
800	235	129	133	129	157	90	86	135
2k	235	129	135	133	208	90	88	193
4k	235	129	145	145	314	90	90	292
8k	235	129	171	172	533	129	129	417

Table A.7: Performance of emboss on the GTX with tiles of 8000xY elements (the tile size in the table indicates the size of the tile on the Y-axis). The array has 8000x8000 elements; padding is 1 element on every side; blocks are 1000x1 elements wide.

	Tesla				GTX			
	Normal		Contiguous		Normal		Contiguous	
Tile size	Basic	Scalable	Basic	Scalable	Basic	Scalable	Basic	Scalable
1	251	245	201	194	265	259	220	214
2	183	176	137	131	201	194	155	149
5	142	135	100	93	162	156	116	110
10	128	122	87	80	149	143	104	97
25	120	113	80	73	141	135	96	90
50	118	111	77	71	139	133	93	87
100	117	110	76	70	137	131	92	86
200	117	110	76	70	137	131	91	86
400	118	112	77	71	136	131	91	86
800	120	116	78	74	136	132	91	87
2k	129	128	84	82	136	135	91	89
4k	144	148	93	97	136	140	91	94
8k	173	187	111	125	136	149	91	104

Table A.8: Predictions of emboss with tiles of 8000xY elements (the tile size in the table indicates the size of the tile on the Y-axis). The array has 8000x8000 elements; padding is 1 element on every side; blocks are 1000x1 elements wide.

			Normal			Contiguous		
Iterations	Sequential	Naive	Basic	Scalable	Buffered	Basic	Scalable	Buffered
1	237	117	113	107	135	78	71	119
2	473	137	113	107	133	78	71	117
3	709	157	113	107	133	78	72	117
4	948	177	121	114	134	96	90	119
5	1183	198	129	123	139	117	111	125
6	1420	218	143	133	140	137	132	138
7	1658	239	160	155	159	159	152	159
8	1892	259	182	174	180	180	173	179
9	2155	279	202	195	201	200	193	200
10	2365	299	221	214	220	220	213	220
15	3559	401	324	318	324	323	317	323
20	4741	503	427	421	427	426	420	426
30	7084	706	633	627	634	633	626	633
40	9472	910	840	834	840	839	832	839
50	11803	1113	1046	1041	1047	1045	1039	1045
75	17783	1621	1562	1557	1563	1561	1555	1561
100	23682	2130	2078	2073	2080	2077	2071	2077
150	35520	3147	3109	3106	3112	3108	3102	3109
200	47332	4163	4141	4138	4144	4140	4134	4140

Table A.9: Performance of kernel-iterated variant of Emboss on the Tesla. The array has 8000x8000 elements; tile size is 8000x100 elements; padding is 1 element on every side; blocks are 1000x1 elements wide.

			Normal			Contiguous		
Iterations	Sequential	Naive	Basic	Scalable	Buffered	Basic	Scalable	Buffered
1	235	129	134	128	134	92	86	114
2	471	158	135	129	132	92	86	114
3	707	186	135	129	133	104	98	116
4	942	215	145	139	136	132	126	132
5	1179	243	162	156	161	161	155	161
6	1414	271	190	184	190	189	183	189
7	1649	300	219	213	218	218	212	218
8	1885	328	247	241	247	246	240	246
9	2120	356	276	270	275	275	269	275
10	2357	385	304	299	304	303	298	303
15	3536	526	447	441	447	446	440	446
20	4714	668	589	584	589	589	583	588
30	7072	952	875	869	875	874	868	874
40	9429	1235	1160	1154	1160	1159	1153	1159
50	11786	1518	1445	1440	1445	1444	1438	1444
75	17681	2227	2158	2153	2159	2157	2151	2157
100	23574	2936	2871	2866	2872	2870	2863	2870
150	35369	4353	4296	4293	4298	4295	4289	4296
200	47242	5770	5722	5719	5725	5721	5715	5722

Table A.10: Performance of kernel-iterated variant of Emboss on the GTX. The array has 8000x8000 elements; tile size is 8000x100 elements; padding is 1 element on every side; blocks are 1000x1 elements wide.

