GPU Compiler Optimizations
CS/ECE 570 – High-Performance Computer Architecture
Fuli Ma, Kit Morton, Austin Sharp, Chris Schultz
Objective
The objective of this survey is to discover and present high-level optimization techniques that are used by compilers for modern graphics processing unit (GPU) architectures. Due to the variety of GPU architectures across manufacturers and product lines, the majority of optimizations are made at higher levels, rather than operating on assembly code. It is also important to note that not all optimizations are equally effective on all architectures.

Why GPU programming?
The obvious use case for GPU programming is rendering graphics: this is the hardware’s main purpose. There can be up to thousands of threads running to produce colors for each pixel on a monitor. Replacing the pixel with any abstract value to be computed, however, GPU architectures can be used to solve many non-graphical problems. Parallelizable problems on a CPU can have 2-8 threads executing at a time. Take the same problem and run it on a GPU and execution may speed up immensely as hundreds to thousands of threads execute simultaneously. Harnessing this power is essential, as every modern computer, whether a desktop or smartphone, includes some form of GPU.

Why Compiler Optimizations?
Taking time to understand the target GPU architecture that a programmer will be using can take up a lot of valuable time. Compiler optimizations allow the programmer to write a correct algorithm without needing to know the hardware intimately. Since a piece of code may not be executed on the same architecture all the time, compiler optimizations can do the job of finding the exact configuration that is best for a given situation. Just as in CPU compilers, a GPU compiler can greatly ease the burden on programmers - even more so for programmers who are only familiar with programming for CPUs.

Background
A review of GPU architectures and compiler optimizations is necessary to motivate and create the context for GPU compiler optimizations. First, we recap some optimizations that are well known in CPU architectures. Next, we give an overview of the GPU architecture components, including threads, blocks, and the memory hierarchy.

Optimization Review
Some standard techniques that are used in CPU compilers are closely related to GPU optimizations. These will be very briefly reviewed to facilitate later discussion of the differences when making similar optimizations in a GPU environment.

Pipeline Scheduling
Assembly code as initially compiled may result in stalls from read-after-write or control hazards, when no forwarding path is available or two instructions need the same functional unit at the same time (e.g. too many simultaneous register writes). Pipeline scheduling is rearranging instructions to eliminate these problems. This is a class of optimization that is not as important in a GPU environment.

Loop Unrolling
One CPU compiler optimization that is very helpful in a GPU environment is loop unrolling: decreasing the number of times a loop runs, and simultaneously increasing the code executed within each iteration of the loop. The degree of loop unrolling possible depends on the number of registers and the size of
the instruction cache, as a larger loop takes more of both. Loop unrolling reduces the number of branch instructions and the associated overhead, as well as allowing more ILP that can be exploited by scheduling or other optimizations.

Memory-based Optimizations
This is the class of optimizations that has the most in common with GPUs in its purpose, although the differences in memory hierarchy mean that the specific optimizations used vary. For CPUs, most memory optimizations are intended to increase cache hits, either by prefetching data before it is needed or increasing spatial or temporal locality.

Spatial locality can be increased, for instance, by merging arrays: rather than declaring arrays A[sz] and B[sz], combine types A and B into struct C and create array C[sz], which is helpful if A and B’s values at the same indices are accessed together. Loop interchange and loop fusion seek to increase temporal locality: either by inverting nested loops to make data accesses sequential instead of random, or merging loops that share the same indices and data used. Finally, blocking is a technique that increases temporal locality by accessing and using entire ‘blocks’ of data at a time, such as reading an entire column or row in a matrix rather than a single element.

