LAUNCH-TIME OPTIMIZATION OF OPENCL KERNELS

by

Andrew Siu Doug Lee

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science Graduate Department of Electrical and Computer Engineering University of Toronto

© Copyright 2017 by Andrew Siu Doug Lee
Abstract

Launch-time Optimization of OpenCL Kernels

Andrew Siu Doug Lee
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2017

OpenCL kernels are compiled first before kernel arguments and launch geometry are provided later at launch time. Although some of these values remain constant during execution, the compiler is unable to optimize for them since it has no access to them.

We propose and implement a novel approach that identifies such arguments, geometry, and optimizations at compile time using a series of annotations. At launch time the annotations, combined with the actual values of the arguments and geometry, are processed and used to optimize the code, thereby improving kernel performance.

We measure the execution times of 16 benchmarks our approach. The results show that annotation processing is fast and that kernel performance is improved by a factor of up to 2.06X and on average by 1.22X. When considering the entire compilation flow, improvement can be up to 1.13X, but can also potentially degrade to 0.44X, depending on the launch scenario.
Acknowledgements

My most humble thanks go to my advisor, Tarek S. Abdelrahman, for his invaluable expertise, candid advice, and generous patience. The document you are now reading, the research contained within, and the many related presentations are all the products of his guidance. I graduate as a better computer engineer and researcher because of his supervision.

My sincere thanks to my research group colleagues both past and present: Wilson Feng, Leslie Barron, David Han, Joe Garvey, and Alton Chiu. They provided much needed critical eyes, ears, and voices through innumerable meetings and practice presentations.

My grateful thanks to Qualcomm Canada for their financial support of my research efforts.

My parents, William and Susan Lee, and my partner, Cinderella Leong, are also deserving of my thanks and appreciation. It was their love and support that helped pull me through this process.
Contents

1 Introduction
   1.1 Research Overview ........................................... 2
   1.2 Thesis Contribution .......................................... 4
   1.3 Thesis Organization ......................................... 4

2 Background
   2.1 Graphics Processing Units (GPUs) .......................... 5
   2.2 Open Compute Language (OpenCL) ...................... 7
   2.3 Parallel Thread Execution (PTX) ....................... 9
   2.4 Caching ....................................................... 11
   2.5 LLVM .......................................................... 12
   2.6 Compiler Optimizations .................................... 13

3 Optimization Opportunity ............................................. 16
   3.1 Matrix Vector Multiply .................................... 17
   3.2 Stencil Computation ....................................... 19

4 Optimization Strategy .................................................... 22
   4.1 Definitions .................................................. 23
   4.1.1 Annotations .............................................. 24
   4.2 Algorithms .................................................. 25
4.2.1 Re-used Set ........................................ 25
4.2.2 Annotating Algorithm .............................. 27

4.3 Annotation Insertion .................................. 30
4.4 Annotation Processing ................................. 32
  4.4.1 Transformations .................................. 35

4.5 System Architecture ................................. 36
4.6 OpenCL shim ......................................... 37
  4.6.1 Caching ........................................ 38

4.7 LLVM Changes ....................................... 39

5 Evaluation ............................................. 40
  5.1 Compilation Flows .................................. 40
  5.2 Experimental Platform and Methodology .......... 42
  5.3 Benchmarks ......................................... 43
    5.3.1 Matrix Vector Multiply (MVM) ................ 43
    5.3.2 Stencil Computation (STN) .................... 44
    5.3.3 Rodinia ...................................... 45
  5.4 Kernel Results ..................................... 46
  5.5 Compilation Flow Results .......................... 47
  5.6 Multiple Kernel Launches ........................... 50
    5.6.1 Same Input Map ................................ 51
    5.6.2 Different Input Map ............................ 52
  5.7 Impact of Kernel Execution Time ................... 53
  5.8 Impact of Conservative Constant Propagation .... 54
  5.9 Summary .......................................... 56

6 Related Work .......................................... 58
  6.1 Constant Propagation ............................... 58
6.2 Optimizing GPU kernels ........................................... 59

7 Conclusions .......................................................... 61
  7.1 Future Work ....................................................... 62

Bibliography .......................................................... 63

A Annotations .......................................................... 69
  A.1 Removals .......................................................... 70
  A.2 Modifiers .......................................................... 72
  A.3 Transformations .................................................. 72
  A.4 Navigations ....................................................... 74

B Shim Library .......................................................... 76
  B.1 Shim Functions ................................................... 76
  B.2 Library Compilation Flags ...................................... 80
  B.3 Limitations ....................................................... 81
List of Tables

3.1 Effect of hand optimizing (in milliseconds) on MVM ..................... 18
3.2 Effect of unrolling (in milliseconds) on MVM .......................... 18
3.3 Effect of hand optimizing (in milliseconds) on STN ..................... 20
3.4 Effect of unrolling (in milliseconds) on STN .......................... 20
3.5 Count of PTX instructions by type for STN ............................. 21

4.1 List of the annotations .................................................. 25
4.2 PTX Strength Reduction Transformations ............................. 35

5.1 Description of MVM benchmarks ......................................... 44
5.2 Description of STN benchmarks ......................................... 44
5.3 List of Rodinia benchmarks used ....................................... 45
5.4 Description of Rodinia benchmarks used ............................. 45
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Simplified view of half of a GTX 780’s SMX layout, taken after [1]</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Visualization of a 2-D index space, taken after [2]</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Simplified OpenCL flowchart</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>addVecs OpenCL C version</td>
<td>10</td>
</tr>
<tr>
<td>2.5</td>
<td>addVecs PTX version</td>
<td>11</td>
</tr>
<tr>
<td>2.6</td>
<td>LLVM’s three phase structure, taken after [3]</td>
<td>12</td>
</tr>
<tr>
<td>2.7</td>
<td>Constant propagation example</td>
<td>13</td>
</tr>
<tr>
<td>2.8</td>
<td>Constant folding example</td>
<td>14</td>
</tr>
<tr>
<td>2.9</td>
<td>Loop unrolling by a factor of 4 example</td>
<td>14</td>
</tr>
<tr>
<td>2.10</td>
<td>Loop reversal example</td>
<td>15</td>
</tr>
<tr>
<td>2.11</td>
<td>Strength reduction example</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>MVM summation inner loop</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>SASS representation of MVM summation inner loop with \texttt{get_local_size(0)} as the loop bound</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>SASS representation of MVM summation inner loop with 256 as the loop bound</td>
<td>19</td>
</tr>
<tr>
<td>3.4</td>
<td>STN partial output computation</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>Flowchart of the proposed solution</td>
<td>23</td>
</tr>
<tr>
<td>4.2</td>
<td>PTX segment from MVM1</td>
<td>32</td>
</tr>
</tbody>
</table>
List of Algorithms

4.1 Re-used set algorithm ........................................................... 27
4.2 Input set algorithm ............................................................... 27
4.3 Annotation algorithm ........................................................... 29
Chapter 1

Introduction

Heterogeneous computing (the use of multiple processor types in a single system) is an increasingly popular approach to computing [4]. One common example is the use of central processing units (CPUs) with graphics processing units (GPUs). This combination, which uses the GPU to accelerate computations that would otherwise be performed on the CPU, is referred to as General-Purpose computing on GPUs (GPGPU). Due to the nature of their architectures, using GPUs in this manner provides the highest benefit for computations that are massively parallel. Because of this, GPGPU differs greatly from traditional computing in both its programming model and its program life cycle.

Open Computing Language (OpenCL) is an industry standard for programming heterogeneous systems [5], including ones that contain both CPUs and GPUs. A program written in OpenCL consists of two parts: a host program that executes on the CPU and a computational function, also known as a kernel, that executes on the GPU. The kernel is launched by the host program and its work is performed by multiple parallel work-items (i.e. threads). This kernel is normally compiled into a binary format only during its first execution and then retrieved from the cache for subsequent executions.

OpenCL provides an application programming interface (API) that the host program uses to compile a kernel, transfer data to and from the GPU and to launch the kernel for
execution on the GPU, providing it with its input arguments. Work-items are virtually organized into a multidimensional grid that defines an index space. We refer to the size and shape of this grid as the \textit{launch geometry}. An executing kernel can query the OpenCL runtime system for the launch geometry specific to its launch.

The kernel input arguments and the launch geometry are commonly used in the computations inside the kernel. For example, an input argument may be used as the upper bound of a loop in the kernel. Similarly, the number of work-items in a given dimension of the launch geometry is often used to calculate the section of the array to be cached into local memory.

We will refer to the kernel arguments, their values, and the launch geometry as the kernel’s \textit{input map} for brevity. Here, map refers to the data structure (also known as an associative array) that consists of keys, the names of the kernel arguments and launch geometry dimensions in this case, and values. They remain constant during the execution of the kernel. However, they cannot be treated as compile-time-known constants during kernel compilation. This is because with the OpenCL API, a kernel is compiled prior to its launch, and it is only at launch time that the input map values become known. This diminishes the ability of the compiler to apply important compiler optimizations to expressions that use input map values, including, constant propagation, constant folding, strength reduction and loop unrolling. Currently, this can only be done by compiling the kernel right before launch for every change in input map, which is by no means expeditious.

\section{Research Overview}

This thesis proposes a novel approach to tackling the aforementioned shortcoming. At compile time, the variables in the kernel that can be treated as constants are identified. The kernel binary code is then annotated to indicate how it would be optimized had
these variables been known as constants at compile time. By having the annotating done at the compilation stage, which is run once, we can minimize the added overhead at the launch stage, which is run every time. When the kernel is launched and the values of the input map are known, the annotations are processed and the kernel binary is re-written to apply any potential optimizations. If improvements in kernel performance resulting from these optimizations outweigh the overhead of processing the annotations, then overall performance benefits.

We implement our approach in an Nvidia specific GPU compilation tool chain that is built around LLVM [6]. In this tool chain, the binary representation of a kernel is the Parallel Thread Execution (PTX) virtual instruction set as opposed to the true binary executed by the GPU, known as Cubin. Thus, in addition to annotating the PTX code, we also use the PTX assembler, \texttt{ptxas}, to convert PTX into Cubin. The tool chain is used to compile 16 benchmarks on an Nvidia GTX 780 GPU. The performance of the benchmark kernels with and without our approach is compared. The results show that kernel runtimes can be improved by up to 2.06X and on average by 1.22X. We further quantify the overhead of processing the annotations and show that it is negligible, amounting to 0.87% of execution time.

Nonetheless, the benefit to kernel execution time when the entire compilation flow is taken into account (including annotation processing and PTX-to-Cubin compilation) depends on how often the kernel is launched and with what values in its input map. In the common case when the kernel is compiled once and then launched many times with the same input map, performance benefits to an average of 1.13X. In contrast, if the kernel is compiled once but is launched with different input map values, the additional PTX-to-Cubin compilation negatively impacts performance to an average of 0.44X. However, this overhead is a result of the specific tool chain we use and is not fundamental to our approach.
Chapter 1. Introduction

1.2 Thesis Contribution

The overall contribution of this thesis is the design, implementation, and experimental evaluation of a novel approach to optimizing an OpenCL kernel binary at launch time. This is done by recognizing that the values of its input map remain constant during execution, even when these values are not known as constants during compilation. More specifically, the contributions are:

- The design of a series of annotations for lightweight processing of OpenCL kernel binaries in conjunction with their input maps.
- The implementation of a system capable of automatically inserting said annotations at compile time and processing them at launch time.
- The experimental evaluation of said system using 16 benchmarks.

1.3 Thesis Organization

The remainder of this thesis further describes the background and motivation, the design of our proposed solution and its implementation, and the system’s performance and limitations over a suite of benchmarks. It is organized as follows: Chapter 2 provides the necessary background knowledge with Chapter 3 outlining the problem and the motivation for this thesis. Chapter 4 provides the details of our design through both a high-level overview and details of the implementation. Chapter 5 presents the results and analysis of evaluating our implementation. Finally, Chapter 6 discusses related works, while Chapter 7 presenting our concluding remarks and future direction.
Chapter 2

Background

2.1 Graphics Processing Units (GPUs)

Traditionally, GPUs consist of thousands of Shader Processors that allow a series of vertices to be transformed into the graphics that are ultimately displayed on a screen [7]. Today they exist in two main forms: integrated graphics (iGPUs) and dedicated graphics cards (dGPUs). iGPUs exist on the same die as the Central Processing Unit (CPU) and utilize the same physical memory (and depending on the iGPU, it may even use the same address space) [8]. dGPUs, as their name implies, are separate devices that connect to the CPU through Peripheral Component Interconnect Express (PCIe) and contain their own dedicated memory.

