If you want to set off and go develop some grand new thing, you don't need millions of dollars of capitalization. You need enough pizza and Diet Coke to stick in your refrigerator, a cheap PC to work on and the dedication to go through with it.

John Carmack

University of Alberta

A compiler for parallel execution of numerical Python programs on graphics processing units

by

Rahul Garg

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

©Rahul Garg
Fall 2009
Edmonton, Alberta

Permission is hereby granted to the University of Alberta Libraries to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only. Where the thesis is converted to, or otherwise made available in digital form, the University of Alberta will advise potential users of the thesis of these terms.

The author reserves all other publication and other rights in association with the copyright in the thesis and, except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatsoever without the author's prior written permission.

Examining Committee

Jose Nelson Amaral, Computing Science

Paul Lu, Computing Science

Bruce Cockburn, Electrical and Computer Engineering

To the never ending advance of technology

Abstract

Modern Graphics Processing Units (GPUs) are providing breakthrough performance for numerical computing at the cost of increased programming complexity. Current programming models for GPUs require that the programmer manually manage the data transfer between CPU and GPU. This thesis proposes a simpler programming model and introduces a new compilation framework to enable Python applications containing numerical computations to be executed on GPUs and multi-core CPUs.

The new programming model minimally extends Python to include type and parallel-loop annotations. Our compiler framework then automatically identifies the data to be transferred between the main memory and the GPU for a particular class of affine array accesses. The compiler also automatically performs loop transformations to improve performance on GPUs.

For kernels with regular loop structure and simple memory access patterns, the GPU code generated by the compiler achieves significant performance improvement over multi-core CPU codes.

Acknowledgements

First and foremost, I would like to thank my supervisor Prof Amaral for his supervision throughout my studies as a Masters student. He helped me to maintain an overall direction in my thesis while still allowing me to freely explore new ideas. He not only taught me the skills necessary to conduct one's research, but he also taught me the importance of communicating ideas to the rest of the research community. This thesis would not have been possible without the patience and guidance of Prof. Amaral.

Next, I would like to thank Kit Barton, Calin Cascaval, George Almasi, Ettore Tiotto and many others from IBM who taught me the difference between toy compilers and real-world production compilers. In particular, Kit voluntarily mentored me and was always ready to answer my naive questions. The experience of working with a world-class group of compiler writers and researchers from IBM Research was very valuable and helped me grow as a researcher and compiler writer.

I would also like to thank Micah Villmoh, Michael Chu and many others from AMD's Stream computing group. They provided us with hardware for initial experiments and also provided us with information and clarifications about the CAL SDK. Their help is greatly appreciated.

The wonderful open source community behind NumPy provided me with useful feedback and suggestions. In particular, I would like to thank Travis Oliphant and developers of Cython for their feedback about syntax of extensions introduced.

I am also thankful to my commitee for providing feedback about the thesis. Their questions and comments have also prompted me to think in greater detail about possible future directions of this thesis.

My Masters studies were partially funded by an iCore international student award.

Finally, I would like to thank my friends Leslie Robinson and Leslie Weigl. Without their support and encouragement, I would not have survived the cold Edmonton winters and would have quit this project long ago.

Contents

1 Introduction **1**
1.1 Contributions . . . 2
1.2 Thesis Organization . . . 2

2 AMD RV770 architecture for GPGPU **4**
2.1 Overview . . . 5
2.1.1 SIMD units . . . 7
2.1.2 Texture units . . . 7
2.2 AMD CAL Programming model . . . 8
2.2.1 Memory management . . . 8
2.2.2 Context management . . . 8
2.2.3 Software execution Model . . . 10
2.3 Hardware execution model . . . 12
2.3.1 Pixel shaders . . . 12
2.3.2 Compute shaders . . . 12
2.4 Code generation for matrix multiplication . . . 13
2.5 On-chip bandwidth bottlenecks . . . 16
2.6 Final Remarks . . . 17

3 Python and the proposed programming model **18**
3.1 Python . . . 19
3.2 NumPy . . . 20
3.3 Python-C API . . . 22
3.4 Programmer's workflow . . . 23
3.5 Proposed extensions to Python . . . 24
3.5.1 Type annotations . . . 25
3.5.2 Parallel annotations . . . 25
3.5.3 Constraints on Python code to be compiled . . . 27
3.6 Conclusion . . . 27

4 Compiler implementation **28**
4.1 CPU code generation . . . 28
4.2 GPU code generation . . . 29
4.2.1 Motivation for the JIT compiler . . . 30
4.2.2 GPU code generation . . . 31
4.3 Final Remarks . . . 32

5 Array access analysis **33**
5.1 Representation and problem statement . . . 33
5.2 General solution . . . 38
5.3 More efficient solutions in specific cases . . . 40
5.3.1 Representation of union . . . 41
5.3.2 One-dimensional RCSLMADs with common stride . . . 42

5.3.3 Multidimensional RCSLMADs with common strides 44
5.4 Loop tiling for handling large loops 47
5.5 Conclusions 48

6 Loop transformations for AMD GPUs 49

7 Experimental evaluation 52
7.1 Matrix multiplication 53
7.2 CP benchmark 54
7.3 Black Scholes option pricing 56
7.4 5-point stencil 57
7.5 RPES benchmark 58
7.6 Remarks 59

8 Related Work 60
8.1 Compiling Python for CPUs 60
8.2 Array access analysis 61
8.3 Compilers for GPGPU 62
8.4 Conclusions 64

9 Conclusions 65

List of Tables

2.1 Comparision of bandwidth available to Radeon 4870 and Phenom II X4 940 . 17

7.1 Execution time for matrix multiplication benchmark for 32-bit floating point (seconds) . . . 54
7.2 Speedups for matrix multiplication using GPU for 32-bit floating point over ATLAS . . . 54
7.3 Execution time for matrix multiplication benchmark for 64-bit floating point (seconds) . . . 55
7.4 Speedups for matrix multiplication using GPU for 64-bit floating point over ATLAS . . . 55
7.5 Execution time for CP benchmark (seconds) . . . 56
7.6 Speedups for CP using GPU over OpenMP . . . 56
7.7 Execution time for Black-Scholes benchmark (seconds) . . . 57
7.8 Speedups for Black-Scholes benchmark using GPU over OpenMP . . . 57
7.9 Execution time for 5-point stencil benchmark (milliseconds) . . . 58

List of Figures

2.1 RV770 Block Diagram. Source : AMD. Printed with permission. 6
2.2 Memory arrangements 9
2.3 CAL programming model 11

3.1 Simple example of a Python function. 19
3.2 C pointer example 20
3.3 Examples of array slicing in Python. 22
3.4 Comparison between state of the art for generation of numerical-intensive applications for execution in CPU/GPU and the proposed programming model. 24
3.5 Example of decorator used as a type declarator. 25

4.1 Block diagram of unPython for CPU code generation 29
4.2 Block diagram of unPython for CPU code generation 32

5.1 Loop nests considered for array access analysis 34
5.2 Visualization of a two-dimensional RCSLMAD example 37

List of acronyms and symbols

ALU Arithmetic Logic Unit

AMD IL AMD Intermediate Language

AOT Ahead-of-time

API Application Programming Interface

AST Abstract Syntax Tree

ATLAS Automatically Tuned Linear Algebra Software

CAL Compute Abstraction Layer

CUDA Compute Unified Device Architecture

ERL Extended Restricted Constant Stride Linear Memory Access Descriptor

FLOP Floating Point Operation

GDDR Graphics Double Data Rate

GFLOP Giga Floating Point Operation

GLSL OpenGL Shading Language

GPU Graphics Processing Unit

GPGPU General Purpose Graphics Processing Unit

HLSL High Level Shader Language

JIT Just-in-time

LDS Local Data Share

LMAD Linear Memory Access Descriptor

PCIe Peripheral Component Interconnect Express

RCSLMAD Resctricted Constant Stride Linear Memory Access Descriptor

RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
SIMD	Single Instruction Multiple Data
SP	Stream Processor
SPMD	Single Program Multiple Data
SSE	Streaming SIMD Extensions
TEX	Texture Unit
TP	Thread Processor
UVD	Universal Video Decode
VLIW	Very Large Instruction Word
b	Base of an LMAD.
D	Domain of an RCSLMAD.
E	Symbol for an ERL
L	Symbol for an LMAD.
p_k	Stride of an RCSLMAD in the k-th dimension
u_k	Upper bound of k-th loop.

Chapter 1

Introduction

In recent years, processor designers have hit a wall in increasing the performance of a single processor core. Processors have adopted multicore designs fitting multiple cores on a single die or a single package. Quad-core CPUs are already common on the desktop and hexa-core processors are available for servers. Another new development in the hardware field has been the emergence of programmable graphics processing units (GPUs). GPUs have evolved from a fixed function pipeline to a highly programmable pipeline. Modern GPUs are a massively parallel architecture containing hundreds of programmable ALUs that can be utilized for tasks such as numerical computation. A programmable GPU which can be utilized for non-graphic related tasks is termed a general-purpose GPU (GPGPU). GPGPUs are less flexible than CPUs but offer much greater floating-point capability. GPGPUs offering up to a teraflop of floating-point performance are already available. This performance is an order of magnitude better than current x86 processors.

The performance gains offered by multicores and GPGPUs comes at the cost of increased programmning effort. To utilize multicores, a programmer is required to write parallel code. Many abstractions, such as threads and parallel loops, are available for writing parallel programs. For numerical programs, expressing parallelism through parallel for-loops (available in APIs such as OpenMP) is a common choice for multicores as well as for symmetric multiprocessor architectures. Programming for GPGPUs requires the programmer to rewrite much of her code to utilize a GPU specific API. GPGPUs usually have a completely separate address space and therefore the programmer has to copy data from the system RAM to the GPGPU address space.

On the software side, languages such as Matlab and Python are becoming more popular for scientific computing. Python in particular is starting to attract a lot of attention in the numerical-programming community. Python is a general-purpose object-oriented language with very clean syntax. An extension library for Python, called NumPy, offers very flexible multi-dimensional array abstractions and is particularly useful for scientific programming. However, while Python offers great flexibility to the programmer, performance is usually

much lower than equivalent C programs. Python also does not currently provide any simple tools for parallel programming for numerical programs.

This thesis presents a compiler for Python that enables the programmer to easily utilize multicore processors and GPUs for maximum performance. The programmer only needs to add minimal annotation to the program and the compiler can automatically generate multicore CPU and GPU code. The annotations are designed to be portable to systems with or without a GPU.