Kernel
A kernel in a General-Purpose Computing on Graphics Processing Units (GPGPU) program can be thought of as a function that is parallel in nature [7]. Ideally, every thread executing a given kernel will run the same instructions on different pieces of data. Examples of such kernels would be implementing matrix operations on individual cells, such as multiplication, addition, etc.

```c
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x; // handle the data at this index
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

*Figure 1: Example of a kernel. Implements a simple matrix addition.*

While it is ideal to have no branching, kernels can handle functions that can take multiple paths. The reason why branching is undesirable will be explained later in the paper.

GPU Architecture
There are three main components of a typical GPU architecture: threads, blocks and memory hierarchy.
Threads
A single execution unit on a GPU runs one thread, and is comprised of basic functional units (ALU, branching, etc). Compared to CPU threads, a GPU thread is much more lightweight. CPU threads are general purpose and run in operating systems, manage I/O interrupts, and context switch more frequently due to the limited number in a CPU. However, GPU threads only run a single kernel at a time, so there is no change in the instruction memory needed per thread. There is also no context switching or I/O interrupts - just pure computation.

Blocks
Groups of GPU threads are organized into blocks. Each thread in a block has a unique ID, and all threads within a block run the same kernel, so within a block all threads use the same instructions, but operate on different data. As a result, each block contains the instruction fetch hardware for all of the threads within the block. Different blocks can execute different kernels, however [7].

Memory Hierarchy
The memory subsystem of the CUDA GPU is separated into five different memory segments with different properties. Listed from fastest to slowest the GPU has the following types of memory [5]:

- Register File - pool of registers which are allocated among threads in a block.
- Shared Memory - Read/Write memory accessible by all threads within a block.
- Constant Memory - GPU read only memory for parallel access from all threads.
- Texture Memory - Constant memory optimized for vectorized operations.
- Local Memory - Overflow memory for register pressure.
- Global Memory - Main memory for GPU programs.

Register File
Each thread has a register file. All the threads in the block share registers within a pool. However, as long as a register is assigned to a thread, the register is only visible to this thread until it is reallocated to another thread within the block as needed. Due to the limitation of the size of the register pool, there may...
not be enough registers in a block for a task. This “register pressure” will result in the register file pushing its boundary into local memory (covered below).

**Shared Memory**

Shared memory is designed to allow thread-to-thread communication, because threads cannot share data through their registers. Like registers, shared memory is located on-chip. Each block has its own shared memory space, and the entire shared memory is visible to all threads within the block. Inside shared memory, there are a number of banks. The number of banks is equivalent to the maximum number of threads in a single block. After a bank is assigned to a thread, the thread will store its shared data into that bank, and that data is visible for all the threads with the block and ready to be read by all threads. However, if more than one thread wants to read the data from a bank, it has to be read in order, which means it can’t be finished in one cycle. Once a kernel completes, the data stored in the shared memory will be cleared. Another aspect of shared memory is that shared memory also plays the role of cache between threads and global memory, when it has extra space [5][7].

**Constant Memory**

The constant memory space is addressable by all threads executing on the GPU. It is located off-chip, elsewhere on the graphics card. It is read-only for threads on the GPU and is populated with constant values by the CPU prior to the execution of a kernel. This memory has been optimized for parallel memory access from many threads. For this reason it provides the largest performance increase when the same data is used by all threads within a block [5].

**Texture Memory**

Texture memory is also a read only space for the GPU, located off-chip [7]. When a thread requires adjacent data from texture memory, there is performance gain due to the memory being vectorized. In NVidia architectures, this is the only memory that has vectorized operations [11]. The unique design of this memory is due to its intended use in graphics: holding RGBA values for elements of textures, which would tend to be accessed in parallel as textures are mapped to objects in a graphical setting.

**Local Memory**

Local memory has the same function as the register files. This memory acts as an overflow protection for register pressure. Each block has its individual local memory space. Physically, local memory shares space with global memory, again off-chip. Since local memory is the slowest memory, GPU performance can drop significantly when registers are stored here [5].

**Global Memory**

Global memory is the main memory shared between the GPU threads and the CPU. This memory is used as the main data communication method between the host application and kernels running on the GPU. This memory, along with local memory, is the slowest because unlike registers and shared memory global memory is located off chip and does not have specialized memory access hardware available for constant or texture memory [5][7].
GPU Compiler Optimizations

Divergence

In the standard symmetric multiprocessing architecture, each core in a processor will have its own dedicated instruction fetch and decode hardware. This theoretically allows each parallel executing thread to fetch and decode a new instruction on every clock cycle (ignoring instruction cache misses). Because GPUs are designed to execute the same instructions on very large sets of data, this is not the case. Since all threads within a block execute the same kernel at any one time, all threads within a block share only one instruction fetch/decode unit. In the majority of cases this significantly reduces the die space required per thread without reducing processing throughput [12].