General-Purpose computing on GPUs (GPGPU) is the use of hundreds to thousands of GPU threads in parallel to offer high performance for parallel non-graphical computations. For the purposes of this thesis, an Nvidia GTX 780 dGPU is used as our reference architecture. The GTX 780 can provide such high performance through the use of 12 Streaming Multiprocessors (SMXs) which are each further divided into 192 single-precision Shader Processors (also referred to as CUDA cores by Nvidia) for a total of 2304 cores [9]. Each SMX also possesses 64 double-precision units, 32 special function
units (to provide support for intrinsic math functions like sine and square root), and 32 load/store units \[^1\]. Figure 2.1 shows a simplified layout of half of a GTX 780’s SMX.

Threads are organized into groups of 32 called warps with each SMX being able to launch 4 warps of 2 independent instructions each per cycle. Threads within each warp execute in a lock-step fashion following a single-instruction multiple-thread (SIMT) model. There is a maximum of 64 warps or 2048 threads scheduled per SMX at any given time.

![Image](image.png)

**Figure 2.1:** Simplified view of half of a GTX 780’s SMX layout, taken after \[^1\]

The GTX 780 is designed with a multi-level memory hierarchy \[^1\]. All threads have access to global memory, which is accessed at dynamic random-access memory (DRAM) speed. Each SMX has a user-managed fast cache, referred to as shared memory, which shares resources with a first level cache. Access to local memory is limited to concurrently running threads on the same SMX. Finally, there are registers and local memory which are accessible only by each thread.
2.2 Open Compute Language (OpenCL)

OpenCL is a framework for parallel applications, designed for heterogeneous systems, and that is platform independent \[5\]. One of its primary uses is in GPGPU computing. OpenCL kernels, the specifically designated parallel functions, can run on devices ranging from GPUs and CPUs to FPGAs. The framework consists of two main parts: a C99 language extension for writing kernels and an official C-language API for interacting with said kernels. Like other frameworks used for GPGPU, its primary use is to accelerate massively parallel computations by launching thousands of threads at once.

The terminology for OpenCL differs from that used for Nvidia devices (which is based upon Nvidia’s proprietary GPGPU framework called CUDA \[10\]). For OpenCL, local memory is the term equivalent to shared memory, while private memory is used in place of CUDA’s local memory term.

A typical OpenCL program will consist of dual-sources: a host program that runs on the CPU to call the API and a kernel function that runs on the target device - both are usually compiled just once, but separately from each other. The host program is compiled like any other C language program by using a compiler such as GCC and linking with an OpenCL library provided by various device vendors. The kernel is compiled at host program runtime by having the host program call upon the OpenCL API.

The threads that execute the kernel function are organized in a 3-dimensional index space, or grid. Each thread in this grid is referred to as a work-item in OpenCL terminology \[2\]. The sizes of this grid in the x, y, and z dimensions are referred to as the global sizes of the grid. OpenCL allows the programmer to optionally group work-items across three dimensions (x, y, and z) in work-groups which run on a single compute unit (the OpenCL term for an SMX). The sizes of a work-group in each dimension are referred to as the local sizes of the grid. Work-groups are traditionally uniform in size due to a requirement that the global size be a multiple of the local size. This organization of work-items and work-groups is collectively referred to as the launch geometry. A visualization
of a 2-D index space can be seen in Figure 2.2. The host program uses the OpenCL API to specify the launch geometry as well as the kernel input arguments when the kernel is launched for execution.

A kernel uses OpenCL intrinsic functions to determine the size and shape of the launch geometry. For example, a running kernel may use \texttt{get\_local\_size(0)} to find out the number of work-items in the \textit{x} dimension of the work-group. Similarly, it may use \texttt{get\_global\_size(1)} to determine the total number of work-items in the \textit{y} dimension of the index space. However, in OpenCL 2.0 and higher, the \texttt{get\_local\_size(dimindx)} function is not guaranteed to return the same value for calls within different work-groups \cite{11}. This can happen because the global size in a particular dimension is no longer required to be a multiple of the local size in the same dimension.

Figure 2.3 shows a simplified view of the main OpenCL phases. The compile phase involves the kernel being compiled to a binary format. Both the compiler and binary format are determined by the device’s vendor. This phase can happen offline (it can be done separately in advance) because the OpenCL API is designed to take in binary
formats as input to save on compiles. The advantage of this is that the kernel can subsequently be launched multiple times with new launch geometries and kernel arguments (the combination of which will be referred to as the input map) without having to be re-compiled due its generic nature (it has not been optimized for any particular input set).

The next two phases (launch and execution) supply the input map and execute the kernel respectively. It is important to observe that once a kernel is launched with a given input map these values are constant and do not change during the execution of the kernel.

### 2.3 Parallel Thread Execution (PTX)

When using an Nvidia device (where OpenCL 1.2 is currently the highest supported version) the OpenCL runtime calls a proprietary, closed-source compiler, NVCC, which compiles the kernel from OpenCL C to Parallel Thread Execution (PTX), a virtual ISA [12]. Using ptxas, the PTX assembler, this PTX code is then be assembled into an architecture specific binary format called Cubin for execution [13]. However, Nvidia has designated PTX, and not Cubin, as the binary format that the OpenCL API accepts for Nvidia devices. It is important to note that this means passing PTX into the OpenCL runtime requires assembling first before it can be executed; albeit, this happens faster than compiling the original OpenCL C kernel.

![Figure 2.3: Simplified OpenCL flowchart](image)
PTX is specifically designed to be independent of any particular Nvidia GPU architecture in order to maintain portability [12]. Its general format is as follows:

@p instruction.type d, a, b;

A typical PTX line consists of an instruction and its type followed by a destination operand (d) and then source operands (a and b). Conditional execution is indicated by an @ symbol in front of the instruction followed by a guard predicate operand (p). Registers are denoted with a % symbol in front to demonstrate that they are compiler-generated (e.g. %r5 for an integer register or %p3 for a predicate register).

```
__kernel void addVecs(__global int *A,
            __global int *B,
            __global int *C)
{
    int tid = get_global_id(0);
}
```

**Figure 2.4:** addVecs OpenCL C version

Figure 2.4 shows a simple but complete OpenCL kernel that adds vectors together. Here, each work-item will add together one index of vectors A and B, and then store the result in vector C. A complete PTX version of the same kernel can be found in Figure 2.5. From the figures, it can be seen that PTX does not use exactly the same concept of work-items and work-groups as OpenCL does. Instead it uses *Cooperative Thread Arrays (CTAs)*, which while similar to work-groups, are not strictly identical because they are the actual hardware implementation on Nvidia devices. This is evident when calculating the global work-item identifier (tid in the OpenCL C version) on lines 12-16. To construct this in PTX the CTA’s ID must be multiplied by the size of the CTA, and then the thread’s ID is added to that. Otherwise, the PTX version should be straightforward to understand from the OpenCL C version. Lines 17-21 load the operands, line 22 is the actual addition of index values, and lines 23-24 store the result.
2.4 Caching

To mitigate the overhead of invoking NVCC and ptxas every execution, Cubin files are automatically cached in a folder called ComputeCache. This prevents the need for OpenCL C kernels to be compiled more than once. The cached binaries are retrieved when presented with an OpenCL C or PTX kernel file that was previously compiled [10]. When upgrading ptxas versions the cache is cleared to allow for any new compilation improvements to be applied. The exact criteria used for determining if a file has been previously cached is not documented and its implementation is closed source.
2.5 LLVM

The LLVM Compiler Infrastructure Project is an open-source compiler project that provides a modular toolchain with support for many programming languages, assembly, and binary formats [6]. This support is possible through its general structure which consists of three phases:

1. a front end for parsing a specific language
2. a target-independent optimizer
3. a back end which performs code generation for a specific target

The second and third phases are collectively referred to as the LLVM Core libraries. An example of the many combinations of phases possible is shown in Figure 2.6.

A front end parses its supported languages, builds an abstract syntax tree, and converts that into LLVM intermediate representation (LLVM IR) [3]. It is LLVM IR that makes the LLVM project so flexible - all front ends emit it and all backends expect it as input. The optimizer operates upon LLVM IR in a series of passes that either analyze it, transform it, or do both. The exact passes which are run can be specified when launching the optimizer, and there exists the support to add one’s own written passes.
A back end will perform target-specific optimizations through its own set of passes, in addition to generating the final output. The LLVM IR will be lowered to a target-dependent instruction format, called MachineInstr \[14\], through a series of steps including instruction selection, instruction scheduling, and register allocation. At the end, a final output (either assembly or binary object code) will be emitted.

2.6 Compiler Optimizations

This section provides brief descriptions of several compiler optimizations that can potentially be enabled by considering the input map as constants. It is by no means exhaustive. Each optimization is accompanied by a basic example. For ease of presentation the examples are shown in C code as opposed to IR.

Constant Propagation

By recognizing that a variable holds a constant value at compile-time a compiler may propagate the value to any subsequent uses of the variable \[15\]. This optimization eliminates the need for the original variable. An example can be seen in Figure 2.7.

```c
int x = 5;
int z = y + x;
```

(a) Before

```c
int x = 5;
int z = y + 5
```

(b) After

**Figure 2.7:** Constant propagation example

Constant Folding

A compiler may calculate the value of a variable which is being assigned the output of an expression involving compile-time constants \[15\]. Once this has been performed, the compiler may subsequently perform Constant Propagation for that variable. An example can be seen in Figure 2.8. Special attention must be paid for constant folding floating
point operations when cross-compiling. This is because the precise implementation for
the operations may differ between architectures [16].

\[
\begin{align*}
\text{int } x &= 5 + 5; \quad \text{(a) Before} \\
\text{int } x &= 10; \quad \text{(b) After}
\end{align*}
\]

**Figure 2.8:** Constant folding example

**Loop Unrolling**

By duplicating a loop’s body using a given factor we also reduce the number of iterations
of the loop by the same factor [15]. This saves on the loop’s per-iteration overhead, which
consists of bounds checking, index adjustment, and branch misprediction penalties. As
seen in the example of Figure 2.9, this will accordingly increase the code size of the new
loop’s body. This optimization can only be performed if the loop trip count and bound
are known. However, this optimization may sometimes degrade performance depending
on several factors such as register pressure and instruction cache overflow [17].

\[
\begin{align*}
\text{for } (i = 0; i < 16; \quad \text{for } (i = 0; i < 16; ++i) \\
\quad \quad \text{sum} \quad \text{for } (i = 0; i < 16; \quad \text{sum} \quad \text{for } (i = 0; i < 16; \quad \text{for } (i = 0; i < 16; ++i) \\
\quad \quad \text{+= a[i];} \quad \text{sum} \quad \text{+= a[i+1];} \quad \text{sum} \quad \text{+= a[i+2];} \\
\quad \quad \text{sum} \quad \text{+= a[i+3];} \end{align*}
\]

\[
\begin{align*}
\text{(a) Before} & \quad \text{sum} \quad \text{+= a[i];} \\
\text{(b) After} & \quad \text{sum} \quad \text{+= a[i+1];} \\
\end{align*}
\]

**Figure 2.9:** Loop unrolling by a factor of 4 example

**Loop Reversal**

When the loop trip count is known and there are no loop carried dependencies, the loop’s
direction of iteration can be reversed as demonstrated in Figure 2.10 [16]. This can lead
to cache access optimizations as the data for the first few iterations may still be in the
cache from a previous loop [18]. The comparison is also now against zero, for which many assembly formats have a special "jump if zero" instruction that performs more efficiently compared to a generic comparison [19].

\[
\text{for (} i = 0; i < 16; ++i) \quad \text{for (} i = 15; i > 0; --i) \\
\text{sum += a[i];} \quad \text{sum += a[i];}
\]

(a) Before \hspace{1cm} (b) After

**Figure 2.10:** Loop reversal example

### Strength Reduction

Mathematical operations that meet certain criteria can be reduced to other operations that give the same result but are computationally cheaper to perform [20]. An example of this can be seen in Figure 2.11 where a division operation is reduced to a right bit-shift operation. This is possible here because the divisor is a power of 2 and thus meets the criteria for reducing the strength of the operation to that of a right bit-shift by the divisor’s base 2 logarithm.

\[
\text{int } z = y / 8; \hspace{1cm} \text{int } z = y >> 3;
\]

(a) Before \hspace{1cm} (b) After

**Figure 2.11:** Strength reduction example

It should be taken into account that performing an optimization may subsequently allow for more optimizations to be performed, such as constant folding enabling loop unrolling by making the loop bound a constant.
Chapter 3

Optimization Opportunity

As mentioned in Section 2.2, the kernel input map remains constant for the duration of a kernel’s execution. However, because they are provided only at launch time the kernel has already been compiled and optimized as best as it can be without knowing the values of the variables in the input map. Had the compiler known those values, it could have potentially performed the optimizations described in Section 2.6 on the kernel. Currently, the only way to optimize with those values would be to eliminate the compile phase and compile the kernel right before launch (i.e. never compile offline). Unfortunately, this compilation would then have to happen for every change in the input map, which is by no means quick. This would only compound the problem on Nvidia with its two-stage compilation, since caching the kernel would not have as much of a benefit anymore.