			Normal			Contiguous		
Tile size	Sequential	Naive	Basic	Scalable	Buffered	Basic	Scalable	Buffered
1	280	110	1904	1902	393	920	907	350
2	280	110	1390	1378	249	485	467	224
5	280	110	1082	1074	171	239	236	156
10	280	110	980	973	145	160	152	130
25	280	110	931	925	130	106	100	132
50	280	110	915	908	131	91	84	142
100	280	110	946	940	134	85	79	145
200	280	110	980	976	139	80	74	150
400	280	110	999	994	150	79	74	132

Table A.11: Performance of Jacobi stencil on the Tesla with tiles of 400xYx25 elements (the tile size in the table indicates the size of the tile on the Y-axis). The array has 400x400x400 elements; padding is 1 element on every side; blocks are 400x1x1 elements wide.

			Normal			Contiguous		
Tile size	Sequential	Naive	Basic	Scalable	Buffered	Basic	Scalable	Buffered
1	276	120	1998	1994	281	1090	1082	242
2	276	120	1485	1480	197	594	587	174
5	276	120	1179	1173	152	300	293	128
10	276	120	1083	1077	131	204	198	118
25	276	120	1024	1018	125	140	134	128
50	276	120	1004	998	128	113	107	136
100	276	120	994	988	134	103	97	145
200	276	120	989	983	137	98	92	152
400	276	120	986	981	147	94	88	127

Table A.12: Performance of Jacobi stencil on the GTX with tiles of 400xYx25 elements (the tile size in the table indicates the size of the tile on the Y-axis). The array has 400x400x400 elements; padding is 1 element on every side; blocks are 400x1x1 elements wide.

	Tesla				GTX			
	Normal		Contiguous		Normal		Contiguous	
Tile size	Basic	Scalable	Basic	Scalable	Basic	Scalable	Basic	Scalable
1	2144	2137	1028	1022	2127	2121	1151	1145
2	1541	1535	552	545	1594	1588	621	615
5	1180	1174	266	259	1274	1268	303	297
10	1060	1053	172	165	1168	1161	198	192
25	989	982	115	109	1104	1098	135	129
50	967	960	97	90	1082	1076	114	108
100	958	952	88	81	1072	1066	104	98
200	960	954	84	78	1066	1061	99	94
400	973	967	80	75	1064	1059	94	89

Table A.13: Predictions of Jacobi stencil with tiles of 400xYx25 elements (the tile size in the table indicates the size of the tile on the Y-axis). The array has 400x400x400 elements; padding is 1 element on every side; blocks are 400x1x1 elements wide.

			Normal			Contiguous		
Tile size	Sequential	Naive	Basic	Scalable	Buffered	Basic	Scalable	Buffered
1	280	110	1848	1858	220	953	971	195
2	280	110	1349	1346	184	509	504	164
5	280	110	1073	1065	157	239	231	138
10	280	110	1034	1026	146	145	137	134
25	280	110	1004	998	150	102	97	154
50	280	110	994	990	171	91	86	180
100	280	110	989	988	217	88	86	228
200	280	110	991	996	335	91	95	346
400	280	110	1001	1018	580	109	124	501

Table A.14: Performance of Jacobi stencil on the Tesla with tiles of 400xYx400 elements (the tile size in the table indicates the size of the tile on the Y-axis). The array has 400x400x400 elements; padding is 1 element on every side; blocks are 400x1x1 elements wide.