While this shared instruction fetch and decode implementation has no negative impact on a simple linearly executing kernel, if a kernel has two or more divergent execution paths the execution efficiency is reduced. When different threads within an execution block take divergent paths in a kernel, it is no longer possible for all threads to execute one instruction per cycle because unlike the non-divergent kernel case, threads are no longer executing the exact same instructions. Additionally, all threads within a block must finish executing before more threads can be scheduled onto that block. If one path through a kernel takes significantly longer than another path, the shorter path will complete first, and the efficiency of the block will be significantly reduced. There are three different optimizations which GPGPU compilers can use to eliminate or reduce the overhead caused by execution of divergent kernels [12].

Thread Regrouping

In a typical GPGPU kernel the data being processed by a particular thread is indexed by the thread IDs. For this reason, the data which is processed and thus the execution path taken in a kernel is in some ways dependent upon the identifier which are passed to the kernel. Because of this property, using the thread ID, a compiler can rearrange the threads into blocks where each thread will always take the same execution path [12].

At compile time, if the compiler is capable of predicting the most likely direction of a branch condition based upon the thread ID, it can rearrange the given thread identifiers so that all threads within a block will take the same path through the kernel instructions. This is generally implemented by creating virtual thread identifier mapping between the actual hardware thread ID and the identifier which is given to the kernel at run time [12].

Of course, determining these ID mappings requires some knowledge of both the kernel implementation and the data which will be processed. While this may be useful in some applications where the important data is known at compile time, it is not applicable for data which will change significantly at runtime. The other disadvantage of this technique is that threads within a block might fail to access memory in a sequential fashion. This can have detrimental effects on the ability of the GPU hardware's ability to coalesce memory accesses (covered later), and thus slow down memory operations [12].
Data Reorganization

An alternative method to thread regrouping is data reorganization. Where thread regrouping reorganizes the threads within a set of blocks, data reorganization moves the location of data so that, once again, all the threads will take the same execution path within one block. Because the thread identifiers are not being changed, if data accesses were initially coalesced this optimization will not prevent coalescing. However, if the data being accessed cannot be rearranged, or re-indexing the data would be an expensive operation, data reorganization can be less effective than thread regrouping [12]. An example of thread regrouping and data reorganization is shown in the following figure:

Warp Subdivision

Unlike thread regrouping or data reorganization, the goal of warp subdivision is not to eliminate the divergent paths within a block. The goal of warp subdivision is to make use of already wasted time on a block to execute the divergent path in a kernel. In many ways this is similar to the behavior of out of order execution on a superscalar processor. For example, if threads executing one path of a kernel request data from global memory, while this access is being executed, instructions from another execution path in the kernel can be fetched and executed. While the actual divergence between threads within a warp are not removed, the overhead caused by having to fetch different instructions for multiple threads is reduced [12].

Memory Optimizations

The next set of optimizations are memory oriented. As opposed to manipulating threads in a block to run the same instructions, memory optimizations organize data or accesses to it in such a way as to take full advantage of the hardware. There are five common techniques used to do this:

- Memory coalescing
- Vectorization
- Thread and thread-block merging
- Data prefetching
- Avoiding partition camping

To put it simply, memory optimizations strive to reduce the number and cost of data memory accesses required by the hardware.

**Memory Coalescing**

Memory coalescing is the act of grouping memory accesses that are indexed by a thread’s ID. Since global memory accesses can take hundreds of instruction cycles to execute, reducing the number of accesses is very beneficial.

Take as an example a block that has 32 threads with IDs from 0 to 31, and each thread wants to access a sequential piece of data. Assuming that the data is cache aligned (e.g. an array of 32 integers reside in one cache line of memory), the block can grab each of thread’s requests in one memory fetch. To achieve coalesced memory accesses, there are two requirements:

1. Sequential Access
2. Aligned Memory Access

Achieving these two characteristics in memory accesses can greatly reduce the number of global memory fetches, which is shown by example later in this paper [1]. The hardware is designed in such a way that when these requirements are met, all the memory access instructions for the threads in a block will be satisfied by one global memory access.

**Sequential and Aligned Memory Access**

Sequential and aligned memory access is where each thread within a block will use, in order by thread ID, a piece of data that is inside a cache line.

![Figure 5: Aligned and sequential data inside a cache line. This is an example of a coalesced memory read.](image)
**Non-Sequential and Aligned Memory Access**

Non-sequential, aligned memory access refers to the case where the data is all on one cache line, but the threads do not access it sequentially. More modern hardware can adapt to this issue, however sequential access is still preferred.

![Address and Thread ID Diagram](image1)

**Figure 6**: Aligned, but non-sequential, memory access. Notice the thread IDs do not directly map to a sequential order. With the data being aligned on a cache line, however, the data can still be pulled in by one global memory fetch.

**Sequential and Non-Aligned Memory Access**

The previous cases both have aligned memory accesses, which means it takes one global memory access to fulfill each thread’s request. When the data is split up into one or more cache lines, this will cause the block to have to do multiple global reads. This case is the most time consuming of the three, and makes optimization by memory coalescing impossible.

![Address and Thread ID Diagram](image2)

**Figure 7**: Non-aligned, sequential memory access. Notice that all but one thread is fulfilled by the green cache line, however the last one needs information from the red cache line. This will result into two global memory fetches by the block.
Example of Coalescing Memory

This section will provide a comparison of coalescable memory versus memory that cannot be coalesced. Consider the two forms of a matrix addition algorithm: algorithm (A) and algorithm (B). In this example, it will be shown that (A) is not coalescable, whereas (B) is [3].