Although we have identified an overlooked opportunity to optimize the kernel, we must first determine whether there is any benefit to doing so. Additionally, we use this examination to reassure ourselves that these optimizations do not have any unintended slowdowns. To do this, we hand-optimize two OpenCL kernels and compare their resulting runtimes with their unaltered versions. The resulting improvements in runtimes give an upper bound on the performance improvements that can be achieved by our approach. The kernels we use are Matrix Vector Multiply (MVM) and Stencil Computation (STN).
They are further described in Section 5.3.

The kernels are hand-optimized for a given input map by: (1) replacing their input arguments with their corresponding constant values and (2) replacing calls to the OpenCL intrinsic functions that provide the sizes of the launch geometry by their resulting constant values. These constants are then hand-propagated throughout the kernel. In instances where this allows for constant folding, further constant propagation, or strength reduction we do so. We do not manually unroll loops whose bounds become constants. Instead we let ptxas perform this unrolling and any other optimizations enabled by our hand-optimizations.

The two kernels also serve to show that the performance benefits stem from different sources. We isolate the impact of propagating constants on loop unrolling by fixing the loop unrolling factors in the hand-optimized code to the factors in the original unoptimized code. We also measure runtimes when controlling for loop unrolling through the use of the unroll pragma in OpenCL and .nounroll directive in PTX.

### 3.1 Matrix Vector Multiply

In the case of MVM, the number of work-items in the x dimension is used as the upper bound of the summation loop. The first work-item in the work-group adds all partial dot products together and writes the result to global memory. The main loop of this kernel is shown in Figure 3.1.

```c
if (get_local_id(0) == 0) {
    float dotProduct = 0;
    for (uint t=0; t < get_local_size(0); ++t)
        dotProduct += partialDotProduct[t];
    W[y] = dotProduct;
}
```

**Figure 3.1:** MVM summation inner loop

As can be seen in Table 3.1, the time of the fixed-unroll factor version is higher than
Table 3.1: Effect of hand optimizing (in milliseconds) on MVM

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Original</th>
<th>Hand-optimized</th>
<th>Hand-optimized fixed-unroll</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVM</td>
<td>48.12</td>
<td>21.82</td>
<td>36.24</td>
</tr>
</tbody>
</table>

Table 3.2: Effect of unrolling (in milliseconds) on MVM

<table>
<thead>
<tr>
<th>Kernel</th>
<th>No unroll</th>
<th>Unroll allowed</th>
<th>Unroll factor</th>
<th>No unroll</th>
<th>Unroll allowed</th>
<th>Unroll factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVM</td>
<td>99.95</td>
<td>48.12</td>
<td>16</td>
<td>79.76</td>
<td>21.82</td>
<td>256</td>
</tr>
</tbody>
</table>

that of the hand-optimized version. This indicates that loop unrolling is a major source of the benefit. This is indeed the case; the loop shown in Figure 3.1 is unrolled by a factor of 256 in the optimized code compared to 16 for the original and the fixed-unroll cases.

However, there is also a speedup between the original and hand-optimized fixed-unroll factor versions. In other words, even when the effect of loop unrolling is held constant between the two versions, we still see a speedup from constant propagation. This can be better understood if the effect of loop unrolling is completely removed, as shown in Table 3.2. The time for the hand-optimized kernel rises to 79.76 milliseconds, but it is still faster than the original (at 99.95 milliseconds). This indicates that there is also benefit coming from non-loop unrolling sources.

Figure 3.2 shows the MVM summation inner loop from Figure 3.1 when disassembled from Cubin to SASS, a human readable format. Here we see R3, the loop counter, being checked that it is less than c[0x0][0x28], the local size in the x dimension stored in constant memory, for every loop iteration (line 7). R3 clearly counts up as it is incremented by 1 (line 4). The address of partialDotProduct[t] is calculated every iteration, with R4 storing both the address (line 3) and the value at that address (line 5).

This is in contrast to Figure 3.3 which shows the effect of loop reversal when the loop
3.2 Stencil Computation

STN performs a 21-point stencil computation on a 3D array in a diamond configuration. Here, there is a series of calculations for each output position using weighted input positions. Figure 3.4 shows an example of the calculations. In this example, the values
of B2_Z, B3_Y, and B3_X are a function of the launch geometry and can be determined by constant propagation and constant folding.

Table 3.3 shows that STN demonstrates a speedup when hand-optimized. In contrast to MVM, the performance of STN is unaffected by limiting the unroll factor. This can be seen in Table 3.4 where the unroll factor is 1 in both cases.

Table 3.3: Effect of hand optimizing (in milliseconds) on STN

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Original</th>
<th>Hand-optimized</th>
<th>Hand-optimized fixed-unroll</th>
</tr>
</thead>
<tbody>
<tr>
<td>STN</td>
<td>21.41</td>
<td>14.55</td>
<td>14.55</td>
</tr>
</tbody>
</table>

Table 3.4: Effect of unrolling (in milliseconds) on STN

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Original No unroll</th>
<th>Original Unroll allowed</th>
<th>Hand-optimized No unroll</th>
<th>Hand-optimized Unroll allowed</th>
<th>Hand-optimized Unroll factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>STN</td>
<td>21.41</td>
<td>21.41</td>
<td>14.55</td>
<td>14.55</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.5 shows the count of each arithmetic PTX instruction by kernel version. In contrast to the Original version, the Hand-optimized version has no rem or div instructions, and significantly fewer mad instructions. It also sees an increase in sub, add, shr, and mul instructions - all of which are computationally cheaper than the ones it lost [10]. This indicates that the benefit to performance stems from optimizations other than loop unrolling, such as constant folding and strength reduction.
## Chapter 3. Optimization Opportunity

<table>
<thead>
<tr>
<th>PTX instruction</th>
<th>sub</th>
<th>add</th>
<th>mad</th>
<th>rem</th>
<th>shl</th>
<th>shr</th>
<th>div</th>
<th>mul</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>6</td>
<td>110</td>
<td>30</td>
<td>14</td>
<td>34</td>
<td>0</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Hand-optimized</td>
<td>17</td>
<td>127</td>
<td>3</td>
<td>0</td>
<td>34</td>
<td>30</td>
<td>0</td>
<td>39</td>
</tr>
</tbody>
</table>

**Table 3.5:** Count of PTX instructions by type for STN
Chapter 4

Optimization Strategy

The strategy to automate the opportunity identified in the previous chapter is described first at a high level in this chapter, and then through its implementation details. We target OpenCL on Nvidia devices using a combination of compile-time annotations and launch-time binary rewriting. We refer to PTX as a binary for our purposes (see the discussion in Section 2.3).

Binary rewriting is the act of taking a functional binary and, without re-compiling, modifying it based upon some desired effect. The desired effect is communicated through annotations (hints in the form of metadata) in this strategy.

We leverage the fact that we know which kernel arguments and OpenCL work-item functions calls can be treated as constants across all threads during run-time to identify optimization opportunities. Using that knowledge, the compiler annotates the produced binary with annotations by way of an annotating algorithm, resulting in an annotated binary. At launch-time a processor parses the annotations and, in combination with the kernel arguments and launch geometry, carries out the optimizations. The resulting processed binary is unique for that particular input map. This process is depicted diagrammatically in Figure 4.1.

The act of processing a kernel renders it specific to a combination of kernel arguments
and launch geometry. Thus, the proposed solution sees two stages of caching involving both the annotated and processed binaries. The first level is a check if this particular kernel has ever been compiled before. If it has then we can rely on a cached version of our previously annotated binary. Otherwise, we need to compile it from the kernel file. If the first level check is true, we can then check if we have encountered this particular input map before to decide if we can use a cached version of our previously processed binary. If not, we need to process the annotated binary again to create a new processed binary.

### 4.1 Definitions

We define several sets and maps that are used during the annotation insertion and processing phases of our strategy. Although we use the term variable here, we are actually referring to registers, as will be explained in Section 4.2.1.

The input set \( IS \) is formally defined as the set containing kernel input arguments and
specific kernel variables that can be treated as constants during kernel execution. The
specific variables are the ones that hold the return values of any used OpenCL intrinsic
functions related to launch geometry. Thus, this set represents variables that become
constant at kernel launch time.

Similarly, the constant set (CS) is defined as the set of kernel variables that are
runtime constants during kernel execution. The constant set is initialized to the input
set and is grown as the annotation algorithm is run. Through constant folding and
constant propagation, new variables are identified for membership. This set is also built
and used during the annotation algorithm after the input set has been created.

The re-used set (RS) is defined to be the set of kernel variables that are written to
more than once within the kernel. This set is used to make our constant propagation
pass more efficient, as will be described in Section 4.2.1. It is built first before the other
two sets.

Finally, the constant map is defined as the constant set with the values of its variables
mapped to the constant values they take. Similarly, the input map (which was previously
defined in Section 2.2) comes from the input set. Clearly these maps are built at kernel
launch time, during the annotation processing phase, when the values of the input set
become known.

4.1.1 Annotations

Annotations are metadata that are attached to individual PTX instructions. They serve
to direct our launch-time processor in its manipulation of the PTX. In our case, they
contain information on the optimizations that can be performed on individual PTX
instructions. However, since they are only metadata, any software module that processes
the PTX is free to ignore them and leave the instructions unoptimized. This means that
the presence of annotations does not affect the correctness or the performance of a binary
if not processed.
Each annotation consists of a keyword and sometimes followed by an argument. This argument is either an integer or a string, depending on the annotation. Table 4.1 lists the annotations we have designed to carry out some of the optimizations described in Section 2.6:

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVRBL</td>
<td>Adds a register to the constant map with either the value of a parameter or an OpenCL intrinsic function</td>
</tr>
<tr>
<td>CFOLD</td>
<td>Adds a register to the constant map with its calculated value</td>
</tr>
<tr>
<td>CONDT</td>
<td>Removes predicated instructions that are ultimately not evaluated</td>
</tr>
<tr>
<td>CPROP</td>
<td>Replaces an input register with its value</td>
</tr>
<tr>
<td>TFSHR</td>
<td>Strength reduces a divide instruction if possible</td>
</tr>
<tr>
<td>TFSHL</td>
<td>Strength reduces a multiply instruction if possible</td>
</tr>
<tr>
<td>TFREM</td>
<td>Strength reduces a remainder instruction if possible</td>
</tr>
<tr>
<td>NEXT</td>
<td>Advances to the next annotation while outputting lines in between</td>
</tr>
<tr>
<td>SEEK</td>
<td>Checks if the current kernel is the one to be processed</td>
</tr>
<tr>
<td>SKIP</td>
<td>Advances to the next annotation without outputting lines in between</td>
</tr>
</tbody>
</table>

Table 4.1: List of the annotations.