			Normal			Contiguous		
Tile size	Sequential	Naive	Basic	Scalable	Buffered	Basic	Scalable	Buffered
1	276	120	2000	1964	209	1148	1116	183
2	276	120	1474	1454	178	620	602	158
5	276	120	1160	1148	153	306	296	136
10	276	120	1053	1045	146	198	190	136
25	276	120	990	984	148	133	127	151
50	276	120	970	965	167	111	107	175
100	276	120	963	964	211	101	99	219
200	276	120	965	966	316	95	95	325
400	276	120	980	982	529	120	120	416

Table A.15: Performance of Jacobi stencil on the GTX with tiles of 400xYx400 elements (the tile size in the table indicates the size of the tile on the Y-axis). The array has 400x400x400 elements; padding is 1 element on every side; blocks are 400x1x1 elements wide.

	Tesla				GTX			
	Normal		Contiguous		Normal		Contiguous	
Tile size	Basic	Scalable	Basic	Scalable	Basic	Scalable	Basic	Scalable
1	2020	2013	981	975	2016	2010	1108	1102
2	1460	1454	527	521	1520	1514	598	592
5	1127	1121	255	249	1222	1216	292	286
10	1021	1015	166	160	1123	1118	191	185
25	973	967	114	108	1064	1059	131	126
50	981	977	98	94	1044	1040	111	107
100	1033	1032	94	92	1034	1033	101	99
200	1150	1154	98	102	1029	1033	96	99
400	1390	1405	111	125	1026	1040	91	104

Table A.16: Predictions of Jacobi stencil with tiles of 400xYx400 elements (the tile size in the table indicates the size of the tile on the Y-axis). The array has 400x400x400 elements; padding is 1 element on every side; blocks are 400x1x1 elements wide.

			Normal			Contiguous		
Tile size	Sequential	Naive	Basic	Scalable	Buffered	Basic	Scalable	Buffered
1	280	110	1801	1795	208	157	150	208
2	280	110	1332	1323	170	114	108	162
5	280	110	1066	1064	151	89	82	134
10	280	110	1025	1018	143	83	76	126
25	280	110	999	993	149	80	74	132
50	280	110	994	988	170	80	76	153
100	280	110	988	988	217	81	79	199
200	280	110	990	995	336	87	91	313
400	280	110	1001	1018	576	109	124	443

Table A.17: Performance of Jacobi stencil on the Tesla with tiles of 400x400xZ elements (the tile size in the table indicates the size of the tile on the Z-axis). The array has 400x400x400 elements; padding is 1 element on every side; blocks are 400x1x1 elements wide.

			Normal			Contiguous		
Tile size	Sequential	Naive	Basic	Scalable	Buffered	Basic	Scalable	Buffered
1	276	120	1879	1874	199	177	171	200
2	276	120	1414	1408	161	133	127	157
5	276	120	1134	1129	152	107	101	129
10	276	120	1042	1037	141	98	93	122
25	276	120	985	981	147	93	88	127
50	276	120	968	965	166	92	88	147
100	276	120	964	963	209	91	89	193
200	276	120	965	966	313	90	90	293
400	276	120	981	982	527	120	120	409

Table A.18: Performance of Jacobi stencil on the GTX with tiles of 400x400xZ elements (the tile size in the table indicates the size of the tile on the Z-axis). The array has 400x400x400 elements; padding is 1 element on every side; blocks are 400x1x1 elements wide.

	Tesla				GTX			
	Normal		Contiguous		Normal		Contiguous	
Tile size	Basic	Scalable	Basic	Scalable	Basic	Scalable	Basic	Scalable
1	2020	2013	163	157	2016	2010	177	171
2	1460	1454	119	112	1520	1514	134	128
5	1127	1121	93	86	1222	1216	108	102
10	1021	1015	84	78	1123	1118	99	94
25	973	967	80	75	1064	1059	94	89
50	981	977	81	77	1044	1040	92	89
100	1033	1032	84	83	1034	1033	92	90
200	1150	1154	93	97	1029	1033	91	95
400	1390	1405	111	125	1026	1040	91	104

Table A.19: Predictions of Jacobi stencil with tiles of 400x400xZ elements (the tile size in the table indicates the size of the tile on the Z-axis). The array has 400x400x400 elements; padding is 1 element on every side; blocks are 400x1x1 elements wide.