```
__global__ void MatrixAdd(int *in1, int *in2, int *out)
{
    int t1 = in1[threadIdx.x * gridDim.x + blockIdx.x];
    int t2 = in2[threadIdx.x * gridDim.x + blockIdx.x];
    out[threadIdx.x * gridDim.x + blockIdx.x] = t1 + t2;
}
```

(a)

```
__global__ void MatrixAdd(int *in1, int *in2, int *out)
{
    int t1 = in1[blockIdx.x * blockDim.x + threadIdx.x];
    int t2 = in2[blockIdx.x * blockDim.x + threadIdx.x];
    out[blockIdx.x * blockDim.x + threadIdx.x] = t1 + t2;
}
```

(b)

Figure 8: Two implementations of matrix addition. Notice that algorithm (A) is using the block ID to index and the thread ID as an offset, where (B) is using the thread ID to index and the block ID to index.

Example Setup

In this example, assume each algorithm will be adding two 2x2 matrices, in which each matrix resides in global memory. Each row of the matrices are aligned in a cache line. Finally, the algorithms will be ran on two blocks, each containing two threads.
Non-Coalescing Example: Algorithm (A)
The way that (A) is accessing the memory is through a column oriented manner. In other words, block 0 is operating on the first column and block 1 is operating on the second. Since each row is on a cache line, this will split up the columns amongst two different cache lines.

```
#define MatrixAdd(int *in1, int *in2, int *out)
{
    int t1 = in1[threadIdx.x * gridDim.x + blockIdx.x];
    int t2 = in2[threadIdx.x * gridDim.x + blockIdx.x];
    out[threadIdx.x * gridDim.x + blockIdx.x] = t1 + t2;
}
```

Will take 4 total global memory accesses since data is split up amongst cache lines

![Figure 10: Example of algorithm (A).](image)

Notice that each thread is going to require the block to perform a global memory read, since one global memory fetch will not satisfy each thread. This will result in 4 total global memory accesses due to the non-aligned, non-sequential accessing.

Coalescing Example: Algorithm (B)
Algorithm (B) does the opposite of (A) to index the data. Rather than indexing by block ID, (B) indexes by thread ID and offsets by block ID. This makes each block operate at a row oriented manner (which happens to be how the data is laid out in memory).