1.1 Contributions

The thesis presents a compiler system that implements the following:

1. A simple type-annotation system and a simple parallel `for` loop API for Python. The parallel `for` loop exposes a data-parallel shared-memory paradigm to the programmer.

2. A new array-access analysis algorithm that can automatically compute the memory addresses to be transferred between the CPU and the GPU for a class of parallel loops. The algorithm handles rectangular loop nests with loop-invariant bounds and with a particular class of affine subscript expressions for array references.

3. A set of loop optimizations for the AMD Radeon 4800 series of GPUs.

The compiler is implemented as a three-part system:

1. An ahead-of-time compiler, unPython, that compiles a subset of annotated Python to C++ and OpenMP.

2. A just-in-time compiler, jit4GPU, that generates optimized GPU code from a tree representation of the Python program. Jit4GPU also computes the memory addresses to be transferred between the CPU and the GPU as required by the computation.

3. A run-time system that supports jit4GPU by providing jit4GPU with a simple API to manage program execution on the GPU as well as data transfer between the CPU and GPU.

Experimental results, discussed in Chapter 7, show that using a GPU resulted in up to 100 times performance improvement over compiler-generated parallel C++ with OpenMP programs.

1.2 Thesis Organization

Chapter 2 discusses the GPGPU architecture used in this thesis, codenamed the AMD RV770. All experiments in this thesis were done using the GPU AMD Radeon 4870 based

upon the RV770 architecture. A brief description of Python, NumPy and the new annotations proposed is given in Chapter 3. Chapter 4 gives an overview of the compiler implementation. Chapter 5 describes the array-access analysis algorithm used by the compiler to automatically copy relevant data between CPU and GPU. Loop optimizations performed by the compiler for GPUs are described in Chapter 6. Performance of the compiler is evaluated over several benchmarks in Chapter 7. Related work is examined in Chapter 8. Finally, Chapter 9 concludes the thesis.

Chapter 2

AMD RV770 architecture for GPGPU

A modern GPU board consists of the core chip, on-board high-bandwidth RAM and various connectors all put on a single board which is then inserted into interconnects such as the PCIe. In this thesis, all experiments were done on AMD's Radeon HD 4870 GPU, which is one of the fastest GPUs available on the market at the time of writing. Radeon 4870 is capable of delivering over a teraflop of performance for 32-bit and 240 gigaflops of performance for 64-bit floating point. Therefore, the theoretical peak of Radeon 4870 is much higher than top-of-the-line x86 CPUs from Intel and AMD.

The Radeon 4870 is based on a chip, codenamed the RV770, and pairs it with high-bandwidth GDDR5 memory. RV770 is a massively parallel-processing core designed both for graphics and more general-purpose computing tasks. RV770 powers various GPUs such as Radeon 4850, Radeon 4870 and Radeon 4830. A power-optimized variant of RV770 powers the Radeon 4890, which is currently the fastest single GPU solution from AMD. AMD has also released smaller and cheaper chips, such as the RV730 and RV740, containing less computational cores, but with designs similar to RV770 and with conceptually the same architecture. Therefore understanding the RV770 is sufficient for understanding the workings of most of the current GPU designs from AMD.

This chapter covers the RV770 architecture as well as the programming model exposed by AMD for GPGPU tasks and code transformations tailored for the architecture. Understanding the architecture is necessary to produce high-performance code for the GPU. As will be shown, the AMD GPU architecture is different from modern CPU architectures and has very different strengths and weaknesses.

The following terminology will be used in the chapter:

1. fp32 : 32-bit floating-point

2. fp64 : 64-bit floating-point

3. float : 32-bit floating-point
4. double : 64-bit floating-point
5. int : 32-bit signed integer
6. float2 : A 64-bit structure of 2 floats. If X is a float2, then it has two float components which can be referenced as X.x and X.y. Numerical indexing is not allowed and .x and .y is the only way to get to components.
7. float4 : A 128-bit structure of 4 floats. The 4 components are respectively x, y, z and w. Numerical indexing is not allowed.
8. double2 : 128-bit structure of 2 doubles with components .x and .y.

2.1 Overview

RV770 is a massively parallel graphics and computing core. An overview of the chip is provided in Figure 2.1. In the figure, the following blocks of RV770 are visible:

1. A setup engine and an ultrathreaded dispatcher that dynamically schedules thread execution on the SIMD units.
2. Ten SIMD units. These form the execution cores of the chip.
3. Ten texture units, where each texture unit is aligned with an SIMD unit. Texture units are equivalent to a load unit on a CPU. Each texture unit also has a dedicated L1 texture cache.
4. Four 64-bit memory controllers (for a total 256-bit memory interface) that can be combined with GDDR3 or GDDR5. Each memory controller has a dedicated L2 cache. The memory controllers are connected to the texture units through a crossbar switch.
5. Various connectors such as PCIe and display controllers.
6. A UVD (Universal Video Decode) block for decoding video codecs.

From the GPGPU performance perspective, understanding the SIMD units and the texture units is important for obtaining the best performance out of the chip. The SIMD units and texture units are described next.

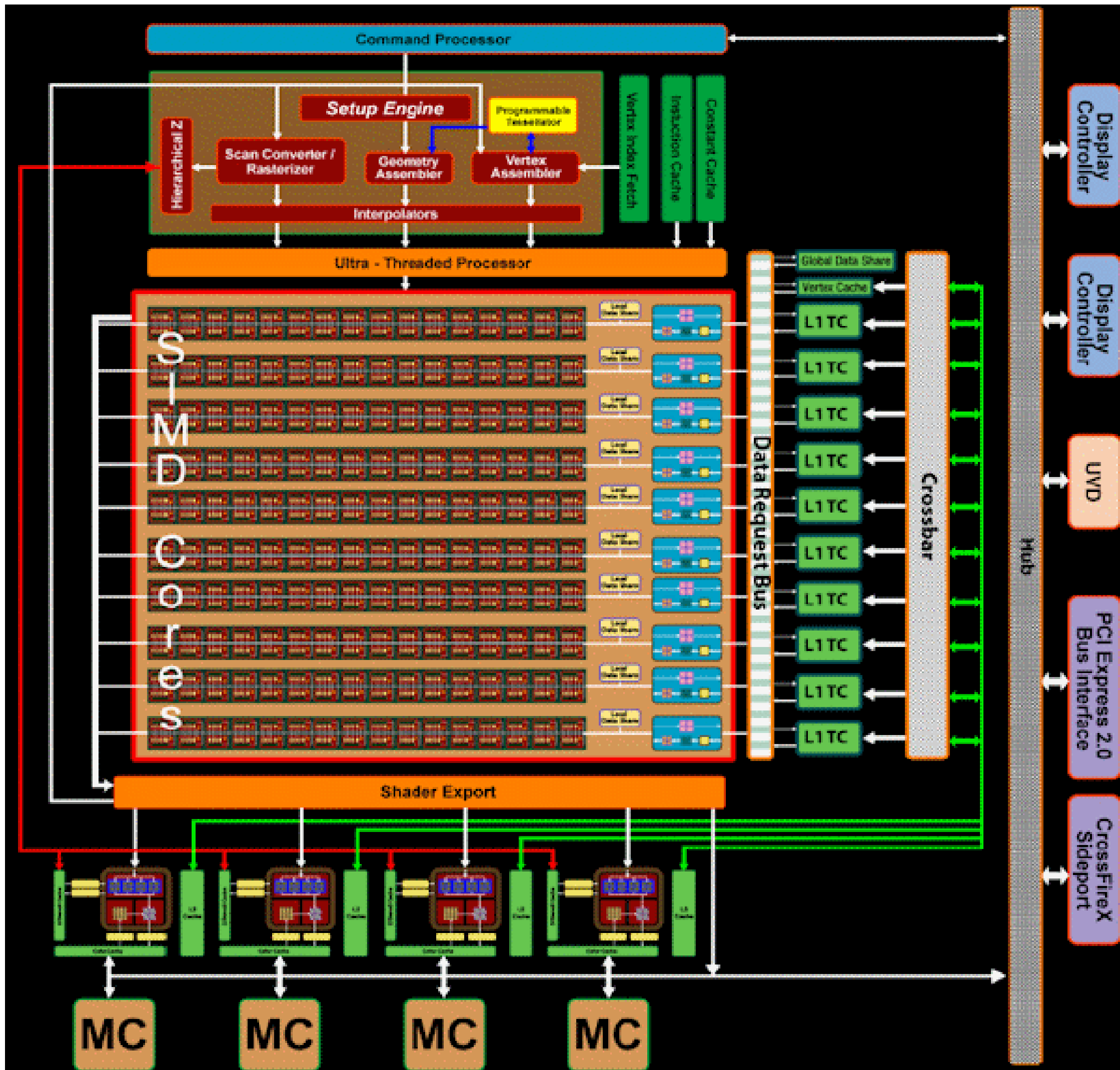


Figure 2.1: RV770 Block Diagram. Source : AMD. Printed with permission.

2.1.1 SIMD units

Each SIMD unit is composed of the following:

1. 16 *thread processors* (TP). Each TP is a VLIW unit that can execute one 5-wide Very Long Instruction Word (VLIW) instructions each cycle. Each TP contains five ALUs. AMD calls each ALU as stream processor (SP). Each SP can execute one fp32 or one int32 multiply-and-add (MAD) operation per cycle. One of the five SPs in a TP is a "fat" SP and only the fat SP can execute transcendental functions such as *sin* or *cos*. Thus, the peak performance of a TP is ten fp32 FLOPs per cycle. For fp64, the scenario is more complex. The four non-fat ALUs co-operate to execute a single fp64 MAD instruction each cycle and thus the fp64 peak of each TP is two FLOPs/cycle or 1/5th of the fp32 peak. If the compiler is unable to pack 5 instructions into one packet, then the peak is not achieved because each TP is a VLIW processor.

2. Each SIMD has a huge register file with 64×256 float4/int4 registers. The registers are untyped with four 32-bit components. Each component of a register can either store a floating-point value or an integer value. Two components of a register can be combined to store a double-precision value. The register file is arranged in sixteen blocks of four banks each. This gives a total of 64 banks and each bank can do one float4 transfer per cycle. Thus the bandwidth to the register file is 1024 bytes/cycle. The peak transfer rate from the register file is less than the peak demand of the ALUs. This is explained by the fact that there exists a forwarding path where a TP can use a result computed in a previous cycle.

3. A 16-Kbyte shared memory called the *local data share* (LDS). The LDS is arranged in four banks, each with 256 128-bit entries. Each bank can do one 128-bit transfer per cycle. Thus the LDS can transfer 64 bytes/cycle. Therefore, the bandwidth from the LDS is much smaller than the bandwidth from the register file. The size and layout of the register file suggests that it is very similar to one block of a register file but with additional indexing hardware.