The annotations listed in Table 4.1 are elaborated upon in Sections 4.3 and 4.4, where we describe how they are actually generated and processed. A full description of the annotations is provided in Appendix A.

4.2 Algorithms

This section describes the algorithms used in determining which instructions should be annotated.

4.2.1 Re-used Set

The re-used set plays a critical role in propagating constants through the kernel. Ideally, the system would track the definitions of a variable at a given use to know if it is a candidate for constant propagation. This information, known as a *use-define chain*, is
created by compilers for exactly that purpose \cite{10}. However, the stage at which the annotating algorithm would be performed may no longer have access to any analysis that was performed by the compiler in earlier stages. This could be due to the code having been transformed into either assembly or binary when the analysis was performed at the IR level. Re-running this analysis can be costly, both in terms of performance and implementation effort. Thus, we opt to make an approximate analysis that is simple to perform but may be over conservative. We simply disallow constant propagation for any variable that is defined more than once in the kernel.

A register that is written to more than once poses a danger to the integrity of the constant set if there is no access to its use-define chain. Since we are aiming for the smallest impact on compilation performance, we will simply create a set of all such re-used registers and use the set to make sure that it does not intersect with the constant set. This process is much more lightweight than building use-define chains for all re-used registers.

Algorithm 4.1 describes how the re-used set ($RS$) is populated. Initially, both $RS$ and $OS$, the output set, are empty. As we iterate over all the instructions, we add their output operands to $OS$. However, if $OS$ already contains that particular operand then it should be added to $RS$ instead since this means it is written to more than once. At the end, $RS$ will contain all output operands that are written to more than once.

Our conservative analysis for constant propagation is less of a problem for PTX specifically because it is a virtual ISA. As such, it uses virtual registers instead of being limited to hardware registers. An example of this is when the MVM kernel from Section 3.1 is assembled using ptxas. The register count falls from 16 floating point, 8 predicate, and 48 signed integer registers (for a total of 72) in the PTX file to 8 in the final Cubin. From a visual inspection of PTX files in general, there is very little re-use of registers, except when there are multiple definitions that could lead to the same use. Nevertheless, the impact of this is explored in Section 5.8.
Algorithm 4.1 Re-used set algorithm

**Input:** a function, $F$

**Output:** a re-used set, $RS$

$RS \leftarrow \emptyset$, $OS \leftarrow \emptyset$

for all instructions $I$ in $F$ do

let $O$ be the output operand of $I$

if $O \in OS$ then

$RS \leftarrow RS \cup \{O\}$

else

$OS \leftarrow OS \cup \{O\}$

end if

end for

return $RS$

4.2.2 Annotating Algorithm

There are two algorithms for generating the remaining sets described in Section 4.1 for use in annotating the binary. Algorithm 4.2 describes how the input set ($IS$) is created. It starts off by containing the launch geometry functions. Then, any integer type parameters are added to $IS$.

Algorithm 4.2 Input set algorithm

**Input:** a function, $F$

**Output:** an input set, $IS$

$IS \leftarrow \{launch\ geometry\ functions\}$

for all parameters $P$ in $F$ do

let $T$ be the data type of $P$

if $T = integer$ then

$RS \leftarrow IS \cup \{P\}$

end if

end for

return $IS$

Any kernel parameters that are passed in the form of pointers are automatically excluded. This is because the memory contents it points to may differ between invocations of the kernel and meaning it cannot be treated as a constant. Additionally, we do not constant fold floating-point operations due to the potential differences in the hardware implementation of said operations between the GPUs we are targeting and CPU on
which it will be calculated [10]. To simplify matters, we do not handle any floating-point constants at all. These limitations are not fundamental to our approach, and thus are candidates for future work.

Although the main algorithm (Algorithm 4.3) does create the constant set (CS), its main purpose is to create an annotated function from the original kernel function, IS, and RS. Its output AF, the annotated function, is an ordered multi-set that contains all of the original instructions and any applicable annotations. This means that order of entries in AF matters, but uniqueness does not. The algorithm starts by assigning CS the contents of IS.

The main body of the annotation algorithm iterates through all the instructions in a function. First it checks if the instruction has a predicate operand in CS - these will get the CONDT annotation. If the output operand is in RS but at least one input operand is in CS, then depending on the operation we can perform strength reduction through TFSHR, TFSHL, or TFREM. Other operations get CPROP instead. However, if the output is not in RS, then there is the potential to add the output operand to CS. Loading something from the input set gets CVRBL, while all other instructions with input operands in CS get CFOLD. Any other cases with at least one input from CS get assigned CPROP. Anything without an operand in CS or that does not meet any of the aforementioned conditions will remain un-annotated. NEXT and SEEK are heavily implementation dependent and as such will be described in greater detail in Section 4.3.

These annotations are inserted into AS, the annotated string. AS contains all the annotations for the instruction currently being annotated. At the end of the for-loop body AS and the instruction, I, are inserted into AF. The algorithm ends when all instructions have been iterated over and AF can be returned.
Algorithm 4.3 Annotation algorithm

\textbf{Input:} a function, $F$, an input set, $IS$, and a re-used set, $RS$

\textbf{Output:} an annotated function, $AF$

\begin{verbatim}
CS ← IS
AF ← ∅  //AF is the annotated function, an ordered multi-set
for all instructions $I$ in $F$ do
  $AS ← ∅$  //AS is the annotation string, a set
  let $O$ be the output operand of $I$
  let $NS$ be the set of input operands of $I$
  if $\exists N ∈ NS : N$ is a predicate, $N ∈ CS$ then
    $AS ← AS \cup \{\text{CONDT}\}$
  end if
  if output operand $O ∈ RS$ then
    if $NS \cap CS ≠ ∅$ then
      let $C$ be the operation type of $I$
      if $C = \text{Divide}$ then
        $AS ← AS \cup \{\text{TFSHR}\}$
      else if $C = \text{Multiply}$ then
        $AS ← AS \cup \{\text{TFSHL}\}$
      else if $C = \text{Remainder}$ then
        $AS ← AS \cup \{\text{TFREM}\}$
      else
        $AS ← AS \cup \{\text{CPROP}\}$
      end if
    end if
  else
    if $NS ⊆ CS$ then
      if $NS ⊆ IS$ and $|NS| = 1$ then
        $AS ← AS \cup \{\text{CVRBL}\}$
      else
        $AS ← AS \cup \{\text{CFOLD}\}$
      end if
      $CS ← CS \cup \{O\}$
    else
      $AS ← AS \cup \{\text{CPROP}\}$
    end if
  end if
$AF ← AF \cup AS \cup \{I\}$
end for
return $AF$
\end{verbatim}
4.3 Annotation Insertion

Although the previous section described how annotations are selected through an algorithm, this section presents further implementation details. The annotation insertion pass is implemented in the NVPTX backend of LLVM. The insertion process is similar to the one described in Section 4.2.2. After the required sets have been created, the kernel instructions are then iterated through. For a given instruction, if any of its input or predicate operands belong to the constant set, the instruction is marked for annotation. Specifically, one of the following annotations is inserted, depending on the instruction’s type and input:

- **CVRBL**: The instruction assigns the value of a parameter or work-item function to an output register that is not in the re-used set.
- **CFOLD**: All inputs of the instruction are in the constant set and the output register is not in the re-used set.
- **CONDT**: The instruction is predicated and the predicate register is in the constant set.
- **CPROP <int>**: Only some of the inputs of the instruction are in the constant set. The argument `<int>` is the input’s position in the instruction’s registers.
- **TFSHL**: The instruction is a multiply instruction and the register containing the multiplier is in the constant set.
- **TFSHR**: The instruction is a divide instruction and the register containing the divisor is in the constant set.
- **TFREM**: The instruction is a remainder instruction and the register containing the divisor is in the constant set.
The **NEXT**, **SKIP**, and **SEEK** annotations serve to speed up the navigation in the PTX file during annotation processing so that not every line of the PTX code need be parsed. Once the annotations for a line of PTX code have been determined, the last action for that line is to add a **NEXT** annotation to the previous annotation line. A string containing these annotations is created and is attached as metadata to the instruction. The final code-emission stage of NVPTX is modified to output the annotation metadata as comments into the resulting PTX.

The **NEXT** annotation serves to indicate how many lines of PTX code fall between the two annotation lines. This will be used later in processing to not have to process the lines in between. The lines passed over in this fashion are simply written out to the output file as they are un-annotated. Since the **NEXT** annotation is always inserted for the previous annotation line, the final annotation line will have no **NEXT** annotation and the processor will then come to a stop.

The annotation insertion process operates on a per-kernel basis with its final action being to add a **SEEK** annotation to the top of the kernel. This is because at that point the total number of lines in the kernel with addition of the annotation lines is known. As its annotation argument, the **SEEK** annotation takes the name of the kernel that immediately follows it. Another annotation, **SKIP**, is used to hold the number of lines and allow navigation across kernels. **SKIP** is used similarly to **NEXT**, except that the lines it indicates are ignored completely and not written to the resulting output. This is useful to avoid writing out kernels that are not the one being processed.

Figure 4.2a shows an example annotated PTX code section (i.e., the original un-annotated code plus the annotations inserted). The register **MVM_param_2** on line 2 is the only non-pointer parameter in this example. Thus, it is annotated with **CVRBL** as a member of the input set. Similarly, **ntid.x** on line 7 represents the local work-group size in the x dimension and is also added to the input set. **r14** and **r18** constitute the input set and form the foundation of the constant set. Next, **r18** is used on line 9, and since it is
the only known input operand, the instruction on this line is annotated with CPROP. The same action is taken for the instruction on line 18 for r14. In contrast, the two operands of the instruction on line 22 are an immediate value and a member of the constant set. Therefore, the instruction is annotated with CFOLD to calculate its result and p2 is added to the constant set. Finally, since the instruction on line 26 has a predicate guard with p2 as the predicate, it is annotated with CONDT.

```
1  // CVRBL NEXT 5
2  ld.param.u32 %r14, [param_2];
3  ld.param.u32 %r13, [param_1];
4  ld.param.u32 %r12, [param_0];
5  mov.u32 %r17, %ctaid.x;
6  // CVRBL NEXT 2
7  mov.u32 %r18, %ntid.x;
8  // CPROP 2 NEXT 4
9  mul.lo.s32 %r19, %r14 ;
10  mov.u32 %r20, %tid.x;
11  ...
17  // CPROP 3 NEXT 4
18  mul.lo.s32 %r21, %r1, %r14;
19  shl.b32 %r22, %r21, 2;
20  add.s32 %r2, %r12, %r22;
21  // CFOLD NEXT 4
22  setp.eq.s32 %p2, %r14, 0;
23  mov.f32 %f9, 0f00000000;
24  mov.f32 %f30, %f9;
25  // CONDT NEXT 4
26  @%p2 bra LBB0_10;
27  bra.uni LBB0_2;
```

(a) Annotated

```
1  ld.param.u32 %r13, [param_1];
2  ld.param.u32 %r12, [param_0];
3  mov.u32 %r17, %ctaid.x;
4  // CPROP 2 NEXT 4
5  mul.lo.s32 %r19, 256, %r17;
6  mov.u32 %r20, %tid.x;
7  ...
13  mul.lo.s32 %r21, %r1, 1000;
14  shl.b32 %r22, %r21, 2;
15  add.s32 %r2, %r12, %r22;
16  // CFOLD NEXT 4
17  mov.f32 %f9, 0f00000000;
18  mov.f32 %f30, %f9;
19  bra.uni LBB0_2;
```

(b) Processed

Figure 4.2: PTX segment from MVM1

### 4.4 Annotation Processing

The Processor takes annotated PTX and, in combination with the input map, produces processed PTX. It is essential that this processor be lightweight relative to the kernel
runtime as it may have to run frequently depending on the run scenario (see Section 5.6.2 for more information). The general idea is that as it processes each annotation, the constant map will grow the same way that the constant set grew as annotations were added during the annotation insertion phase. This means that other than the constant map, the processor should not need to retain any recollection of previous annotations that it has processed up to that point. It does this by parsing one line of annotations at a time while keeping track of register values in an incremental fashion as it processes annotations. Once processed, the PTX code will have had all of its annotations removed in addition to having any of their applicable effects carried out.