```
#define MatrixAdd(int *in1, int *in2, int *out)
{
    int t1 = in1[blockIdx.x * blockDim.x + threadIdx.x];
    int t2 = in2[blockIdx.x * blockDim.x + threadIdx.x];
    out[blockIdx.x * blockDim.x + threadIdx.x] = t1 + t2;
}
```

Will take 2 total global memory accesses since data grabbed by blocks are aligned on the same cache line

![Figure 11: Example of algorithm (B).](image)
Notice that the accesses are both sequential and aligned inside the cache lines. Each block will only need to execute one global memory read in order to gather all the data to satisfy each thread. This results in two global memory reads, one per block.

Vectorization
The notion of vectorization is to segment data into groups and have each thread pull out that group of data [11]. An example of this can be seen as texture sampling. One color inside a texture has four values: red, green, blue, and alpha (RGBA). In a graphics application, typically each thread samples a texture (taking one pixel of color). Non-vectorized memory makes it so that the thread would have to do multiple reads to the texture, pulling each RGBA value separately. On the other hand, vectorized memory allows a thread to call one memory instruction and pull multiple pieces of consecutive data out (such as the RGBA). This reduces the overall number of memory accesses. The next sub-section introduces an example of vectorization.

Vectorization: An Example
In this example, a matrix-vector multiplication algorithm will be used. The naïve way of implementing this is looping through each value in a row of the matrix and multiplying the values together of the vector and adding it to a sum to form part of the resulting vector. The following figure shows this implementation in code [11].

```
#define A(y,x) A[(y)* width+(x)]
#define globalDimY I
__global__ void mv_naiive(float *A, float *B, float *C, int width) {
    float sum = 0;
    for (int i=0; i< width; i=i+1) {
        float a;
        float b;
        a = A[idx, i];
        b = B[i];
        sum += a*b;
    }
    C[idx] = sum;
}
```

*Figure 12: A naive implementation of matrix-vector multiplication. Notice that every loop requires a memory read.*

During a vectorization compiler pass, a compiler can notice that the matrix and vector are being only accessed and not written to. On top of that, it can also be concluded that each thread is going to be operating on consecutive pieces of data (in this case, a row of the matrix and the vector). Noticing these two things, storing data into a read-only, vectorized piece of memory would prove useful. In the case of
In this example, the memory would be texture memory. The following figure is the vectorized version of the algorithm [11].