2.1.2 Texture units

Each texture unit can compute four addresses per cycle. Therefore, the ten texture units can calculate a total of 40 addresses per cycle. Each texture unit has a dedicated L1 cache. The size of the L1 cache has not been published by AMD. Each texture unit is aligned with and services exactly one SIMD unit. Given that an SIMD can execute 800 ALU operations or equivalently 1600 FLOPs per cycle, the asymmetry in the ALU:TEX ratio is apparent. The texture unit provides many facilities, such as filtering, but those are typically not used in compute workloads and are thus not discussed in this thesis.

2.2 AMD CAL Programming model

Various APIs and programming languages are available to program the RV770 GPU for GPGPU purposes. The AMD Compute Abstraction Layer (CAL) Application Programming Interface [8] provides the ability to program the GPU using either the R700 family ISA[10] or using the AMD Intermediate Language (IL) [9]. OpenCL support and DirectX-11 compute shader 4.1 support are also expected in the future for the RV770. To program the GPU using the CAL API, the programmer explicitly manages the GPU. For most efficiency, the programmer must explicitly manage the GPU's on-board memory and must explicitly transfer data between the system RAM and the GPU on-board RAM. While the hardware does provide some ability to directly access the system RAM without explicit transfers to the GPU on-board RAM, the ability is very limited. Accessing the system memory directly over the PCI-e bus is also highly inefficient due to the high latency and relatively low bandwidth.

This document discusses the AMD IL programming model since it corresponds closely to the hardware. AMD IL is a RISC-like program representation derived from the Shader Model 4.0 assembler. The AMD CAL API includes a JIT compiler to compile and run AMD IL programs and also provides routines for managing the GPU memory, initialization and shutdown of the GPU and querying the GPU for available resources. The CAL runtime and the CAL compiler are distributed as part of the AMD graphic driver.

2.2.1 Memory management

In the CAL API, the memory on the GPU is allocated as two-dimensional *resources*. To allocate a resource, the programmer specifies the height, width, format, memory type and location of the resource. The width and height are both restricted to 8192 elements. The format specifies the element type of the resource and can be of any of the numeric datatypes up to 128-bit width. For example, float, float2, float4, int, int2, int4, double and double2 are all valid format specifiers. The location specifies whether the resource is located in the GPU memory or in a driver-allocated portion of system RAM.

For GPU resources, another specifier is the memory type. AMD GPUs are capable of storing resources in two different physical arrangements. The default arrangement is a tiled arrangement where a block of 16×4 bytes is stored contiguously. The other option is to store resources in linear memory that corresponds to row major order, similar to 2-dimensional arrays in C. The two memory arrangements are illustrated in Figure 2.2.

2.2.2 Context management

A GPU can support one or more execution contexts. All execution on a GPU is done through a context. Within a context, resources can be mapped to one or more predefined names. A resource is mapped to a name that specifies how the resource can be used within

0	1	2	3				
4	5	6	7				
8	9	10	11				
12	13	14	15				

Tiled memory arrangement

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15

Linear memory arrangement

Figure 2.2: Memory arrangements

a context. The predefined names are i0, i1, .. i7, o0, o1, .. , o7 and g. A resource mapped as i0, i1,.. i7 can be used as a read-only resource in any kernel launched within the context. A kernel mapped as o0, o1, .. o7 can be used as a write-only render backend within any kernel launched in the context. A resource mapped as 'g' can be used as the unique read-write global buffer that can be read or written to by any thread in a kernel. Any resource can be mapped to any of the names with the only constraint being that the resource mapped to g must have been allocated in linear memory. A resource can be mapped to multiple names but no guarantees are provided about coherence.

One severe limitation in the current AMD SDK is that the global buffer only supports 128-bit elements. For example, address computation is always done assuming that the elements are float4/int4 etc. This means that write patterns that are not aligned to 128-bit boundaries are very hard to support.

2.2.3 Software execution Model

The AMD IL allows two types of shader programs : pixel shaders and compute shaders. The main difference between these programming modes is the way the threads are organized and the memory elements the threads can access.

In pixel-shader mode, threads are organized in a two-dimensional grid and each thread knows its two-dimensional thread id. Pixel shader programs can output to either the render backends or to the global buffer. The write capability to a render backend is limited. A thread with ID (x,y) can only output to index (x,y) in a render backend. The global buffer is indexed through a one-dimensional index. Any thread can read from, or write to, any index in the global buffer but no guarantees are provided about coherence if there is a data race between different threads reading or writing from the same location. Any thread can read from any index from the read-only resources i0 to i7. The end of a kernel is an implicit synchronization point but no other communication or synchronization is provided between threads. An illustration of the pixel-shader programming model is provided in Figure 2.3

In compute-shader mode, threads are arranged in one-dimensional thread groups and thread groups are organized in a one-dimensional grid. Each thread knows its one-dimensional absolute id as well as one-dimensional id within its group. A thread group can have a size of up to 1024 threads. In compute-shader mode, a thread cannot write to render backends and can only write to the global buffer. Threads within a thread group can communicate through the LDS or through shared registers. To utilize the LDS, each thread declares the size of memory that the thread owns in the LDS. Each thread can only write to the piece of memory it owns in the LDS but can read from any index in the LDS. Compute shaders also provide synchronization primitives used to synchronize within a thread group. No communication or synchronization is possible between thread groups.

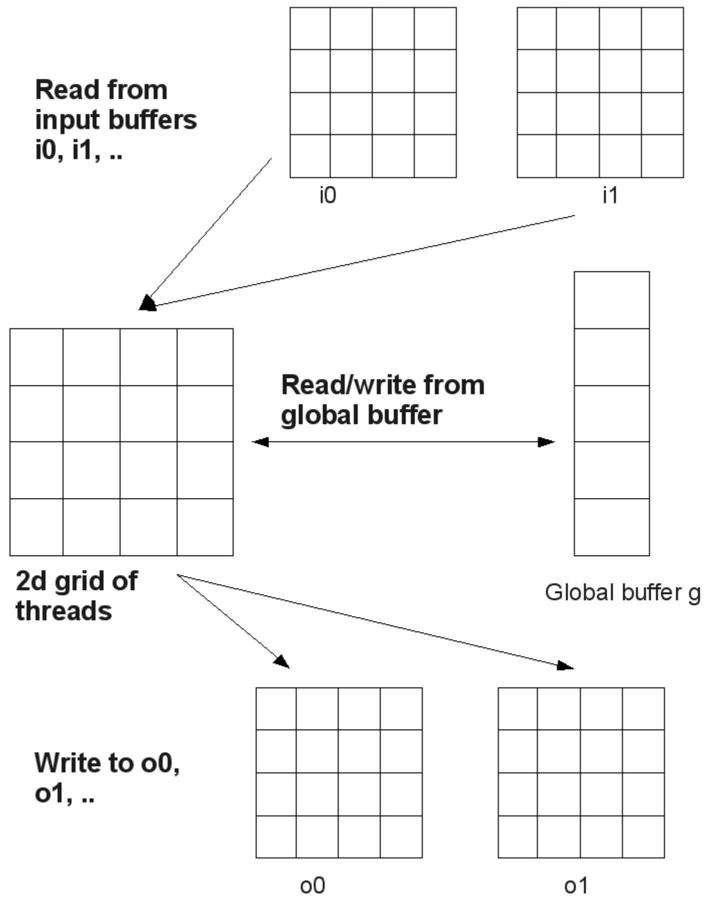


Figure 2.3: CAL programming model

2.3 Hardware execution model

2.3.1 Pixel shaders

The programmer can specify millions of threads to be executed. However not all threads are executed at the same time on the GPU. In pixel-shader mode, the hardware divides the two-dimensional grid of threads into groups of 8×8 thread groups called wavefronts. An SIMD runs one or more wavefronts in parallel. The wavefront sizes vary from GPU to GPU: it is 64 on the RV770 but it is 32 on the RV730. A wavefront starts executing on an SIMD and continues executing until it encounters a non-ALU instruction (such as a texture related instruction). Then the wavefront is swapped out and replaced with another wavefront while the original non-ALU wavefront is sent back to be scheduled on the unit required to complete the instruction that it is waiting on. Thus the GPU uses the massive parallelism to hide the latency of memory loads.

The number of wavefronts that can be run in parallel on an SIMD in pixel-shader mode is decided by the availability of registers. If R denotes the number of registers required per thread, W denotes the wavefront size (64 on RV770), N is the number of wavefronts running in parallel and P is the number of physical registers in the register file, then $N = P/(W * R)$. The driver is responsible for partitioning the physical register file among all the running wavefronts. When a wavefront is swapped out from an ALU to a texture unit, the registers being used by the wavefront are still retained in the register file until the wavefront finishes execution.

For RV770, the wavefront size of 64 is four times the number of thread processors per SIMD. Thus each thread processor is running four threads in parallel and is likely to hide the latency of ALU units. The register file is divided into banks of 64 and thus I speculate that each thread in a wavefront reads and writes to exactly one bank.

2.3.2 Compute shaders

Compute shaders are similar to pixel shaders in most respects. To execute a compute shader, 64 continuous thread-ids in a thread are chosen as a wavefront. The maximum number of wavefronts that can run in parallel in a compute shader is determined by the number of registers used per thread as well as the amount of local data share owned by each thread.

Shared registers

The register file in a compute shader is used not only for the per-thread registers but also for shared registers. Shared registers are registers shared amongst threads in a thread group. However shared registers have the restriction that the register is actually only shared between threads n, n+64, n+128.. and so on within a thread group. Thus thread 0 and thread 1 within a thread group do not actually see the same register. Instead there are 64

distinct and independent copies of a shared register. This is explained by the fact that a shared register is actually shared by threads in the different wavefronts.

There are 64 threads in a wavefront and 64 banks in the register file. When a shared register is declared, one entry in each of the 64 banks is reserved as the shared register thus creating 64 independent copies that are not consistent with each other. Thread 0 and thread 1 can only read/write to different banks. For instance, thread 0 cannot see the copies of the shared register seen by thread 1 because threads in the same wavefront cannot access the same banks. Thread 0 and thread 64, thread 1 and thread 65 *etc* are in different wavefronts but associated with the same bank in the register file and hence can read/write to the 0th and 1st copy of the shared register respectively. Shared registers can be useful in reduction-type kernels. Shared registers increase register pressure because shared registers reduce the register availability in the register file.