Annotation processing takes place once the kernel is launched. The input map is created by the OpenCL shim with the kernel input arguments and launch geometry values it intercepts before this point (see Appendix Section B.1 for the full details). The constant map is created using the input map as a foundation.

The processor then iterates over the inserted annotations. For this purpose, the NEXT, SKIP, and SEEK annotations are used (an OpenCL file can contain multiple kernels). Each annotation is processed until there are no more NEXT annotations, which indicates the end of annotations for that kernel. Each annotation instructs the processor to perform an action, as follows:

**CVRBL <int>** Instruction is removed and its output register is added to the constant map with the value of the parameter or work-item function. For parameters, the argument specifies which parameter it is (1 being the first, 2 being the second, etc.)

**CFOLD** Instruction is removed, its output value is calculated using the constant map, and the output register is added to the constant map with this value.

**CONDT** Using the predicate register value from the constant map, determine whether or not to remove the instruction.
Chapter 4. Optimization Strategy

**CPROP <int>** Instruction is rewritten with the specified register replaced by its value from the constant map.

**TFSHR** Perform CPROP on the second register; if the value is a power of 2 then change the instruction to a shift right.

**TFSHL** Perform CPROP on the second register; if the value is a power of 2 then change the instruction to a shift left.

**TFREM** Perform CPROP on the second register; if the value is a power of 2 then strength reduce as specified in Section 4.4.1.

**NEXT <int>** Advance the specified number of lines to the next line of annotations. Write the in between lines to the output file.

**SKIP <int>** Skip the specified number of lines to the next line of annotations. Ignore the in between lines.

**SEEK <str>** If the searched for kernel name matches the annotation argument, skip the rest of annotation line and start processing the next one.

Figure 4.2b shows the processed PTX code for the example shown in Figure 4.2a. In this example, it is assumed that the `param_2` argument of the kernel (matrix width) is 1000 and that `ntid.x` (work-group size in the x dimension) is 256.

The load and move instructions on lines 2 and 7 are removed due to their CVRBL annotations. Their respective values from the constant map are used as the values of the output registers `r14` and `r18`, respectively - the registers are also added to the constant map. For lines 9 and 17, the values of the second and third registers respectively are known, as indicated by the CPROP annotation. Thus, they are rewritten with those values replacing their respective registers. The CFOLD annotation causes the processor to compare `r14`’s value (1000) and 0, and puts the result of false as the value in the constant
map with \( p2 \) as the key. Finally, the CONDT annotation on line 25 makes the processor use that same value of \( p2 \) to determine that the branch instruction on the following line should be removed as it will not be evaluated.

### 4.4.1 Transformations

<table>
<thead>
<tr>
<th>General form</th>
<th>PTX form</th>
<th>Transformed PTX form</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q = n \times 2^k )</td>
<td>( \text{mul.l} \text{o.} \text{u32} %r3, %r2, %r1; )</td>
<td>( \text{shr.o} \text{b} \text{u32} %r3, %r2, k; )</td>
</tr>
<tr>
<td></td>
<td>( \text{mul.l} \text{o.} \text{s32} %r3, %r2, %r1; )</td>
<td>( \text{shr.o} \text{b} \text{u32} %r3, %r2, k; )</td>
</tr>
<tr>
<td>( q = n \div 2^k )</td>
<td>( \text{div.o32} %r3, %r2, %r1; )</td>
<td>( \text{sub.o32} %r3, 2^k, 1; )</td>
</tr>
<tr>
<td></td>
<td>( \text{div.s32} %r3, %r2, %r1; )</td>
<td>( \text{and.o32} %r3, %r3, -2^k; )</td>
</tr>
<tr>
<td>( q = n % 2^k )</td>
<td>( \text{rem.o32} %r3, %r2, %r1; )</td>
<td>( \text{sub.o32} %r3, 2^k, 1; )</td>
</tr>
<tr>
<td></td>
<td>( \text{rem.s32} %r3, %r2, %r1; )</td>
<td>( \text{and.o32} %r3, %r3, -2^k; )</td>
</tr>
</tbody>
</table>

Table 4.2: PTX Strength Reduction Transformations

Division and modulo operations are broken into several instructions and are computationally expensive compared to shifts on Nvidia GPUs [10] [21]. Integer multiplication and unsigned division by powers of 2 can be done through left and right bit shifting respectively using the power itself [20]. Unsigned remainder can done as show in in Table 4.2 through a combination of subtraction and a bitwise and operation. Signed integer division by a power of 2 is not as simple due to the possibility of a negative numerator [22]. The problem lays in the fact that bit shifting will truncate the decimal part of the number instead of rounding it towards 0 as most programming languages (including OpenCL C [2]) mandate. An example of this would be -3/2, which would be expected to produce -1 (since -1.5 is rounded towards 0). In 4-bit two’s complement -3 is represented
Chapter 4. Optimization Strategy

36

as 1101. When shifted right by 1 position this will result in 1110 which is -2. A solution to this problem is taken from [20] for both signed division and signed remainder, which can be found in Table 4.2.

4.5 System Architecture

A high-level view of the system that implements our approach is shown in Figure 4.3. It consists of a compile-time subsystem and a launch-time subsystem. The 3 main components that we have added are highlighted in grey - they are:

1. A compile-time annotation insertion pass

2. At launch-time annotation processor

3. An OpenCL Shim

The compile-time subsystem (shown in Figure 4.3a) extends the LLVM OpenCL compilation flow by adding an annotation insertion pass to the NVPTX backend of the LLVM compiler. NVPTX is a backend initially created by Nvidia, before being open sourced, to output PTX [23]. This pass analyzes the MachineInstr code and inserts the annotations. The compile-time subsystem also includes a shim library that intercepts calls to the OpenCL API and directs them to our system. It is used to call the modified LLVM flow instead of the OpenCL runtime to compile the kernel file and generate the annotated PTX.

The launch-time subsystem (shown in Figure 4.3b) mainly consists of the annotation processor, which modifies the PTX code based on the annotations that were inserted at compile-time. It too uses the shim library to facilitates the interception of OpenCL API calls in order to collect kernel arguments and launch geometry values for use by the annotation processor. In addition, the launch-time subsystem passes the processed PTX to the OpenCL runtime for the PTX-to-Cubin compilation.
Chapter 4. Optimization Strategy

4.6 OpenCL shim

The OpenCL shim is used to modify OpenCL host programs to use the system (as described in Section 4.5) without any major modifications. By acting as a layer between the host code and the OpenCL API library the shim allows us to call Clang & LLVM as our compiler instead of the vendor supplied one for example. It is composed of two main
parts: a header file for intercepting selected OpenCL API calls and a shim library that defines the API replacement functions. The header file is composed of `#define` macros that replace the specific OpenCL API calls with calls to their replacements from the shim library. This means that the replacement is done transparently once it is included, with the only other intervention being that the shim library must be linked against. The shim library can be compiled several different flags to determine the level of interception that the user wishes. This can range from not doing anything to printing debug messages with how long each API call took. Of course, its primary purpose is to allow the use of the system, including the modified Clang & LLVM compiler and processor, by modifying the OpenCL work flow. The full description of how the Shim modifies the OpenCL workflow and definitions of the select intercepted API calls can be found in Appendix B.

4.6.1 Caching

To reduce the compilation overheads and study the effects of running a kernel multiple times, we have implemented a basic version of the caching system described at the beginning of this chapter that works on the PTX level. This is in contrast to the system employed by Nvidia which works at a true binary level, as described in Section 2.4.

If the kernel has been compiled before then there will exist a cached annotated PTX file which can be used. Additionally, if this combination of kernel arguments and launch geometry has been used before then there will exist a cached processed PTX file which will then take priority over the annotated PTX file. When we refer to a kernel having the same input map, we are excluding pointer and float values from the kernel arguments. At the moment, these admissible kernel arguments are determined manually due to the way `clSetKernelArgs` is intercepted (see Appendix Section B.1). The impact of caching can be seen in Section 5.6.
4.7 LLVM Changes

Clang and LLVM can be used to generate PTX when the source OpenCL kernel is linked with libclc, a library that provides support for the use of OpenCL intrinsic functions \[24\]. We modified NVPTX in order to implement the Annotating Algorithm as described in Section 4.2 and this section. Additionally, we had to modify parts of the generic LLVM backend in order to support metadata at the MC Inst level, which is a format between MachineInstr and the emitted PTX (see Section 2.5). It is this addition of metadata support that allows us to propagate our annotations right until the code emission where we added code to output it alongside the original PTX line. This work was done on LLVM version 3.8.
Chapter 5

Evaluation

In this chapter, we present the results of evaluating our implementation. We first demonstrate the improvements attained by our approach to the runtimes of the kernels. We then report the benefit when taking the entire compilation flow time into account, i.e., including the overheads of annotation processing and the PTX-to-Cubin compile. We finally explore this benefit under different kernel execution scenarios.

5.1 Compilation Flows

We refer to the invocations and steps necessary to get from an OpenCL C source kernel to an executable binary as a compilation flow. There are three flows available and they are shown in Figure 5.1. The Vendor flow represents the most common way of compiling OpenCL on Nvidia systems. It invokes the proprietary NVCC compiler from Nvidia to compile from OpenCL C to Cubin in one API call, with PTX available for retrieval as the binary format. This flow is closed source and thus cannot be modified by us to implement our approach.

Figure 5.1b shows the LLVM flow that uses Clang & LLVM to compile from OpenCL to PTX and then relies on ptxas to assemble from PTX to Cubin. The generated LLVM IR is also linked with libclc, a library implementing the OpenCL intrinsic functions,
Figure 5.1: Compilation flows

(a) Vendor flow  (b) LLVM flow  (c) Modified LLVM flow
before being passed to the backend. We elect to use NVPTX [23] as the LLVM backend because it supports the emission of PTX. NVPTX is open-source and can be modified to implement our approach. However, one downside of using it is that the compile is now split into two steps, OpenCL compilation and PTX-to-Cubin compilation, possibly adding overhead compared to the Vendor flow.

We modify the LLVM flow as shown in Figure 5.1c to create the Modified LLVM flow (Modified flow for short). The backend is augmented to output annotated PTX, which is then run through the annotation processor at launch-time to generate the optimized processed PTX. As a result, the PTX-to-Cubin compilation is now shifted form compile-time to launch-time. One major disadvantage of this move is that each kernel within a file contains multiple kernels must be compiled separately from PTX to Cubin. The impact of this shift on performance is explored in our evaluation.

In all three flows, compilation time can be a significant contributor to runtime. Thus, the GPU’s device driver automatically caches a copy of the Cubin binary when an application is first compiled (as mentioned in Section 2.4). In the case of the Modified flow, a kernel’s binary code is optimized for a given input map, and thus can be different than another binary of the same kernel. Thus, we add another level of caching that keeps track not only of the kernel, but also of its input map, allowing us to avoid recompilation when the same kernel with the same input map is launched, as described earlier in Section 4.6.1.

5.2 Experimental Platform and Methodology

We report the execution times of several benchmarks compiled with the LLVM and Modified flows, described in the previous section. We also report the speedup of our Modified flow, which is defined as the execution time of a benchmark using the LLVM flow to its execution time using the Modified flow. Averages are computed as the geometric.
All results are collected on a system with an Nvidia GTX 780 GPU (described in Section 2.1). This GPU uses driver version 340.96 and supports OpenCL up to version 1.2. However, libclc only provides official support for OpenCL 1.1 [24]. The host system uses Ubuntu 14.04.4 LTS, and contains an Intel quad-core Q9550 CPU clocked at 2.83 GHz and 4 GB of memory.

5.3 Benchmarks

We use several benchmarks in our evaluation. They can be grouped into 3 categories. The first is a Matrix Vector Multiply program (MVM) from which 6 variants are derived. The variants represent different levels in which the program is optimized. The second category is a set of stencil applications (STN) and they are used to demonstrate the benefit of our strategy across different shapes and patterns of stencil computations. The final category of benchmarks is a subset of the Rodinia benchmarks and they are selected to represent different computational styles. We elaborate on these benchmarks below.

5.3.1 Matrix Vector Multiply (MVM)

The Matrix Vector Multiplication benchmarks come from Nvidia’s OpenCL SDK [25]. As their name implies, they each calculate the output of multiplying a matrix by a vector. The input problem is a matrix that is 1100 elements wide by 100000 elements high multiplied by a vector of size 1100.