```c
#define A(y,x) read_imagef(A, imageSampler, (int2)(x,y))
#define B(y,x) read_imagef(B, imageSampler, (int2)(x,y))
#define globalDimXY 1
__kernel void mv_vec(__read_only image2d_t A,
                    __read_only image2d_t B,
                    __global float* C, int width) {
    float sum = 0;
    for (int i=0; i< width/4; i+=4) {
        float a;
        float b;
        a = A(idx, i);
        b = B(0, i);
        sum += a.x*b.x;
        sum += a.y*b.y;
        sum += a.z*b.z;
        sum += a.w*b.w;
    }
    C[idx] = sum;
}
```

*Figure 13: The vectorized implementation of matrix-vector multiplication. Notice that the data are now located inside texture memory (which in this hardware, is the vectorized memory). This reduces the amount of memory operations by 1/4th.*

Vectorized memory combines data into groups of four (hardware dependent) as opposed to one. This allows the thread to pull in more than one piece of data with a single instruction, better known as Single Instruction Multiple Data (SIMD) instructions [7].

*Figure 14: Moving data into a vectorized memory. Notice how in block 0, thread 0 would need to pull in data 4 times in order to satisfy what it needs. Using vectorized memory allows thread 0 (and each thread) to pull in everything it needs in one instruction.*

Thread and Thread-block Merging

Although GPUs can execute many more threads in parallel than CPUs can, it is often the case that a workload can be over-parallelized, where there are far more threads than can be handled by the
GPU. For example, consider the case of graphics: most modern displays contain hundreds of thousands of pixels, while even high-end graphics cards can execute only a thousand threads simultaneously. When new threads are constantly being brought into the GPU, overhead costs are incurred, similar to context switching in a CPU. In particular, registers and shared memory may no longer be useful. Thread and thread-block merging are optimizations that aim to reduce this overhead.

Thread merging is exactly what it sounds like: multiple threads within a block are merged into one. The main idea is that this can facilitate register reuse - rather than bringing in a new thread that may or may not have the same data in registers, two threads that are sharing the same data can be combined into one. Thread-block merging is a very similar idea implemented on a greater scale: where two blocks are operating on the same memory, their groups of threads may be merged into a single block. This allows for reuse of the block’s shared memory, again reducing overhead and memory fetches [3][11].

While it may seem counterintuitive to decrease parallelism, thread and thread-block merging do so when the problem has far more parallel paths than the GPU can execute at once, and decrease the incidental costs and memory accesses associated with a greater number of threads or fewer threads per block. Of course, the exact degree to which this kind of merging is helpful depends entirely on the specifications of the hardware: the number of registers available, the size of shared memory, and so on [3][11].

Data Prefetching
This optimization is very similar to data prefetching in a CPU compiler. The main idea is to overlap memory I/O with computation, and doing memory accesses ahead of time to prevent threads from blocking on global memory, for example. This is particularly useful in cases where memory accesses are predictable - for instance, fetching the data needed in the next iteration of a loop. Since most parallelizable problems will result in a pattern of data accesses based on the threads involved, prediction is often a solvable problem. The ideal implementation of this technique does not bind prefetched data to registers or shared memory - the prefetched data does not interfere or compete with data currently in use, and thus can avoid hurting thread-level parallelism [11].

Partition Camping
As explained in the section on memory, GPU shared memory is designed explicitly for threads to access separate banks in parallel. Partition camping is when many threads are accessing shared memory at locations within the same block, so that the hardware’s parallelism cannot be exploited [11].

To avoid partition camping, two main techniques are employed. The first is to use address offsets. Data is actually spread out over more addresses than it would normally require so as to spread out the pattern of accesses - in other words, real data is padded with dummy data. The other technique is to remap block identifiers, again with the goal of distributing the addresses that are accessed [11].

Loop Unrolling
As yet another optimization common in CPU compilers, loop unrolling carries many of the same benefits as a GPU technique. First and foremost, it reduces the number of costly branches in the code (which also helps reduce divergence between threads). Second, it also leads to fewer instructions overall, and creates more opportunities to exploit instruction-level parallelism. As one would expect, it carries similar costs - execution time will be hurt if the loop size is greater than the instruction cache size, and more registers must be used as the loop is unrolled more [6].
This last issue is particularly problematic for GPU architectures, however. The number of registers per thread is not fixed, because logical registers can come from either the per-block pool or from the much slower local memory. Greater register usage and the resulting register pressure means that fewer threads can be executed per block, and since large numbers of threads are often context-switched to cover for memory I/O times, fewer threads means that memory accesses can cause more stalls [6][11].

As a result of these factors, it is more difficult to decide what degree of loop unrolling to implement in each case. One method [done by such and such authors] leverages the CUDA compiler’s optimizations and a disassembler to perform static analysis of optimized code and estimate its running time for a certain architecture.

<table>
<thead>
<tr>
<th>Unroll Factor</th>
<th>Register Usage</th>
<th>( O_{\text{max}} )</th>
<th>( W_{\text{full}} )</th>
<th>Estimated Stall Cycles</th>
<th>Measured Perf (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
<td>4</td>
<td>32</td>
<td>0</td>
<td>169.81</td>
</tr>
<tr>
<td>2</td>
<td>17</td>
<td>3</td>
<td>24</td>
<td>0</td>
<td>140.74</td>
</tr>
<tr>
<td>4</td>
<td>21</td>
<td>3</td>
<td>24</td>
<td>0</td>
<td>130.87</td>
</tr>
<tr>
<td>8</td>
<td>29</td>
<td>2</td>
<td>16</td>
<td>0</td>
<td>127.22</td>
</tr>
<tr>
<td>16</td>
<td>37</td>
<td>1</td>