Local Data Share

Apart from shared registers, the LDS can be used for data sharing within a thread group. The size of LDS is declared in terms of the number of 128-bit values owned by each thread. All the entries owned by a thread are stored in the same bank in continuous locations. The banks are assigned to threads in a round-robin fashion. Thus, thread 0 is associated with bank 0, thread 1 with bank 1, thread 2 with bank 2, thread 3 with bank 3, thread 4 with bank 0 and so on. Each thread can read/write from any index in the LDS by specifying the thread number that owns the data and the offset within the portion of memory owned by the target thread. However, a thread may need to wait for many cycles before reading from or writing to the LDS because the LDS has only 4 banks and each bank can only service one 128-bit value per cycle.

For data sharing, shared registers access is as fast as non-shared registers but have restricted sharing semantics. LDS has more flexible sharing semantics but the bandwidth from the LDS is much lower than that of the register file.

2.4 Code generation for matrix multiplication

A matrix-multiplication example can be used to illustrate the effect of the ALU:TEX ratio on the performance on a GPU performing general-purpose computations. If the code were to do N loads and N FLOPs, then the ALUs would be idle 95% of the time. Any code that needs to be executed on the RV770 should execute very few load instructions and lots of arithmetic instructions. Ideally the code should contain 20 ALU operations for every TEX instruction.

Two important observations can be made about the GPU. First, the ALUs can only operate on data present in registers. To execute a load, the ALU invokes a `sample` instruction

on the texture unit to bring the data first into a register. Second, the texture units run in parallel to the ALUs. Therefore, to predict performance, the number of cycles required to execute the texture instructions and the number of cycles required to execute ALU instructions are calculated separately. The larger of the two values represents the bottleneck in the computation.

Consider matrix multiplication of matrices of size N*N. the syntax used in the examples is very similar to code accepted by the compiler library, `calseum`, that I wrote. This syntax corresponds roughly to C-like code. First, consider a naive code. This code launches N*N threads each computing one scalar value. Consider the following pseudo-C code that represents the code being executed by one thread.

```
float[] i0;
float[] i1;
float o0;
float sum = 0.0;
for(int i=0;i<N;i++) sum += i0[threadId.x][i]*i1[i][threadId.y];
o0 = sum;
```

The code is performing 2N loads and N ALU operations. A multiply-and-add (MAD) instruction is counted as a single ALU operation but 2 FLOPs. To understand the performance, we need to consider the time required to execute the loop by the texture units, the time required by the ALUs and then take the maximum of these times. In other words, the slowest step determines the execution step.

The total number of loads is 2*N*N*N because there are N*N threads and 2N load instructions per thread. Let the number of cycles required by the texture units be denoted by c_{tex} and the number of cycles required by ALUs be c_{alu}. The texture units can execute 40 instructions per cycle. Therefore, c_{tex} can be calculated as:

$c_{tex} = 2 * N * N * N/(40)$.

The ALUs can do 800 operations per cycle and therefore c_{alu} can be calculated as:

$c_{alu} = N * N * N/(800)$

This performance estimation assumes perfect packing into VLIW instructions and no other overhead on the ALUs. Therefore, the texture units take 40 times longer than the ALUs and are the limiting factor in the above code. Assuming a 750MHz clock rate, the theoretical peak of the code above is limited to 30Gflops due to the texture unit bottleneck. We absolutely have to get rid of the bottleneck to reach anywhere near the 1200 Gflops promised by the hardware.

Let's take a look at ways to optimize the above code. First, try unrolling the loop:

```
float[] i0;
float[] i1;
float o0;
```

```
float sum = 0.0;
for(int i=0;i<N;i+=4){
sum += i0[threadId.x][i]*i1[i][threadId.y];
sum += i0[threadId.x][i+1]*i1[i+1][threadId.y];
sum += i0[threadId.x][i+2]*i1[i+2][threadId.y];
sum += i0[threadId.x][i+3]*i1[i+3][threadId.y];
}
o0 = sum;
```

The loop has been unrolled but it still contains exactly the same ALU:TEX ratio of 0.5 (as opposed to ideal case of 20) and the texture units remain the bottleneck. However, we can take advantage of the fact that the GPU can deal with float4. We can reduce the address calculations in the case of float4 by a factor of 4. Assuming that i0 contains float4 values, the following transformation is possible:

```
float4[] i0;
float[] i1;
float o0;
float sum = 0.0;
for(int i=0;i<N;i+=4){
float4 temp = i0[threadId.x][i/4];
sum += temp.x*i1[i][threadId.y];
sum += temp.y*i1[i+1][threadId.y];
sum += temp.z*i1[i+2][threadId.y];
sum += temp.w*i1[i+3][threadId.y];
}
o0 = sum;
```

Now the GPU is performing four MADs and five loads in every loop iteration giving a slightly better ALU:TEX ratio of 0.8 (as opposed to ideal of 20) resulting in a peak performance of of 1200*0.8/20 = 48 GFLOPs but still limited by texture units. To increase efficiency, we need to increase the amount of work done per thread. By taking advantage of data locality we can reduce the number of load instructions by bringing data into registers and reusing it as many times as possible. The code above only brings data into registers once, does one ALU operation on the data and then discards it. Instead of computing just a single scalar value per thread, we should be calculating a tile of floats.

Lets first consider the case of a 1×4 tile such that each thread is writing a float4 instead of a single float. Assume that i0, i1 and o0 contain float4 values. Then the code is transformed into the following loop (omitting the output lines):

```
float4[] i0;
float4[] i1;
float4[] g;
float4 sum = {0.0, 0.0, 0.0, 0.0 };
for(int i=0;i<N;i+=4){
float4 temp = i0[threadId.x][i];
```

```
float4 temp1 = i1[i][threadId.y/4];
float4 temp2 = i1[i+1][threadId.y/4];
float4 temp1 = i1[i+2][threadId.y/4];
float4 temp1 = i1[i+3][threadId.y/4];
sum.x += temp.x*temp1.x;
sum.x += temp.y*temp2.x;
sum.x += temp.z*temp3.x;
sum.x += temp.w*temp4.x;
sum.y += temp.x*temp1.y;
sum.y += temp.y*temp2.y;
sum.y += temp.z*temp3.y;
sum.y += temp.w*temp4.y;
sum.z += temp.x*temp1.z;
sum.z += temp.y*temp2.z;
sum.z += temp.z*temp3.z;
sum.z += temp.w*temp4.z;
sum.w += temp.x*temp1.w;
sum.w += temp.y*temp2.w;
sum.w += temp.z*temp3.w;
sum.w += temp.w*temp4.w;
}
```

The code above contains sixteen ALU operations and five load instructions resulting in a ALU:TEX ratio of 3.2. This yields a peak theoretical performance of 172 Gflops for the Radeon 4870. We can further increase the tile size to further increase the ALU:TEX ratio. The tile size is restricted by the number of registers available per thread. To hide memory latency, let's assume we run 8 wavefronts per SIMD giving 5120 threads on the chip. Then we get 32 registers available per thread, each of size 128 bits. 32 registers allows for a tile of up to 5×8 giving a ALU:TEX ratio of 12 with a predicted performance of 720 GFLOPs. If the memory access latency is not completely hidden, then the actual performance will be less than the predicted value.

From the matrix multiplication example, it can be concluded that increasing the ratio of ALU instructions to texture instructions is essential for achieving maximum performance on the GPU. Loop transformations such as unrolling and tiling and the use of packed 128-bit datatypes wherever possible can substantially reduce the number of texture instructions executed.

2.5 On-chip bandwidth bottlenecks

A simple way to understand the ALU:TEX bottleneck is to understand the on-chip bandwidth available compared to the bandwidth available on a modern CPU. Lets compare the bandwidth from the L1 cache and the bandwidth from the register file on the Radeon 4870 to an AMD Phenom II X4 940 CPU. Phenom II X4 940 is a relatively high end x86 3.0 GHz quad-core. We compare the bandwidths available to the x4 940 to the bandwidths available

Table 2.1: Comparision of bandwidth available to Radeon 4870 and Phenom II X4 940

Property	Radeon 4870	Phenom x4 940	GPU/CPU ratio
Register file bandwidth	7680 GB/s	768 GB/s	10
L1 cache bandwidth	480 GB/s	384 GB/s	1.25
RAM bandwidth	115.2	17 GB/s	6.77

to the Radeon 4870, which runs at 750MHz, coupled with the 900MHz GDDR5. One core of X4 940 can do one SSE ADD and one SSE MUL each cycle. Thus it must be capable of reading 16 floats or 64 bytes from the SSE register file each cycle. Each core the x4 940 can also load 32 bytes each cycle from the L1 data cache. The bandwidth estimates for the Radeon 4870 and Phenom II 940 are provided in Table 2.1.

The GPU has almost the same bandwidth from the texture unit as the bandwidth to a CPU from the L1 data cache. In situations where the cache bandwidth is the limitation, the GPU simply cannot reach anywhere near the theoretical ALU peak. Therefore the programmer has to spend a considerable amount of effort on loading as much data as possible into registers. Further, while the GPU cache sizes have not been published by AMD, the cache sizes on GPUs have traditionally been much smaller than the cache sizes on a CPU. Taking into account the number of threads that are utilizing the cache at the same time, the cache is of very limited use on the GPU when compared to a CPU. The GPU relies almost entirely upon its large register file and multi-threading to hide load latencies.

2.6 Final Remarks

This chapter described the architecture of a typical contemporary GPU, the AMD RV770. Then it analysed the issues involved with the generation of efficient code for such a GPU. As this analysis showed, the compiler must take into account the memory hierarchy of the GPU and must properly utilize the register file to efficiently feed the ALU.

Chapter 3

Python and the proposed programming model

The objective of this thesis is to provide a high-productivity programming model to the scientific programmer which is portable to both GPUs and multi-core CPUs. Increasingly, scientific programmers are choosing high-level programming languages such as Matlab and Python over languages such as C and Fortran. For parallel programming, shared-memory paradigms, such as OpenMP, are popular with C/C++ programmers.

This thesis provides the programmer with a programming model based on a combination of Python and the concept of parallel loops inspired by OpenMP parallel loops. The choice of Python was based on multiple reasons.

1. Unlike special-purpose languages such as Matlab, Python is a fully featured general-purpose programming language well known for its simplicity and productivity. NumPy is a multi-dimensional array library for Python providing a very flexible array abstraction with concise notations for operations such as slicing. Python in combination with NumPy is well suited for rapidly developing full applications in a wide variety of domains.

2. Python and NumPy are both open source and therefore it was much easier to understand the inner workings of Python and NumPy.