The MVM benchmarks all have increasing levels of optimization. The MVM1 version is the naive unoptimized implementation that suffers from uncoalesced memory accesses and no use of local memory. Subsequent versions each add an optimization technique to the previous version, meaning that the 6 MVM variants represent increasingly optimized versions. Table 5.1 gives a description of each version. We opt to use such increasingly optimized versions to demonstrate that our Modified flow yields benefit irrespective of
Chapter 5. Evaluation

Table 5.1: Description of MVM benchmarks

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Variant</th>
<th>Lines of PTX Code</th>
<th>Number of Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVM1</td>
<td>1 row/work-item</td>
<td>149</td>
<td>20</td>
</tr>
<tr>
<td>MVM2</td>
<td>multiple rows/work-item</td>
<td>165</td>
<td>24</td>
</tr>
<tr>
<td>MVM3</td>
<td>multiple rows/work-item, shared memory</td>
<td>137</td>
<td>17</td>
</tr>
<tr>
<td>MVM4</td>
<td>multiple rows/work-item, shared memory, interleaved parallel reduction</td>
<td>144</td>
<td>18</td>
</tr>
<tr>
<td>MVM5</td>
<td>multiple rows/work-item, shared memory, sequential parallel reduction</td>
<td>138</td>
<td>16</td>
</tr>
<tr>
<td>MVM6</td>
<td>multiple rows/work-item, shared memory, sequential parallel reduction, warp-aligned</td>
<td>203</td>
<td>22</td>
</tr>
</tbody>
</table>

Table 5.2: Description of STN benchmarks

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Variant</th>
<th>Lines of PTX Code</th>
<th>Number of Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>STN1</td>
<td>Thumbtack with radius 4 and size 89</td>
<td>814</td>
<td>114</td>
</tr>
<tr>
<td>STN2</td>
<td>Dense with radius 3 and size 49</td>
<td>618</td>
<td>84</td>
</tr>
<tr>
<td>STN3</td>
<td>Star with radius 5 and size 21</td>
<td>707</td>
<td>124</td>
</tr>
<tr>
<td>STN4</td>
<td>Diamond with radius 3 and size 25</td>
<td>570</td>
<td>84</td>
</tr>
<tr>
<td>STN5</td>
<td>No Corners with radius 3 and size 45</td>
<td>610</td>
<td>84</td>
</tr>
</tbody>
</table>

how optimized the GPU kernel is. The MVM kernel used in Section 3 is repeated here as MVM3.

5.3.2 Stencil Computation (STN)

The Stencil Computation programs are synthetic stencil benchmark codes developed by Garvey et al. [26]. The STN kernels apply a stencil across a $256 \times 256 \times 256$ 3D input array to produce a similarly sized output array. However, they differ in the shape, radius, and size of the stencil itself, as indicated in Table 5.2. Here radius refers to the maximum distance between the edge of the stencil and its centre. The size refers to the number of input elements needed to produce a result for an output element. The STN kernel used in Section 3 is repeated here as STN3.
5.3.3 Rodinia

The 5 Rodinia benchmarks are selected from the overall Rodinia 3.1 benchmark suite [27] to each represent a unique memory access and computation style, also known as a dwarf [4]. Table 5.3 lists each selected Rodinia benchmark’s dwarf and application domain. They differ from the MVM and STN benchmarks because they are usually complete real world applications that consists of multiple kernels that may also be launched multiple times. Table 5.4 lists the kernels used in each selected Rodinia benchmark. It should be noted that some of the Rodinia benchmarks not used here are incompatible with the shim and thus are not candidates for inclusion (although through a non-trivial effort the shim could be modified to support them).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Abbreviation</th>
<th>Dwarf</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>HotSpot</td>
<td>HS</td>
<td>Structured Grid</td>
<td>Physics Simulation</td>
</tr>
<tr>
<td>Stream Cluster</td>
<td>SC</td>
<td>Dense Linear Algebra</td>
<td>Data Mining</td>
</tr>
<tr>
<td>Back Propagation</td>
<td>BP</td>
<td>Unstructured Grid</td>
<td>Pattern Recognition</td>
</tr>
<tr>
<td>Breadth-First Search</td>
<td>BFS</td>
<td>Graph Traversal</td>
<td>Graph Algorithms</td>
</tr>
<tr>
<td>LavaMD</td>
<td>LM</td>
<td>N-Body</td>
<td>Molecular Dynamics</td>
</tr>
</tbody>
</table>

Table 5.3: List of Rodinia benchmarks used

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Problem Size</th>
<th>Kernel Name</th>
<th>Lines of PTX Code</th>
<th>Number of Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>HS</td>
<td>4,096 x 4,096</td>
<td>hotspot</td>
<td>272</td>
<td>37</td>
</tr>
<tr>
<td>SC</td>
<td>4,096</td>
<td>memset_kernel</td>
<td>24</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pgain_kernel</td>
<td>295</td>
<td>50</td>
</tr>
<tr>
<td>BP</td>
<td>8,388,608</td>
<td>bpnn_layerforward_ocl</td>
<td>148</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bpnn_adjust_weights_ocl</td>
<td>82</td>
<td>4</td>
</tr>
<tr>
<td>BFS</td>
<td>4,194,304</td>
<td>BFS_1</td>
<td>107</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BFS_2</td>
<td>56</td>
<td>10</td>
</tr>
<tr>
<td>LM</td>
<td>100,000</td>
<td>kernel_gpu_opencl</td>
<td>400</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 5.4: Description of Rodinia benchmarks used
5.4 Kernel Results

Figure 5.2 shows the runtimes of the kernels when compiled using the all the three compilation flows. We include the Vendor flow to show that switching to the LLVM flow does not dramatically affect kernel runtimes. There are four main observations that can be made from the figure.

![Figure 5.2: Kernel runtimes](image)

The first is that for some kernels there can be significant performance improvement by using our Modified flow. For example, the Modified flow speeds up the execution of MVM3 by a factor of 2.06X, the execution of STN5 by a factor of 1.31X, and the execution of SC by a factor of 1.35X when compared to the LLVM flow. This demonstrates the benefit to kernel performance of our approach.

The second observation that can be made from the figure is that the use of our Modified flow benefits the performance of kernels with variants. For example, compared to the LLVM flow, the Modified flow speeds up both STN1 and STN2 by a factor of 1.16X and 1.29X despite their differences in shape. Similarly, the Modified flow speeds up both MVM1 and MVM2 by factors of 1.01X and 1.11X, respectively. These are both lower
than the speedup for MVM3 despite the fact that they are initially less optimized. This demonstrates that the benefit of our strategy exists for both kernels that are unoptimized and ones that are optimized to varying degrees.

The next observation that can be made is that for kernels that do not show improvement in performance, there is no or only negligible degradation in performance. There is almost no slowdown in kernel execution time using the Modified flow compared to the LLVM flow. The lowest speedup is 0.99X for BP. Slowdowns have been observed due to the uncontrolled nature of the loop unrolling. That is, our implementation does not manually unroll the loops, but instead relies on ptxas to do this task.

The final observation is that in most cases the Vendor and LLVM flows create kernels that are similar in runtime. The Vendor, LLVM, and Modified flows have geometric average runtimes of 23.08, 25.89, and 21.52 ms respectively. Although the Vendor geometric average is lower than that of the LLVM flow, the Modified one still lower showing the optimization potential through this technique. Additionally, as demonstrated in Chapter 3, there still exists the opportunity to optimize the Vendor flow.

Across all the kernels, the Modified flow achieves an average speedup of 1.31X, 1.26X, and 1.08X for the MVM, STN, and Rodinia benchmarks respectively - giving the overall average speedup as 1.22X. This validates our approach for improving kernel performance.

5.5 Compilation Flow Results

Since the input map is the collection of launch geometry and kernel arguments that can be treated as constants, there are three basic scenarios of how an OpenCL kernel can be launched:

1. A single time with a single input map
2. Multiple times, each with the same input map
3. Multiple times, each with a unique input map
For any compilation flow, the first scenario always necessitates the use of the entire system since this is the first time the kernel is processed. Because of the generic nature (i.e. input map independent) of its produced PTX, the Vendor and LLVM systems can cache this for the second and third scenarios while our system can only cache for the second. Since the processed PTX file is the result of processing the annotations in combination with its input map, the annotated kernel must always be re-processed for any changes in the input map. This means that for the third scenario there will always be the processing and PTX-to-Cubin costs.

![Figure 5.3: Compilation flow runtimes](image)

The execution times of the benchmarks for a single program run (i.e. scenario 1), including the entire compilation flow (as defined in Section 5.1), is shown in Figure 5.3. It is important to note that some of these programs have kernels that are run multiple times, but this is unavoidable due to the way the programs are structured. The displayed time is broken into OpenCL-to-PTX compilation (including our compile-time analysis), PTX-to-Cubin compilation, annotation processing, and kernel execution time. Each segment represents the sum of the respective times if there are multiple kernels or they are invoked multiple times. We no longer include the Vendor flow because its components
are opaque to us since it is closed-source.

SC is not represented on this chart (but is included in any presented averages) because of its extraordinarily long total processing and PTX-to-Cubin times that would skew the charts. This is as a result of its two kernels being invoked over 3000 times combined. Each kernel invocation suffers from the fact that it uses different kernel parameters and must therefore be re-processed and recompiled. SC has a total processing time of 1899 ms and a total PTX-to-Cubin time of 12212 ms. In contrast, BFS also has multiple kernels that are launched multiple times but does not require re-processing and recompilation due to its input map remaining the same.

The main observation that can be made from Figure 5.3 is that the OpenCL compile time dominates: it accounts for an average of 82.17% of total time in the LLVM flow. This drops slightly to 79.38% for the Modified flow due to the addition of the processing time and having to compile each kernel separately from PTX to Cubin. This OpenCL compile time percentage is not unexpected since compile time is normally longer than kernel execution time and the kernel may only be executed once. In contrast, kernel execution accounts for only 7.58% and 6.06% of the time on average. Since kernel execution is such a slow percent, the net impact is that on average the speedup of using the Modified flow compared to the LLVM flow is reduced to 0.95X (1.00X without SC) - meaning no significant speedup or slowdown. The speedup is 1.05X, 0.99X, and 0.78X (0.97X without SC) for the MVM, STN, and Rodinia benchmark groups respectively.

The annotation processing time amounts to no more than 0.87% of the execution time, making it barely visible in the chart for most benchmarks. This helps explain why the resulting speedup (0.95X) is so close to 1.00X. Nevertheless, for benchmarks that are large in terms of PTX lines (see Tables 5.1, 5.2, and 5.3), especially the STN benchmarks, the annotation processing time can vary up to an observed maximum of 3.32% of total measured time. The average processing time is equal to 14.42% of the average kernel execution time.
5.6 Multiple KernelLaunches

The above execution scenario of compiling a kernel and running it once is not typical. More often, the OpenCL source is compiled off-line to produce PTX code. Thus, benchmark execution time includes only the PTX-to-Cubin compile time. More importantly, once a kernel is compiled to Cubin, it is repeatedly executed multiple times. Thus, it is more realistic to examine execution time under such multiple invocation scenarios (scenarios 2 & 3 from the previous section).

When a kernel is run multiple times, the Cubin code of the kernel is cached to save compilation time, as was described earlier in Section 5.1. This is certainly the case for the Vendor and the LLVM flows. However, in the case of our Modified flow, the processing of the annotations at kernel launch time combines the annotations with the input map to produce the processed PTX code. This implies that the resulting Cubin can only be cached if it is launched with the same arguments and geometry. Otherwise the Cubin must be re-generated every time the kernel is launched with different arguments or with a different launch geometry.

Thus, we distinguish between two multiple kernel launch scenarios. The first launches the kernel multiple times with the same kernel arguments and launch geometry. Such a scenario occurs, for example, during video processing, during which the same kernel with the same arguments and launch geometry is launched for every frame of the video [28]. In this scenario, our Modified flow can cache the kernel’s code. The second scenario launches the kernel multiple times with different arguments and/or launch geometries. This scenario is typical of repeatedly executing a kernel, sweeping through input/geometry values to arrive at good performing configurations [29]. In this scenario, our Modified flow is unable to cache the kernel’s binary.