			Normal			Contiguous		
Iterations	Sequential	Naive	Basic	Scalable	Buffered	Basic	Scalable	Buffered
1	280	110	1002	998	152	80	74	132
2	554	124	1003	999	152	80	75	131
3	846	138	1003	999	151	81	76	131
4	1120	152	1004	998	154	83	77	132
5	1395	166	1005	1002	155	88	82	134
6	1677	180	1005	1000	158	102	96	138
7	1951	194	1006	1001	162	116	110	145
8	2222	208	1006	1006	162	130	125	155
9	2503	222	1007	1002	180	145	138	169
10	2799	236	1007	1002	186	158	152	182
15	4150	306	1013	1006	257	228	223	253
20	5610	377	1013	1010	328	299	293	323
30	8309	517	1021	1017	470	440	434	464
40	11181	658	1063	1061	611	581	575	605
50	14020	799	1204	1204	754	722	716	746
75	21004	1152	1556	1559	1109	1075	1069	1099
100	27978	1504	1909	1914	1464	1427	1422	1451
150	41946	2209	2615	2624	2174	2133	2128	2158
200	55899	2915	3321	3334	2884	2839	2834	2863

Table A.20: Performance of kernel-iterated variant of Jacobi stencil on the Tesla. The array has 400x400x400 elements; tile size is 400x400x25 elements; padding is 1 element on every side; blocks are 400x1x1 elements wide.

			Normal			Contiguous		
Iterations	Sequential	Naive	Basic	Scalable	Buffered	Basic	Scalable	Buffered
1	276	120	984	979	148	93	88	127
2	553	143	985	980	149	95	89	127
3	830	166	987	982	153	97	92	129
4	1106	188	987	983	155	110	104	134
5	1383	210	990	985	159	132	127	153
6	1659	233	989	985	179	154	149	176
7	1935	255	991	986	202	177	172	199
8	2212	278	993	988	224	199	194	221
9	2489	300	993	989	247	222	217	244
10	2767	323	995	991	270	245	239	266
15	4151	436	1002	998	383	358	352	379
20	5532	549	1009	1005	496	471	466	493
30	8295	775	1161	1158	722	697	692	719
40	11069	1002	1387	1385	949	924	918	945
50	13839	1228	1614	1610	1176	1150	1145	1173
75	20763	1794	2180	2178	1743	1716	1711	1738
100	27679	2359	2746	2745	2310	2282	2277	2304
150	41455	3491	3878	3877	3443	3414	3409	3436
200	55347	4623	5010	5012	4577	4547	4541	4568

Table A.21: Performance of kernel-iterated variant of Jacobi stencil on the GTX. The array has 400x400x400 elements; tile size is 400x400x25 elements; padding is 1 element on every side; blocks are 400x1x1 elements wide.

			Normal			Contiguous		
Padding	Sequential	Naive	Basic	Scalable	Buffered	Basic	Scalable	Buffered
0	263	111	966	960	147	78	72	127
1	263	111	1001	996	158	80	74	134
2	263	111	1039	1034	162	82	77	137
3	263	111	1074	1070	172	84	79	140
4	263	111	1109	1105	179	87	81	148
5	263	111	1147	1141	179	90	84	151
6	263	111	1183	1179	187	93	88	162
7	263	111	1218	1214	202	97	91	159
8	263	111	1252	1244	198	100	95	168
9	263	111	1292	1293	205	103	97	170
10	263	111	1326	1322	213	106	101	175
11	263	111	1359	1354	224	109	104	178
12	263	111	1398	1393	228	113	108	183
13	263	111	1434	1430	233	115	110	187
14	263	111	1468	1464	250	119	114	192
15	263	111	1501	1498	248	122	117	196
16	263	111	1541	1538	244	125	120	201

Table A.22: Performance of a variant of Jacobi stencil on the Tesla, where padding size is variable (and indicates the number of elements that are padded on each side of each tile). The array has 400x400x400 elements; tile size is 400x400x25 elements; blocks are 10x10x10 elements wide.