3. Python, in combination with NumPy, is becoming popular in the scientific and numeric computation community because of simplicity and productivity. A library collection aimed at scientific computation for Python, called SciPy, and built upon NumPy is also available. SciPy includes libraries for tasks such as function optimization. An active community is forming around NumPy and SciPy. Various mailing lists, wikis and project listings can be found at the scipy.org website.

This chapter introduces the relevant features of Python and NumPy followed by the proposed extensions to Python and NumPy.

3.1 Python

Python is a high-level, dynamically typed, object-oriented programming language. Python has syntactic support for high-level data structures such as growable vectors (called lists in Python), hashtables (called dictionaries) and tuples. Python also supports operator overloading, decorators, first-class functions and metaclasses.

```
1  def f(n):
2      sum = 0
3      for i in range(n):
4          sum += i
5      #This is a comment
6      return i
```

Figure 3.1: Simple example of a Python function.

Python functions are defined using the keyword `def` and support default and keyword arguments. If a function does not contain a `return` statement, then the function implicitly returns a built-in value called `None`. Figure 3.1 shows a simple example of a Python program containing a function definition and a `for` loop. Python's `for` loop is more general than the `for` loop found in languages such as C and is based on the concept of iterators. Python's `for` loops are written using the notation `for x in y` where the expression `y` must evaluate to an iterable object such as a list. One common expression used for iteration is Python's built-in `range` function. The invocation `range(n)` returns a list of values from `0` to `n-1`. The function `range` requires three arguments: the starting value, the stop value and the step. The starting value defaults to zero and the step defaults to one. In the code of Figure 3.1 the expression `range(n)` returns a list and the resulting list is then iterated over. Any object that supports the iterator protocol can be iterated over. Built-in types such as lists, dictionaries and tuples can all be iterated over elegantly using `for` loops.

Python also supports `generators` to enable a lazier iteration style. For example, the function `xrange` can be used instead of range. Function `xrange` does not return a list but rather it returns a generator. The elements produced by this generator can be iterated, one at a time, without holding the entire list in memory. This form of iteration is advantageous because it reduces memory consumption. In numerical applications, loops of form `for x in range(x,y,z)` and `for x in xrange(x,y,z)` are often used to implement the C-style `for` loops with fixed integer bounds.

3.2 NumPy

Python has no native-language support for arrays. NumPy is an extension module that defines a fast multi-dimensional array class. NumPy is implemented in C and uses Python's support for operator overloading to provide flexible indexing. Indexing in NumPy arrays is zero based and subscripting is bound checked.

NumPy arrays are much more general abstractions than arrays in languages such as C. Before going into a detailed description of NumPy arrays, I first examine some properties of C arrays and pointers.

C arrays differ from C pointers in that C arrays are actually allocated regions of memory while pointers are just views into already allocated data. If a C array goes out of scope, then the memory region allocated for the array is also destroyed. If a pointer goes out of scope, memory being pointed to by it is not automatically freed. Therefore, C arrays can be thought of as owning the data while pointers do not own the data they point to.

NumPy arrays are a general abstraction that encompasses both C array types and C pointers. NumPy arrays contain a field called `data` that points to the raw data buffer holding the data. NumPy arrays also have a flag bit that indicates whether or not the NumPy array owns the data. If a NumPy array owns the data, then the raw data buffer is freed when the array destructor is called. If the NumPy array does not own the data, then the data buffer is not freed when the array destructor is called. Therefore, NumPy arrays can either act as true owners of data or as *views* (or pointers) into data owned by some other object.

NumPy arrays can be indexed with integers just like C arrays or pointers but the address arithmetic is much more general than C pointers. Consider a snippet of C source code in Figure 3.2. The expression $p[i][j][k]$ refers to the address $p + s_1 * i + s_2 * j + s_3 * k$ where $s_1 = m * n$, $s_2 = n$ and $s_3 = 1$. The values s_1, s_2 and s_3 can be thought of as strides in 3 dimensions representing the number of memory locations moved when the corresponding subscript is incremented by unity. Due to the pointer declaration syntax and semantics of C, the last stride (in this case s_3) is always constrained to be 1. The strides are also always positive in C and follow a very regular ordering. For example, s_i is always greater than s_{i+1} in C.

```
1 int i,j,k;
2 .
3 .
4 .
5 char (*p)[m][n] = some_function();
6 char c = p[i][j][k];
```

Figure 3.2: C pointer example

Unlike C, NumPy does not impose any order upon the strides used in index computations. NumPy disassociates strides from the dimensions of the array and allows the programmer to specify arbitrary integer strides for memory address computation. Each NumPy array has a field called `strides`. The field `strides` for an n-dimensional NumPy array is an array of n integers representing the stride in each dimension. A stride in dimension d defines the number of bytes in memory that must be jumped when the index in dimension d is incremented by 1. If arr is a NumPy array with n dimensions, and $(i_0, i_1, ..., i_{n-1})$ is an index into the array, then the actual memory address M referenced is computed as follows:

$$M = arr.data + \sum_{j=0}^{n-1} a.strides[j] * i_j \tag{3.1}$$

The programmer specifies the strides of an array. A stride is an arbitrary integer including zero. The generalized strides enable a NumPy array to emulate C-style row-major array indexing, Fortran-style column-major indexing and many types of non-contiguous data layouts depending upon the programmer-specified strides.

Apart from the generalized stride-based indexing, the second key feature of NumPy is the support for multiple *views* into the same underlying data. NumPy provides many ways to create such views, the most common of which is the slicing operator. Slicing for arrays is defined as follows. Let ar be an one-dimensional array. Let $br = ar[start : stop : step]$ be a slice of the array ar. Then element $br[i]$ is the same element as $ar[i * step + start]$. The length of the array br is given by $\lfloor (stop - start)/step \rfloor$. Slices are views into the original array and no data copying occurs when creating a slice. Multi-dimensional arrays can be sliced independently in each direction.

Figure 3.3 shows some examples of array creation and slicing. Line 1 imports the NumPy library. Line 2 allocates an array `arr` of doubles of size `100`×`300` initialized to zero. The strides of this array emulate a row-major storage order. Line 3 creates a slice of `arr`. The notation `5:97:3` states that the slice starts at `5`, stops at `97` (excluding `97` itself), and has a step size of `3`. Line 4 writes the constant `1.0` in the fifth row, third column of `arr`. Line 5 prints `1.0` because the element `[0,0]` of the slice `v1` corresponds to the element `[5,3]` of the array `arr`. Line 7 prints `2.0`, the constant assigned to the array via slice `v1` in line 6. Each NumPy array has a field called strides that is a tuple storing the strides of the array in each dimension in bytes. Lines 8 and 9 print (2400,8) and (7200,16), respectively, assuming that each element is 8 bytes. Line 10 creates a new view, `v2`, of `arr` where the axes are swapped. Thus, line 11 prints `(8,2400)`. In line 12, a new one dimensional view `v3` of `arr` is created. Line 12 prints `1.0` because element `300*5+3` of `v3` is the same as element `[5,3]` of `arr`. Line 13 creates a view `v4` of `v3` using the slice `[3:100*300:5]` which starts at the element 3 and has stride of 5 elements or 40 bytes. Line 14 prints `1.0` because element `300` of `v4` is the same as element `5*300+3` of `v3` or, equivalently, element `[5,3]` of `arr`.

```
1  import numpy
2  arr = numpy.zeros((100,300))
3  v1 = arr[5:97:3,3:300:2]
4  arr[5,3] = 1.0
5  print v1[0,0]
6  v1[1,2] = 2.0
7  print arr[8,7]
8  print arr.strides
9  print v1.strides
10 v2 = arr.swapaxes(0,1)
11 print v2.strides
12 v3 = arr.reshape(100*300)
13 print v3[300*5+3]
14 v4 = v3[3:100*300:5]
15 print v4[300]
```

Figure 3.3: Examples of array slicing in Python.

The `strides` field can be changed directly without going through slicing or method calls such as reshape. The strides of a NumPy array can also be changed through the NumPy C API.

3.3 Python-C API

Python combines the development flexibility of a dynamically-typed language with an API for efficient C implementation of performance critical portions — functions and classes — of an application. This Python-C API provides many utility functions for passing data back and forth between C and Python. Python exposes all Python objects as a PyObject datatype in C. The API provides various convenience functions and macros to convert between PyObject and C native types such as int, long and structures. Various other C datatypes are also provided for Python types such as list and tuple. However, the base type of all these types is PyObject. NumPy's C-API exposes NumPy arrays as PyArrayObject datatype in C and allows direct manipulation of all fields of the NumPy object.

For implementing functions, the Python-C API requires that any function that is to be used as a Python function must have a predefined type signature and must pass and return pointers to PyObjects because the Python interpreter only knows about PyObjects.

Python modules implemented in C utilizing the Python-C API are called Python extension modules. To write an extension module `foo` in C, a programmer first writes the code for the functions to be implemented in C in one or more C files. The programmer then writes a special module initialization function, which must be named `initfoo` if the desired module name is `foo`. The C code is then compiled into a single dynamically loadable library (DLL on windows or shared object files on linux) called `foo.so` on Linux or `foo.dll` on

Windows. The module initialization function passes a symbol table mapping C strings to C pointers to functions or variables of suitable type to be exported to Python. When the Python interpreter encounters an `import foo` statement, it first looks for a file called foo.so. If foo.so is found, then the interpreter looks for the function initfoo to initialize the module. If the initfoo function is not found, then Python throws an exception. If the file foo.so is not found, then Python looks for a Python source file called foo.py.

For details, see Python's official C-API documentation[6].