As in the previous section, we present the whole flow even if there are multiple kernel launches. Program launches in scenario 2 have the same problem to solve (e.g. the exact same matrix sizes for MVM). This means that even if there are multiple kernel launches
that have different input maps during one run of the program, there will still be a benefit from cached binaries across multiple program runs. Similarly, programs in scenario 3 have different problems to solve where possible, or have their processed binaries removed from their cache for the Modified flow. Again, SC and the Vendor flow are not shown in the charts for the same reasons discussed above (SC is still included in the geometric averages).

5.6.1 Same Input Map

Figure 5.4 shows kernel execution times including the entire compilation flow when the kernel is compiled once, but run 100 times as described for scenario 2. Again, the time is broken down as described in the previous section. The chart shows that the runtime of the kernels dominates, accounting for 89.17% and 86.59% of the total time on average. The OpenCL compilation, processing, and PTX-to-Cubin times are now all amortized across the runs and represent 11.33%, 0.12%, and 1.95% respectively for the Modified flow.

![Figure 5.4: Benchmark runtimes for multiple launches with the same parameters](image)

The use of our Modified flow gives a speedup that is consistently lower than that
of the kernel runtimes, but still improved over the LLVM flow, at 1.13X on average. Because of how the compilation flow time is divided, programs that showed a speedup when only the kernel execution time was considered show a speedup here for the same reason: the kernel executes faster. This can be seen visually through the similarities between Figures 5.4 and 5.2. The speedup is 1.27X, 1.22X, and 0.87X (1.01X without SC) for the MVM, STN, and Rodinia benchmark groups respectively.

### 5.6.2 Different Input Map

Figure 5.5 shows the kernel execution time for scenario 3, including the compilation flow when the program is run 100 times with different input maps. Instead of running with a new set of inputs for every run, we disable our caching in the Modified LLVM flow, to achieve the same effect of necessitating a PTX-to-Cubin compile every launch. The figure shows that while the initial compile is amortized across all the runs, neither the processing nor the PTX-to-Cubin compiles are. This drastically reduces performance. Although the annotation processing time is also incurred every launch, it is low enough to not impact performance.

![Figure 5.5: Benchmark runtimes for multiple launches with the different parameters](image-url)
The average speedup of our Modified flow is now reduced to 0.44X against the LLVM flow. However, it should be noted that the PTX-to-Cubin cost, which accounts for 63.88% of the Modified flow’s time in this scenario, is only a manifestation of our implementation. That time is completely avoidable if the Modified flow operated on Cubin, the true binary. This would be possible if, for example, the Vendor’s flow was open to modification. Under such a scenario with no PTX-to-Cubin component, the average speedup would be 0.96X, with a high of 1.22X for the MVM benchmarks due to their low processing times.

### 5.7 Impact of Kernel Execution Time

In this section, we investigate the impact of kernel execution time on the benefit derived from our technique. To evaluate this, we use input data size as a proxy for kernel execution time because it can be more easily controlled. We evaluate the variants of the MVM program because they have wide-ranging levels of speedup in the kernel execution resulting from our technique.

We evaluate at 4 different input data sizes by increasing the height of the input matrix from 100 to 100000 by factors of 10 while keeping the width fixed at 1100. This is the maximum height allowed by the GPU’s device memory capacity. The resulting kernel execution times scale near linearly in line with the increase in input data size. For example, MVM3 has average LLVM flow runtimes of 0.07, 0.48, 4.64, and 42.53 milliseconds as we increase the input data size. We see that the Modified flow runtimes for MVM3 scale similarly by about a factor of 10 at 0.04, 0.24, 2.36, and 21.64 milliseconds. This validates our choice in using input data size as a proxy for kernel execution time.

Figure 5.6 shows the speedup for the variants of MVM as a function of input matrix size. From the figure, it can be observed that the speedups hold and remain relatively flat as we increase the input data size, and thus the kernel execution time, for all variants. This indicates that our technique’s benefit exists irrespective of the kernel execution time.
Chapter 5. Evaluation

5.8 Impact of Conservative Constant Propagation

The annotating algorithm performs a conservative constant propagation through the re-used set, as described in Section 4.2.1. There is a concern that this conservative approach, which avoids full-blown analysis, may result in registers being assigned to the re-used set that would not necessarily have been had a more detailed analysis been performed. To assess this concern, we define the overlap set as the set of registers that were rejected from the constant set due to their membership in the re-used set. In other words, registers that are re-used but also meet the requirements of constant set membership.

Figures 5.7 and 5.8 show the sizes of the constant, re-used, and overlap sets for each kernel of each benchmark. The overlap set across the benchmarks is always quite small with a maximum of 4 elements (pgain_kernel), an average of 1.16 elements, and both a median and mode of 1 across all the kernels. Through manual inspection, it has been determined that in no cases was there an opportunity to constant propagate the elements of the overlap set. This is due to the fact that any instruction that uses an overlap set element is usually inside of a loop; meaning that the value of the element changes with

Figure 5.6: Speedup versus input matrix size for MVM
Figure 5.7: Size of annotation algorithm sets for the benchmarks with variants

Figure 5.8: Size of annotation algorithm sets for the kernels of the selected Rodinia benchmarks

each iteration and thus, is not constant.

Figure 5.9 demonstrates an example of such a scenario. On line 2 we see that r33 is being assigned a value of r24, a member of the constant set. Normally a simple mov operation with a known input would result in a CFOLD annotation and r33 being added
5.9 Summary

The overall results look good within reason due to the kernels not experiencing any significant slowdowns - the individual kernel speedup averages 1.22X with the lowest being 0.99X. When we consider the entire compilation flow, the added cost of the PTX-to-Cubin time decreases the average speedup to 0.94X. However, the annotation processing
accounts for only 0.87% of total time here - meaning it has very little negative impact on the speedup.

When multiple runs are considered, the system is able to amortize the added PTX-to-Cubin and annotation processing costs across those runs if the kernel arguments do not change. For such a scenario, the average speedup is 1.13X. If the kernel arguments change and were forced to re-process and recompile every time then the speedup drops to 0.44X. However, the PTX-to-Cubin compilation stage is only necessary since we do not operate on a true binary, as described in Section 2.3.
Chapter 6

Related Work

6.1 Constant Propagation

Compilers often propagate constants across functions calls; either through inline expansion or by data flow analysis \[30\]. However, applying this optimization requires that the values of propagated variables be compile-time-known constants. In contrast, OpenCL kernel inputs and launch geometry values are known only at launch time, preventing a compiler from propagating them as constants and applying subsequent optimization. Our work addresses this issue by efficiently performing the propagation and subsequent optimizations at runtime. The approach to runtime constant propagation and other optimizations presented in this thesis is novel, although runtime constant propagation has been used in other domains, such as specializing FPGA circuits \[31\].

There are several manual approaches to provide constant input values to the kernel at compile time. They include the use of macro definitions \[2\] and kernel specialization for common input values, either manually or using templates \[32, 33\]. However, these approaches suffer from several drawbacks. First, the kernel must be rewritten to use these techniques. Further, it must be recompiled for every new input map which is costly. Finally, these approaches work only when the input map is known a priori. This
is as opposed to our work which automates this process in a compiler with no changes to the kernel and for any input map, albeit with some launch time overhead.

OpenCL kernels can be optionally qualified with attributes that provide further information to the compiler [2]. Most relevant is the attribute `work_group_size_hint(x, y, z)`, which gives the compiler a hint as to what the local work-group size will be, but without any guarantees. Presumably, the hint allows the compiler to generate a specialized kernel version for the hinted launch geometry. There also exists `reqd_work_group_size(X, Y, Z)` which defines what the local work-group size must be or else queuing the kernel for launch will return an error. However, the use of these qualifiers on our experimental platform did not result in any performance improvements.

The use of such qualifiers is dependent on a vendor’s use of this information since the OpenCL specification does not mandate how to handle them. Even worse, since the attributes are provided at the OpenCL C level, any change would necessitate a new compilation as described in Section 2.2 and would eliminate any improvements in kernel performance.

### 6.2 Optimizing GPU kernels

There has been considerable work focused on optimizing GPU kernels, mostly to better exploit the memory hierarchy (e.g., [34, 35]) or to avoid thread divergence [36, 37]. In contrast, our work focuses on optimizing kernel code through propagation of constant kernel arguments and launch geometry when their values become known at launch time.

GPUCC [38] is an optimizing compiler for GPUs built around Clang & LLVM that also applies important optimizations such as strength reduction and loop unrolling. However, it is also limited by the lack of knowledge of the input map at compile time. Nonetheless, it can still specialize the code to use 32-bit operations instead of 64-bit ones. Checks are inserted at compile time to examine the values of operands and branch to an appropriate
version of the code. It important to note that GPUCC’s changes have been upstreamed
and is now a part of the NVPTX backend we use. Thus, our performance improvements
are on the top of any achieved by GPUCC.

GPU Ocelot is a framework that allows CUDA kernels to be emulated or run on Nvidia
or other platforms [39], and to provide analysis at the PTX level. It uses PTX rewriting
both for cross-platform execution and for profiling PTX through instrumentation [40].
Instead of a shim library that depends on the underlying OpenCL library as we have
used, GPU Ocelot completely replaces the CUDA Runtime API library with its own.
Although the framework itself does not optimize, it is extensible and allows for the
addition of optimization passes as demonstrated in [41].

pocl [42] is an open-source performance-portable implementation of OpenCL 1.2 based
upon Clang & LLVM for compilation. OpenCL is already platform portable, but across
a wide variety of platforms, each with distinct characteristics and benefiting from differ-
ent optimization strategies. pocl attempts to automatically optimize the kernel for the
user based upon which platform it is run on. At this point in time it is not usable in
combination with GPUs.
Chapter 7

Conclusions

This thesis describes the design, implementation, and evaluation of an approach for improving the performance of OpenCL GPU kernels. The optimization opportunity we exploit stems from the fact that some kernel input arguments and launch geometry remain constant during the execution of a kernel. Yet, the variables holding these values cannot be treated by the compiler as constants because they are supplied at kernel launch time, which happens after kernel compilation.

Our approach employs compiler analysis to annotate the PTX code of a kernel to reflect actions that would have been taken by the compiler had the aforementioned variables been compile-time-known constants. These annotations are then processed at kernel launch time, and, combined with the input map, are used to apply the optimizations. Although this approach introduces launch time overhead, the resulting improvements in kernel execution time yield overall benefits.

The evaluation of the approach, implemented in a modified LLVM compilation flow and using 16 benchmarks, shows that kernel execution time can improve by up to 2.06X and on average by 1.22X when compared to the original LLVM flow. The annotation processing time is minimal and has negligible impact. However, when taking the entire compilation flow into account, performance improvements are a function of how often
the kernel is launched. In one common scenario in which the same kernel is repeatedly launched with the same arguments and launch geometry, the benefits to kernel execution time are reflected in the compilation flow time. However, when the kernel is launched multiple times, each with a different launch geometry or input arguments, the need for a separate PTX-to-Cubin compilation eliminates kernel runtime benefits.

7.1 Future Work

There are multiple avenues that would provide worthwhile extensions to this work. One is to explore the effectiveness of the approach on newer devices. The GTX 780 used in this work has since been superseded by subsequent releases such as the GTX 980 and GTX 1080. Additionally, Nvidia manufactures a line of GPUs specifically for GPGPU known as the Tesla series. Alternatively, other platforms could be investigated such as AMD GPUs. This could be done through the use of LLVM’s AMDGPU backend [43]. Another extension is to support other compute APIs such as RenderScript [44] for Android devices or CUDA [45] for Nvidia devices.