<td>8</td>
<td>23808</td>
<td>154.09</td>
</tr>
<tr>
<td>32</td>
<td>50</td>
<td>1</td>
<td>8</td>
<td>27776</td>
<td>167.12</td>
</tr>
</tbody>
</table>

TABLE II
Stall cycle estimation for Monte-Carlo kernel

Figure 15: Effects of different degrees of loop unrolling

Their results showcase both the benefits of loop unrolling (reduced execution time) as well as the potential pitfalls (register usage, memory stalls). In particular, in the above example, as the number of registers used per thread increases, the number of threads in a warp and the number of warps cycled through each block both decrease. As a result, this eventually leads to memory stalls when there is no other warp to execute while waiting, and the overall execution time suffers.

Source-to-source Compilers

Source to source GPU compilers are a class of compilers which are used to convert portions of a program to more parallelizable versions of the same language or another language such as OpenCL or CUDA [2][9]. The purpose of source-to-source compilers is significantly different from that of the compiler optimizations discussed thus far. In the previous sections we have introduced compiler techniques which can be applied during the translation from CUDA or OpenCL to GPU assembly instructions. These optimizations are generally focused on tailoring a programmer’s hand parallelized CUDA or OpenCL implementation so that it makes the best use of specific hardware in a GPU. On the other hand, the goal of a source-to-source compiler is to optimize a generally implemented parallelizable algorithm so that it can be run on a GPU.

The main purpose of a source to source compiler is to simplify the implementation of GPGPU programs for the programmer. Without explicate knowledge of a compiler’s abilities it can be difficult for a programmer to code efficiently for a given platform. For example the Hybrid Multicore Parallel Programming language (HMPP) is a high level language for implementing parallel algorithms, however it
can only parallelize up to double nested loops. A source to source compiler can unroll these loops such that the HMPP compiler can most effectively parallelize the code [9]. Figure 16 shows different ways that the source-to-source compiler can do this.

(a) Transformation using 'contiguous' unroll, the codelet on the left shows the code with pragma, and the codelet on the right shows the resulting transformation.

(b) Transformation using 'split' unroll, the codelet on the left shows the code with pragma, and the codelet on the right shows the resulting transformation.

(c) Transformation using tiling, the codelet on the left shows the code with pragma, and the codelet on the right shows the resulting transformation.

Figure 16: Examples of different loop unrolling techniques through a HMPP to CUDA compiler

Another form of source to source compilers are designed to automatically parallelize a sequentially implemented program [2]. These tools allow the programmer to implement their code in a non-hardware specific way, then allow the source to source compiler to generate an optimized CUDA or OpenCL kernel which can be run on the GPU. While these compilers can be useful for simplifying the process of migrating a sequential program to execute on a GPU there is one main disadvantage: the kernel is automatically generated by the compiler. For this reason the effectiveness of the source to source compiler is dependent upon the code being parallelized. However, in some cases this technique can be very effective. The following figure shows a comparison between a source to source compiled Fast Fourier Transform (FFT) implementation and a hand coded CUDA FFT implementation. The runtime of the sequential CPU implementation (shown in blue) increases significantly with input data size. However the run time of same sequential algorithm run through a source to source compiler (shown in red) is comparable to the NVidia provided, hand coded CUFFT algorithm (shown in green) [2]. With very little work, a significant performance gain can be achieved over the sequential CPU implementation.

Figure 17: Comparison of multiple FFT implementations of different data sizes.
Conclusion

GPUs are not going anywhere. In fact, they are becoming more prevalent, as consumer computing devices are increasingly designed for multimedia and gaming performance rather than CPU compute power. In order to take advantage of this additional hardware and the massively parallel capability it offers, GPU compilers can implement a number of optimizations to speed up execution and memory accesses - some of them similar to what is found in CPU compilers, and some unique to GPU architectures.

All GPU optimizations are in some way a response to the strengths and limitations found in in the architecture. The existence of a single fetch unit per block leads to the problem of thread divergence, which is countered with thread regrouping, data reorganization and block subdivision. The unique memory hierarchy in GPUs leads to a set of memory optimizations: coalescing sequential and aligned accesses of threads in a block, vectorizing small sequential accesses into a single operation in texture memory, merging threads or blocks to increase shared register or shared-memory use and reduce overhead, prefetching data to reduce stalls for accesses, and introducing address offsets or remapping block IDs to avoid partition camping. Optimizations sometimes have more in common with CPU compiler optimizations, as in the case of loop unrolling, but even then additional techniques are necessary to determine the proper amount to unroll a loop.

Generally these kinds of optimizations are found in NVidia’s CUDA compiler or the like, but for those who haven’t learned a specialized GPGPU framework, source-to-source compilers offer the option to write code in a familiar language and still reap the benefit of the best frameworks and compilers. In most cases, source-to-source code performs similarly to hand-tuned CUDA code, and clearly outperforms CPU-only implementations.

Overall, GPU compiler optimizations are not revolutionary or completely alien to anyone familiar with the optimizations performed for CPU code. The differences can be traced directly to architecture designs, from the presence of many more threads to the graphics-focused memory hierarchy. Still, these optimizations are essential for efficient implementation of parallel algorithms on GPUs, from physics computations to cryptography, and will surely continue to be improved upon as compilers become more complex and GPUs are utilized ever more heavily.
References