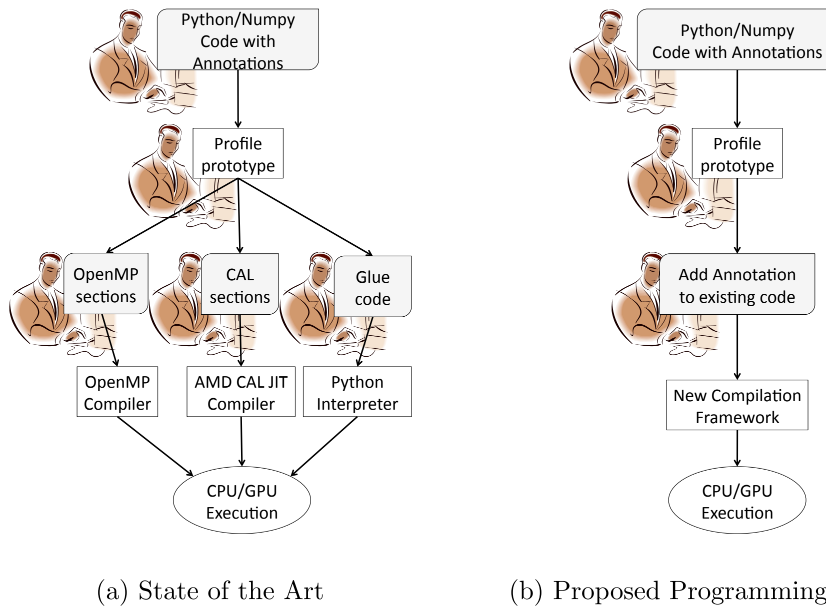
3.4 Programmer's workflow

The typical development of applications containing performance-critical, numerical-intensive, sections of code in Python consists of writing a prototype in Python, profiling the code to identify performance-critical sections, and then re-writing these sections in a compiled language such as C or C++. The programmer also writes *glue code* in C/C++ using the Python-C API. Figure 3.4(a) presents a block diagram that illustrates this state-of-the-art development process and emphasizes the code that has to be written by a developer. Rewriting the code in C/C++ is a tedious and error-prone process that reduces developer's productivity and results in a code base that is less maintainable. If a GPU version of the code is also required, then the amount of code to be written by the programmer, and the complexity of the code base, is further increased.

This thesis presents an alternative for the development of such applications in Python, as illustrated in Figure 3.4(b). It introduces a new compiler system for automatic generation of C++ code for the performance-critical Python sections. The programmer is not required to rewrite the code in C++. Instead, the programmer annotates the Python code with type declarations and parallel loop annotations.

The compiler framework that supports this new programming model is described in detail in chapter 4. The compiler automatically generates the C++ code as well as the required glue code for the annotated code. The programmer then invokes a standard C++ compiler to compile the code into a DLL. This DLL is a drop-in replacement for the original Python module. To generate code for multi-core CPUs, the programmer only needs to add parallel-loop annotations, where possible, to the code before compilation. To take advantage of GPU acceleration, the programmer simply annotates certain parallel loops to be executed on the GPU and the compiler handles everything else.

The implemented compiler can only deal with a subset of Python and requires type annotation. However, since only a small portion of the application needs to be compiled to C++, only a small portion of the Python code needs to conform to these restrictions. The rest of the application, which does not need to be compiled, remains as is and requires no changes.



(a) State of the Art (b) Proposed Programming Model

Figure 3.4: Comparison between state of the art for generation of numerical-intensive applications for execution in CPU/GPU and the proposed programming model.

Furthermore, we designed the annotations in such a way that they are still compliant with the Python grammar and are transparent to the Python interpreter. Thus if the programmer does not wish to compile the module to C++ during development to avoid the build and link steps, the annotated Python will run on the interpreter.

3.5 Proposed extensions to Python

For efficient and simple compilation, several extensions to Python were introduced in the form of various annotations. The key idea behind the syntax of the annotations is that no new keywords should be introduced and the grammar of the language should not be altered. The language should remain compatible with the standard Python interpreter and any annotations introduced should be essentially transparent to the interpreter. However, when the unPython compiler encounters the annotations, it treats them as special compiler directives and accordingly generates the appropriate C code.

Thus, even if the code is not compiled, the code can run on any standard Python interpreter. The ability to run on the standard Python interpreter has the benefit of faster development and debugging cycles for the programmer by avoiding the compilation step until the code is fully debugged. The interpreter compatibility also ensures easier debugging of the compiler because a mismatch between the compiled and the interpreted versions of the code usually indicates a bug in the compiler.

3.5.1 Type annotations

Type declarations enable the compiler to generate efficient C code. Python is a dynamically typed language and has no type declaration syntax. For this purpose, an extension was introduced to Python to declare the type signature of a function.

In this extension, the programmer creates a special decorator to annotate each function to be compiled. This decorator declares the types of the function's parameters and of its return value. Decorators are annotations added just before a function definition and are a standard Python syntax feature. Optionally, the types of local variables may also be declared in a similar fashion. If no such declarations are provided, the compiler infers the types of a local variable based on the type of the first value assigned to the variable in the function. In this framework, a variable cannot change its type within a function.

Figure 3.5 shows an example of type-declaration syntax. The decorator `unpython.type` is added to the function declaration. Types are passed to the decorator as arguments. If the function takes x number of parameters, then $x + 1$ types are required. The first x types are for the parameters and the last type specified is for the return type of the function. In the example, the first type is for the parameter `n` and the second type is for the return type. The types of `sum` and `i` are automatically inferred by the compiler to be int64 which is the default integer type. These types can be forced to be `int32` by adding type declarations for the local variables.

```
1  @unpython.type('int64','int64')
2  def f(n):
3      sum = 0
4      for i in range(n):
5          sum += i
6      return sum
```

Figure 3.5: Example of decorator used as a type declarator.

3.5.2 Parallel annotations

The Python interpreter does not provide any support for parallel programming. One way to take advantage of multiple processor cores is to utilize process-based parallelism where multiple instances of the Python interpreter may be launched. Process-based parallelism is not suitable for all types of problems and is not considered in this thesis. The only way to take advantage of multiple processor cores within a single Python process is to write a multithreaded function in C using an API such as pthreads or OpenMP. However, the current Python interpreter is not multithreaded and calling Python-C API functions from two different C threads is unsafe. Therefore, the programmer must be careful about calling

Python-C API functions from multithreaded functions.

I introduce an extension to mark certain `for` loops as parallel to provide a shared-memory data-parallel model similar to (but less general than) the OpenMP parallel `for` loop directives. To avoid the complexity of specifying parallel semantics for arbitrary iterators, only a parallel version of `xrange` iterator is defined. A special iterator, `prange`, is introduced that is used to specify that a loop is parallel. The iterator `prange` is only a hint to the compiler that the programmer intends to execute the loop in parallel but the compiler is not obligated to compile the loop to a parallel C++ loop. To the Python interpreter, `prange` is identical to `xrange`. However, unPython treats the annotations as a special parallel-loop annotation. A `for` loop over the parallel iterator `prange` has the following properties:

1. Let the program point immediately before the loop be P. Then the program execution is serial before P.

2. Let the program point immediately after the loop be Q. Then the program may spawn multiple threads between P and Q. The point Q is an implicit join-point. Program execution does not continue until all threads spawned reach Q. After Q, serial execution of the program continues.

3. Each iteration of the loop is independent and may be executed in any order.

4. Parallel loops may be nested but, in this model, a join point only exists at the ending of the outermost parallel loop.

5. Parallel loops are a compiler directive and the compiler is not obligated to parallelize the loop.

6. If there is a data-race between any two iterations, then the output of the program is undefined.

7. The body of the loop may only access variables of basic numeric types (float32, float64, int32 and int64) or NumPy arrays of basic numeric types. If any other object type is referenced in the loop, then the loop is not parallelized by the compiler to avoid the possibility of calling Python-C API functions from two different C threads.

8. All variables defined outside the loop are treated as shared variables among threads. However, a programmer can define some variables as private by passing the keyword argument `private` with the value equal to a tuple containing a list of variable names as strings.

9. All variables local to the loop are treated as private.

UnPython generates OpenMP code for parallel loops that will be executed in multi-core CPUs.

A variant of `prange` called `gpurange` is also introduced. This parallel loop type is meant for acceleration by the GPU. The compiler is responsible for generating GPU code as well as for managing data transfers between CPU and GPU automatically. However, the compiler is not obligated to utilize this directive and may not generate GPU code. Further, if the target machine does not have a suitable GPU or if GPU code generation is disabled through compiler command line options, then the compiler does not generate any GPU code. Thus, the same code can easily be ported to multiple platforms.

3.5.3 Constraints on Python code to be compiled

To efficiently compile Python, unPython only accepts a subset of Python that is critical for the performance of numerical applications. Currently, unPython does not support features such as runtime code execution, higher order functions, generators, metaclasses and special methods such as `setitem`. UnPython supports 32- and 64-bit integers and 32- and 64-bit floating point numbers, but it does not support arbitrary-precision arithmetics. Ensuring no overflows is the responsibility of the programmer. These restrictions only apply to the compiled code. All features are available for the non-compiled portion of the application code that is executed by the Python interpreter.

3.6 Conclusion

This chapter provided a brief overview of the syntax and semantics of Python, NumPy and the proposed extensions. These extensions were carefully designed to ensure that the parallel-annotated code written for execution on a GPU can also be run by an unmodified Python interpreter. The flexibility of NumPy arrays combined with the proposed parallel loop extensions provides the programmer with very productive tools to write parallel numerical programs.

Chapter 4

Compiler implementation

The compiler system presented in this thesis is a hybrid combination of two different compilers: unPython, an ahead-of-time compiler and jit4GPU, a just-in-time compiler. Generating CPU code is solely the responsibility of unPython. For GPU code generation, several analyses need to be performed that cannot be done completely ahead of time. Therefore, code generation for GPU is primarily done by jit4GPU assisted by information supplied by unPython. To understand the working of the entire system, it is simpler to consider the code generation process for CPUs and GPUs separately.

4.1 CPU code generation

Consider a Python program to be compiled for a multi-core CPU target. The programmer invokes unPython passing the Python filename to unPython. A block diagram of unPython is given in Figure 4.1. UnPython reads the Python source code and generates C++ and OpenMP code for execution of the program on the CPU. UnPython is a simple translator and therefore it only performs a few transformations.

The front-end of unPython is simply a Python script that parses Python code using the standard Python interpreter's parsing interface. The parser returns an abstract syntax tree (AST) and the AST is dumped into a file by the front-end script. The AST is then read back into the middle end of unPython. After parsing, unPython performs type-checking which includes type inference for local variables. The initially untyped AST is converted into a typed AST by the type-checking phase. The typed AST is then *lowered* into a simpler typed AST. Lowering the AST involves breaking down complex expressions and statements into simpler expressions and statements that are closer to the desired C++ code. After lowering, unPython performs liveness analysis using an iterative dataflow algorithm. The liveness analysis algorithm is currently very restricted and only works on functions that only manipulate numeric scalar types and numeric NumPy arrays. If a function involves objects of other types, the liveness analysis is not performed. Results of the liveness analysis are used

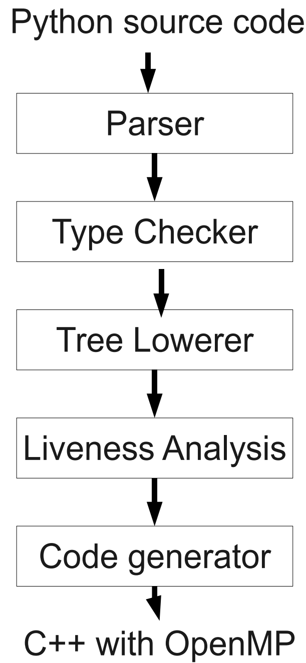


Figure 4.1: Block diagram of unPython for CPU code generation

by the compiler to determine which scalar variables referred to within a parallel loop can be declared as *private* when generating OpenMP code. After liveness analysis is complete, the compiler simply traverses the typed AST to generate the final C++ and OpenMP code. To generate code, the compiler generates the function and method bodies required. All boiler-plate code required (such as wrapper functions for Python-C API) is generated with the help of a text-templating system. UnPython is implemented in a combination of Java and Scala languages. Scala is a statically-typed object-oriented language with many features derived from functional programming languages and it runs on the JVM. FreeMarker text templating library for Java is used for generating boiler-plate code.