A different avenue would be to consider fundamental changes to the system itself. The use of a shim library facilitated rapid prototyping. However, a more ambitious and definitive choice would be to modify an existing vendor OpenCL implementation. For GPUs, this would require modifications to the driver and, as of when this is being written, only the (similarly named, but different) AMDGPU driver [46] for latest generation AMD devices has been open-sourced. Finally, it is possible to support other compiler optimizations in the processor using additional annotations, such as loop unrolling or loop reversal.
Bibliography


Appendix A

Annotations

An annotation is a meta-instruction that direct the parser on how to manipulate the following PTX instruction (see Section 4.1.1). They are represented as comments in the PTX code, and each comment can consist of multiple annotations. Each annotation keyword is followed by a pair of parenthesis for arguments, even if they do not take any. The format of a line of annotations looks like the following:

\[\text{ANNOTATION}\text{ANNOTATION}\ldots\text{NEXT(ANNOTATION)}\]

They are generated by the modified LLVM implementation as described in Sections 4.3 and 4.2 and processed in order from left to right as described in Section 4.4. There are four categories of directives:

A.1 Removals Potentially removes the instruction

A.2 Modifiers Modifies the arguments of an instruction

A.3 Transformations Potentially modifies the instruction type

A.4 Navigations All other defined annotations
This appendix defines each annotation using the following format:

**Full name of the annotation**

Use What the annotation is used for

Conditions Under what conditions the annotation can be used

Mechanism What actions are taken upon processing the annotation

Notes Any further notes about the annotation

### A.1 Removals

**CVALU(ﬁnt)***

*Constant Value*

Use Forces an entry into the constant map with a specific value

Conditions The instruction is a parameter load

Mechanism The following PTX instruction is removed and the output register is added to the constant map as a key with its value being the value provided in the annotation argument.

Notes For debug purposes only and is not mentioned in the main sections of the thesis

**CVRBL()**

*Constant Variable*

Use Adds a register to the constant map with either the value of a parameter or an OpenCL intrinsic function.
Conditions The instruction is a parameter load or a move instruction involving an intrinsic register

Mechanism The following PTX instruction is removed and the output register is added to the constant map as a key. If there is an argument, then the value is that of a parameter and the annotation argument specifies which parameter it is (1 being the first, 2 being the second, etc.). Otherwise, the value is that of an OpenCL intrinsic function with the first input register specifying which function.

Notes None

**CFOLD()**

*Constant Fold*

Use Adds a register to the constant map with its calculated value

Conditions All the input registers are in the constant map

Mechanism The following PTX instruction is removed and the output register is added to the constant map as a key. Uses the instruction type in combination with the values of the input registers to calculates the value for the constant map entry.

Notes Currently only `bfe` (bit field extract) and `st` (store) instructions are considered unsupported for *CFOLD* and are demoted to *CPROP* instead

**CONDT()**

*Conditional Removal*

Use Removes predicated instructions that are ultimately not evaluated
Appendix A. Annotations

A.2 Modifiers

**CPROP(‹value›)**

*Constant Propagation*

**Use** Replaces an input register with its value

**Conditions** The input register specified by the annotation argument is in the constant map.

**Mechanism** Rewrites the following PTX instruction with the input register specified by the annotation argument replaced with its value from the constant map.

**Notes** None

**A.3 Transformations**

**TFSHR()**

*Transform Shift Right*

**Use** Strength reduces a divide instruction if possible

**Conditions** Instruction is a divide and the divisor is in the constant map
Appendix A. Annotations

Mechanism If the value of the divisor is a power of 2, the following PTX instruction is rewritten as a shift right (shr) with the divisor’s value replaced by its base-2 logarithm. Otherwise, it has the same effect as $CPROP(3)$.

Notes See Section 4.4.1 for implementation details

TFSHL()

Transform Shift Left

Use Strength reduces a multiply instruction if possible

Conditions Instruction is a multiplication and the multiplier is in the constant map

Mechanism If the value of the multiplier is a power of 2, the following PTX instruction is rewritten as a shift left (shl) with the multiplier’s value replaced by its base-2 logarithm. Otherwise, it has the same effect as $CPROP(3)$.

Notes See Section 4.4.1 for implementation details

TFREM()

Transform Remainder

Use Strength reduces a remainder instruction if possible

Conditions Instruction is a remainder and the divisor is in the constant map

Mechanism If the value of the divisor is a power of 2, the following PTX instruction is rewritten with the divisor’s value replaced by its base-2 logarithm. Otherwise, it has the same effect as $CPROP(3)$.

Notes See Section 4.4.1 for implementation details
A.4 Navigations

NEXT(⟨int⟩)

*Next Annotation Line*

*Use* Advances to the next annotation while outputting lines in between

*Conditions* None

*Mechanism* The processor advances a number of lines equal to the annotation argument while writing each line to output processed PTX file.

*Notes* None

SEEK(⟨str⟩)

*Seek Kernel*

*Use* Checks if the current kernel is the one to be processed

*Conditions* Must be placed at the beginning of a kernel

*Mechanism* If the kernel to be processed has the same name as the annotation argument, then immediately stop processing annotations on this line and advance to the next. If the names do not match, then continue processing annotations on this line. A *SEEK* annotation is followed by a *SKIP* annotation that indicates the number of lines that constitute the current kernel that can be skipped over until the next *SEEK* annotation.

*Notes* Can be followed by a *NEXT* annotation instead at the cost of reduced processor performance
Appendix A. Annotations

SKIP(⟨int⟩)

*Skip Unnecessary Lines*

**Use** Advances to the next annotation without outputting lines in between

**Conditions** Any lines between this annotation and the number of lines indicated by its annotation argument are not needed for the execution of the kernel being processed

**Mechanism** The processor advances a number of lines equal to the annotation argument, ignoring all lines in between.

**Notes** Not defined in Section 4.1.1 with the rest of the annotations
Appendix B

Shim Library

This appendix documents the OpenCL shim (first mentioned in Section 4.6) and consists of the following sections:

**B.1 Shim Functions**
Intercepted OpenCL API functions

**B.2 Library Compilation Flags**
Flags that modify the shim’s behaviour

**B.3 Limitations**
Any unsupported OpenCL API functions or use cases

Due to the way memory is handled in Nvidia’s implementation of the OpenCL standard (which may or may not be intentional), use of this shim in ways unforeseen to the author may cause a memory leak [47].

**B.1 Shim Functions**

The following `lok` functions will intercept their `cl` OpenCL 1.2 API counterparts and represent only a subset of the API. For example, `lokCreateContext` will replace all uses of `clCreateContext`. Other than the name, the shim functions have the same function signature, except for an optional final argument that is used to pass the line number on which it is called. The recommended way to use the shim is to include `lok_intercept.h`
and let the replacements happen during preprocessing. See [2] for more information on the OpenCL API.

Context APIs

lokCreateContext

Prints which compilation flow and caching flags have been used. Calls clCreateContext and stores the returned context, the number of devices, and a pointer to the device array.

lokCreateContextFromType

Prints which compilation flow and caching flags have been used. Calls clCreateContextFromType and stores the context. Calls clGetDeviceIDs and creates a dynamic array to store the device IDs.

lokReleaseContext

Frees the dynamic array of device IDs if lokCreateContextFromType was called earlier. Calls clReleaseContext.

Command Queue APIs

lokCreateCommandQueue

Sets the CL_QUEUE_PROFILING_ENABLE flag in the command queue properties and then calls clCreateCommandQueue.

Program Object APIs

lokCreateProgramWithSource

Begins by removing the ComputeCache if caching is disabled. For the LLVM and Modified flows, if caching is disabled or the annotated PTX file does not exist, it writes the
kernel out to a file called *source.cl* for later use. In the Vendor flow it simply calls *clCreateProgramWithSource*.

**lokReleaseProgram**

If the flow is Modified then it simply returns **CL_SUCCESS** as the *cl_program*’s lifespan is entirely contained within **lokEnqueueNDRangeKernel**. For the other flows, it calls **clReleaseProgram**.

**lokBuildProgram**

For the Vendor flow it calls **clBuildProgram**. For the other two flows, it will first check if there is already an annotated version of kernel. If there is not or caching is disabled then it will launch a series of calls to Clang and LLVM that will compile the OpenCL code to LLVM IR, link it with libclc [24], and pass it through the NVPTX backend generating the annotations. This resulting PTX file is outputted as *source.annotated.ptx*.

Additionally, if this is the LLVM flow then the annotated PTX is immediately compiled using **clCreateProgramWithBinary** so that a proper *cl_program* can replace the empty one provided previously by **lokCreateProgramWithSource**. This is possible because **lokBuildProgram**, unlike **clBuildProgram** has been modified to read in the *cl_program* object by reference.

**Kernel Object APIs**

**lokCreateKernel**

For the LLVM and Vendor flows it calls **clCreateKernel**.

For the Modified flow, we cannot yet create a kernel object because we have a proper *cl_program* object. Instead, we return a dummy kernel object which points to a dynamically allocated int. This unique address serves as the kernel’s identifier for later stages when it is just the kernel object and not the kernel’s name which is passed in. This
dummy kernel object is stored inside of a kernel struct which we use to store all relevant information about the kernel.

**lokReleaseKernel**

For the LLVM and Vendor flows it calls `clReleaseKernel`. 
For the Modified flow, we retrieve the kernel struct whose identifier matches the kernel object that is passed in. Then we delete the identifier and the kernel struct.

**lokSetKernelArg**

For the LLVM and Vendor flows it calls `clSetKernelArg`. 
For the Modified flow, we retrieve the kernel struct whose identifier matches the kernel object that is passed in. Then we store the kernel argument that was passed in. If this is the first time copying in the argument, or the argument is non-empty, not a memory location, and has changed values, then any previous processed PTX versions are nullified.

**Enqueued Command APIs**

**lokEnqueueNDRangeKernel**

For the LLVM and Vendor flows it calls `clEnqueueNDRangeKernel`. 
For the Modified flow, it first retrieves the kernel struct that has with a dummy kernel object that corresponds to the kernel being launched. This is necessary since we do not have access to the kernel’s name, only the kernel object. The input map is constructed from the values intercepted during calls to `lokSetKernelArg` in combination with the launch geometry values supplied to this function. If there is already a processed version of the PTX file, the kernel arguments have not changed in a way to nullify the previous processing, and caching is not disabled, then it loads the previously processed PTX file. Otherwise, it calls the processor in conjunction with the launch geometry and value of the kernel arguments to create a processed version that is written as `<kernelName>.processed.ptx`. 
Now `clCreateProgramWithBinary`, `clBuildProgram`, and `clCreateKernel` are called to create the proper kernel object. Finally, the kernel arguments are set for real using `clSetKernelArg` and the kernel is enqueued for launch.

In all flows, if the timing flag is set and no `cl_event` object is passed, then one is created.

## B.2 Library Compilation Flags

The following macro definitions can be supplied when compiling the shim library to change its behaviour as described below.

### B.2.1 Caching

- **LOK_DISABLE_CACHING**: Disable all caching (both annotated and processed PTX) and supersedes `LOK_DISABLE_CACHING_PROCESSED`. Can be used with any compilation flow.

- **LOK_DISABLE_CACHING_PROCESSED**: Disable only the use of cached processed PTX files. This is only applicable when used in conjunction with `LOK_FLOW_MODIFIED`.

### B.2.2 Timing

- **LOK_TIME**: Turns on automatic timing functionality and timing outputs. Kernels and most intercepted API calls will be timed.

### B.2.3 Verbosity

- **LOK_VERBOSE**: Turns on debugging outputs

- **LOK_DEBUG**: Deprecated. Turns on `LOK_VERBOSE`.

*Deprecate the LOK_DEBUG macro.*
B.2.4 Compilation Flow

These flags allow for the choice of which compilation flow to use (as described in Section 5.1). They are mutually exclusive with the default being LOK_FLOW_MODIFIED.

- LOK_FLOW_VENDOR Use Vendor compilation flow
- LOK_FLOW_LLVM Use LLVM compilation flow
- LOK_FLOW_MODIFIED Use Modified LLVM compilation flow

B.3 Limitations

The following non-exhaustive list documents OpenCL functions which are not intercepted at all. They may cause errors to be generated and/or return incorrect values if used in the LLVM or Modified compilation flows.

- clGetProgramInfo
- clGetProgramBuildInfo
- clGetKernelInfo
- clGetKernelWorkGroupInfo
- clGetKernelArgInfo

The following non-exhaustive list documents use cases which also not supported:

- multiple contexts
- multiple command queues
- multiple programs
- compilation flags not recognized by Clang
- multiple compilation flags