4.2 GPU code generation

For GPU code generation, simple translation is not sufficient. The objective is to map a shared-memory parallel-loop programming model to a heterogeneous system with two distinct address spaces. Therefore, the compiler needs to not only generate GPU code, but also perform analysis to determine which data must be transferred between the two distinct address spaces. The compiler needs to know the memory access patterns of all the array references inside the loop to analyze the data to be transferred to the GPU. The memory

access pattern analysis and GPU code generation is done by a JIT compiler jit4GPU and unPython only plays a supporting role in GPU code generation.

4.2.1 Motivation for the JIT compiler

One of the original goals in the project to compile Python was to extend unPython to generate GPU code. However, after some effort, it became clear that unPython does not have sufficient information to perform the analysis required for GPU code generation. For example, the compiler does not know the data layout and the aliasing of NumPy arrays that are parameters to a function being compiled. NumPy arrays are generally views of an underlying data array and are more like pointers than true arrays from the perspective of compiler analysis. The memory layout of a NumPy array is determined by the strides of the NumPy array and these strides are dynamic quantities unknown to unPython. Two apparently distinct NumPy arrays passed into a function may be two different views of the same piece of memory. A somewhat similar problem arises in C functions that have pointer parameters. Typically C compilers perform a global analysis and can customize the function differently for different calling contexts. However, such an analysis is not applicable for unPython. UnPython is not a whole-program compiler and therefore cannot do global analysis, such as finding the calling context of all functions.

The objective of unPython is to compile Python libraries that can be used by various applications. The idea is that unPython will only see a small portion of the application because only a small portion of the application is performance critical. Python programmers cannot be expected to impose the typing restrictions required by unPython on their entire program. Even if a whole-program compiler were implemented for Python, Python applications often contain bindings to C libraries and those libraries are opaque to a Python compiler unless it is also interfaced with a full C compiler. The complexity of such a system can be prohibitive to implement. Thus, effectively, unPython cannot do any global analysis. In absence of a global analysis, one other possible solution for generating code from unPython was to generate different versions of the loop for different cases of layouts of NumPy arrays accessed within a loop. But NumPy arrays are very general structures and the number of possible loop versions can become intractable as the number of NumPy arrays increases.

A simpler solution to the problem of performing analysis for GPU code generation is to compile a parallel loop to GPU code just before the loop is to be executed. At this stage, jit4GPU can query all the NumPy arrays to determine their data layouts and also knows the value of loop-invariant numeric constants. For example, loop bounds may appear to be unknown symbolic constants to unPython but those constants are known to jit4gpu. Jit4GPU operates on a typed AST of the loop to be compiled and performs several analysis

and optimizations detailed later in Chapters 5 and 6.

The JIT compiler is actually not visible to the programmer. As far as the programmer is concerned, an additional library (the JIT compiler) is linked into the application but the programmer is not concerned with the function or inner workings of the JIT compiler. The JIT compiler also makes no difference to codes that do not target GPUs.

4.2.2 GPU code generation

Now consider the full process of generating GPU code. When unPython encounters a loop that is marked by the programmer to be compiled for execution on the GPU, it outputs two versions of code for that loop. First, unPython outputs a typed AST representation of the parallel loop to be used by jit4GPU for analysis and code generation. The representation is printed out in a Lisp-like S-expression format read by jit4GPU along with a symbol table and a table of array references inside the loop. However, jit4GPU is not guaranteed to produce GPU code and can return failure if any of the analysis phases fails. Therefore, unPython also generates a C++ and OpenMP codepath for the parallel loop which is used as a fallback in case jit4GPU fails to generate GPU code.

Jit4GPU is implemented as a multi-phase compiler operating on typed ASTs. An overview of jit4GPU is provided in Figure 4.2. In the first phase, jit4GPU performs an array-access analysis detailed in chapter 5 to determine the data to be transferred to the GPU. The array access-analysis phase produces two outputs: a data structure representing the data transfers to be performed and a tree of code to be executed on the GPU. If jit4GPU fails to perform the access analysis, then it returns failure. In the next phase, jit4GPU optimizes the typed AST using loop optimizations detailed in chapter 6. The loop optimization phases optimize the AST and can also alter the data layout chosen on the GPU and can therefore also change the parameters for data transfer. In the next phase, jit4GPU generates AMD IL assembly from the AST and passes this assembly code to the compiler that converts AD IL code to GPU binaries. The AMD IL to binary compiler performs extensive low-level code transformations, such as physical register allocation, on the GPU and instruction scheduling on various units of the GPU. The exact optimizations performed by the AMD compiler are not published by AMD but I have observed, looking at the disassembler output, that the AMD CAL compiler performs code transformations. In the final phase, jit4GPU issues a call to a supporting runtime library to perform the actual data transfer and to execute the code on the GPU.

Once the execution is finished and data is transfered back to the CPU, jit4GPU reports success and the execution of the program resumes on the CPU from the program point just after the parallel loop.

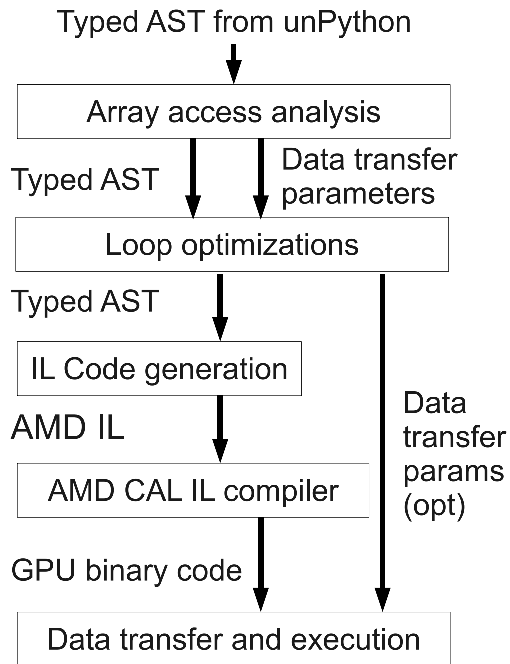


Figure 4.2: Block diagram of unPython for CPU code generation

4.3 Final Remarks

This chapter described the design of a new compiler system that combines an ahead-of-time compiler, a just-in-time compiler and the AMD CAL IL compiler to create a compilation path for parallel-annotated Python numerical code. This compilation infrastructure generates flexible code that execute either in a GPU or in a multicore CPU.

Chapter 5

Array access analysis

Array access analysis is performed by a compiler to compute the set of memory locations that may be accessed by array references occuring in a loop nest. Jit4GPU performs array access analysis to compute the set of memory locations accessed in a parallel loop to be executed on the GPU. Jit4GPU then computes a one-to-one mapping of the set of memory locations accessed in the system memory to a set of memory locations on the on-board memory of the GPU. Jit4GPU also checks if all the memory potentially accessed within the loop can fit in the limited on-board memory of the GPU. If the data accessed during computation does not fit on the GPU, then jit4GPU attempts to tile the loop. Finally, if the array access analysis and potential tiling of the loop are successful, then jit4GPU generates data transfer parameters and GPU binary and executes the loop.

This chapter describes a new array access analysis algorithm used by jit4GPU .

5.1 Representation and problem statement

To execute a loop nest on the GPU, the compiler first needs to compute the set of memory locations that can potentially be accessed by the loop nest and then transfer the data to the GPU. The compiler also needs to compute the size of the set of memory locations accessed to determine if the set will fit on the GPU. In this work, the analysis is restricted to loop nests of the form shown in Figure 5.1.

In Figure 5.1, a loop nest of depth d, where the top m loops are parallel, is shown. Variables i_1 to i_d above are loop counters while u_1 to u_d are upper bounds of the loops. The loop body may contain zero or more array accesses. The set of memory locations accessed by an array reference depends on the subscript expression used in the reference, the mapping of the subscript expression to memory locations based on strides of the array and on the loop bounds.

For array access analysis, the compiler requires a formal representation of the memory access pattern. Such a representation is a mapping from the d-dimensional space of tuples of

```
1  for i_1 in prange(u_1):
2      for i_2 in prange(u_2):
3          ...
4          for i_m in prange(u_m):
5              for i_{m+1} in range(u_{m+1}):
6                  .
7                  .
8                  for i_d in range(u_d)
9                  #loop body
```

Figure 5.1: Loop nests considered for array access analysis

loop counters to one-dimensional space of memory locations. The space of memory locations represents the locations in virtual memory. This space starts at zero and ends at positive infinity. The memory space is assumed to consist of discrete cells where each cell is composed of one discrete unit of memory. If all the arrays in the loop have the same element size s, (for example if all arrays are arrays of 4-byte values), and if the starting position of all arrays is aligned on s bytes, then one cell of memory can be thought of as having s bytes. In this work, I restrict myself to the case where one cell of memory is equated to the size of one element of the array.

For array-access analysis, representing and reasoning about arbitrary memory access patterns is not feasible. Thus array-access analysis is typically restricted to representing one particular class of memory access patterns. Linear Memory Access Descriptors or LMADs, first introduced by Paek *et al.* [17], are one such representation capable of precisely representing a wide variety of memory access patterns.

Definition 1. *An LMAD L is a mapping from a d-dimensional space of tuples of loop counters to the 1-dimensional space of memory locations of the form:*

$$L(i_1, i_2, .., i_d) = c_0 + \sum_{k=1}^{d} f_k(i_k) \tag{5.1}$$

where $\forall(1 \le k \le d)$, starting with $k = 1$ and going to $k = d$:

$$0 \le i_k < u_k \tag{5.2}$$

$$u_k = g_k(i_1, i_2, .., i_{k-1}) \tag{5.3}$$

$$i_k, u_k, c_0, f_k(i_k), g_k(i_1, i_2, .., i_{k-1}) \epsilon \mathbb{Z} \tag{5.4}$$

where L is the LMAD, $\mathbb{Z}$ is the set of integers, i_k is the k-th loop counter, c_0 to c_d are integer constants and u_k is the upper bound of the k-th loop counter. Functions f_k and g_k are arbitrary integer functions.

LMADs can represent a wide variety of access patterns but require the compiler to retain the symbolic representations of the functions f_k and g_k. Thus representing and reasoning about general LMADs requires a general symbolic algebra engine to be embedded in the compiler. Instead of dealing with general LMADs, this thesis is limited to a simpler, yet common, subset which I term as RCSLMADs for restricted constant stride LMADs. Intuitively, RCSLMADs represent a class of affine subscript expressions occurring inside a rectangular loop nest with loop invariant bounds.

Definition 2. *An LMAD L is an RCSLMAD if and only if it satisfies all of the following conditions:*

1. *Let N be the set of integers $\{1, 2, .., d\}$. Then L must be of the following form:*

$$L(i_1, i_2, ..., i_d) = c_0 + \sum_{k=1}^{d} p_k * i_k \tag{5.5}$$

where $\forall(k \epsilon N)$:

$$0 \leq i_k < u_k \tag{5.6}$$

$$p_k, u_k \epsilon \mathbb{Z}^+ \tag{5.7}$$

$$c_0 \epsilon \mathbb{Z} \tag{5.8}$$

$$c_0 \geq 0 \tag{5.9}$$

$\mathbb{Z}^+$ *is the set of positive integers.*

2. *There must exist a one-to-one mapping G from N to N such that $\forall(k \epsilon N)$:*

$$p_{G(k)} > \sum_{k+1}^{n=d} p_{G(n)} * (u_{G(n)} - 1) + p_{G(n)} - 1 \tag{5.10}$$

where $G(k)$ represents the application of the mapping G on the value k. The mapping G is called the ordering function *of L.*

For convenience, a number of auxiliary terms can be defined. Let L be an RCSLMAD. Then:

Definition 3. *The integer constant c_0 is called the base of the RCSLMAD.*

Definition 4. $\forall(k \epsilon N)$*, the integer constant p_k is called the stride in the k-th dimension.*

Definition 5. *The mapping G is called the ordering function of the RCSLMAD L.*

Definition 6. *$\forall(k \epsilon N)$, the value $p_k * (u_k - 1)$ is called the* span *in the k-th dimension and represents the distance traveled in memory space when the k-th loop counter i_k goes from 0 to the upper bound u_k.*

The definitions of stride and span are compatible with the definitions given by Paek *et al.* for stride and span of general LMADs.

Definition 7. *Let there be a set D such that*

$$D = \{(i_1, i_2, ..., i_d) : \forall(1 \leq k \leq d), 0 \leq i_k < u_k\} \tag{5.11}$$

The set D is called the domain of the RCSLMAD L.

Definition 8. *Let D be the domain of RCSLMAD L and let $V \epsilon D$ be a d-tuple. Then the value $v = c_0 + \sum_{k=1}^{d} V(k) * p_k$ is called the application of L on V and is denoted by $L(V)$.*

To understand the concept of RCSLMADs, an example can be considered. Consider a two-dimensional example:

$$L_1 = 5 + 20 * i + 3 * j \tag{5.12}$$

$$0 \leq i < 7 \tag{5.13}$$

$$0 \leq j < 5 \tag{5.14}$$

To visualize the RCSLMAD, consider a matrix with 7 rows and 20 columns. The number of rows was chosen to be equal to the upper bound of i while the number of columns was chosen to be equal to the stride of i. Then set the element in i-th row and j-th column of the matrix to $5 + 20 * i + j$. The matrix is effectively constructed to represent a row-major two-dimensional array (such as those present in programming languages C/C++) of size $7 * 20$ beginning at memory address 5. Figure 5.2 highlights all the elements in the matrix which belong to the RCSLMAD L_1. The RCSLMAD is representing a regular pattern of accessing the matrix similar to the access of the form $[i][3*j]$ of a C-style two-dimensional array. This is the intuitive idea behind RCSLMADs. All the accesses are effectively accessing a very simple strided pattern in a C-style d-dimensional matrix.

RCSLMADs have several useful properties. To describe the properties of RCSLMADs, we will use notation consistent with the above definitions. L will represent an RCSLMAD defined over a d dimensional domain D. Symbols i_k, u_k, p_k, σ_k will represent the loop counter, upper bound, stride and span respectively in the k-th dimension.

Theorem 1. *L is a one-to-one mapping from D to the memory space. Alternately, each d-tuple of loop counters from the domain D is mapped to a unique location in memory by the RCSLMAD L.*

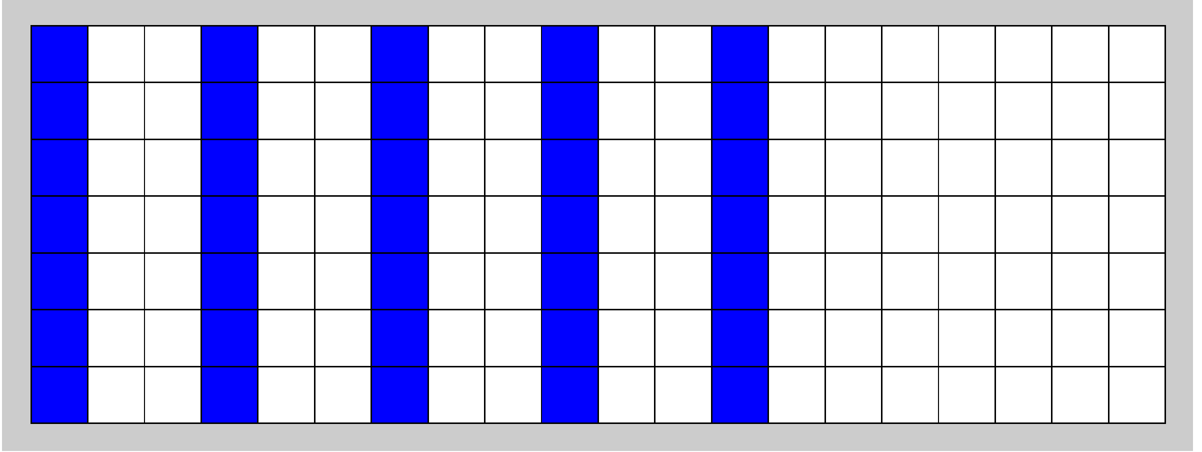


Figure 5.2: Visualization of a two-dimensional RCSLMAD example

Proof. Without loss of generality, the ordering function G is assumed to be the identity function. The theorem can be proved by contradiction. Let us assume that there exist 2 tuples V_1 and V_2 such that

$$L(V_1) = L(V_2), V_1 \epsilon D, V_2 \epsilon D, V_1 \neq V_2 \tag{5.15}$$

Then the d-dimensional tuples V_1 and V_2 must differ in at least one dimension. Without loss of generality, we assume $V_1 > V_2$ lexicographically. Let m be the smallest integer such that $V_1(m) > V_2(m)$. Then

$$L(V_1) - L(V_2) = p_m * (V_1(m) - V_2(m)) - \sum_{k=m+1}^{d} p_k * (V_2(k) - V_1(k)) = 0 \tag{5.16}$$

$$p_m * (V_1(m) - V_2(m)) = \sum_{k=m+1}^{d} p_k * (V_2(k) - V_1(k)) \tag{5.17}$$

$$min(p_m * (V_1(m) - V_2(m))) = p_m \tag{5.18}$$

$$max(\sum_{k=m+1}^{d} (p_k * (V_2(k) - V_1(k)))) = \sum_{k=m+1}^{d} (p_k * (u_k - 1)) \tag{5.19}$$

However, by definition of RCSLMAD:

$$p_m > \sum_{k=m+1}^{d} (p_k * (u_k - 1)) \tag{5.20}$$

$$\therefore min(p_m * (V_1(m) - V_2(m))) > max(\sum_{k=m+1}^{d} (p_k * (V_2(k) - V_1(k)))) \tag{5.21}$$

$$\therefore L(V_1) - L(V_2) > 0 \tag{5.22}$$

Thus we have arrived at a contradiction and proved that for RCSLMADs, each d-tuple in the domain D maps to a distinct location in memory. □

Theorem 2. *The size S of the set represented by the RCSLMAD L is given by* $S = \prod_{k=1}^{d} u_k$.

Proof. The size S of the set is equal to the size of the domain D because each element in the domain maps to a distinct memory location. The size S can simply be computed as a product of the upper bounds because the domain is rectangular. □

A loop nest may have multiple RCSLMADs and they may overlap at one or more memory locations. If each RCSLMAD is transferred independently, then the same memory location on the CPU may be copied to multiple copies on the GPU. If there is a read/write dependence involving a memory location, then creating multiple copies on the GPU is cumbersome because the copies must be kept consistent. A better solution is to compute a set union of the RCSLMADs and then transfer the union to the GPU. This way there will be exactly one copy of M on the GPU. The compiler also ensures that no other CPU thread writes to the memory locations copied to the GPU before the GPU execution and data transfers are complete. For correctness, it is not necessary to compute an exact union. If the union is represented by U, then it is sufficient to compute a superset U of the union. We now present the problem statement including the design criteria of an analysis to compute the union:

Problem Statement 1. *Given a list* $\{L_1, L_2, .., L_n\}$ *of RCSLMADs defined over a d-dimensional domain D:*

1. *Compute the set M of memory locations to be transferred to the GPU. The set M should be a superset of the union U of RCSLMADs. The number of elements in $M - U$ should be as small as possible. M should be easily representable by the compiler.*

2. *Compute the size S of the set M.*

3. *Construct a map F with domain M. The range R of F should be a set of valid memory locations in the GPU address space. If $F(m), m \epsilon M$ represents the address of the memory location m on the GPU, then F should be such that $F(m_1) \neq F(m_2)$ if $m_1 \neq m_2$. The Range R should be as small as possible. The computational complexity and memory requirement for computing F should be as low as possible.*

5.2 General solution

Consider the case where the list contains a single RCSLMAD L defined over a domain D. Without loss of generality, assume that the ordering function of L is the identity function. In such a case, an exact solution for the problem 1 can be calculated by algorithm 1.

Algorithm 1 solves the data transfer problem for a single RCSLMAD L defined over a domain D.

1: $M = \{L(V) | V \epsilon D\}$.

2: $S = \prod_{k=1}^{d} u_k$ from property of RCSLMAD.

3: Let $m = L(V), V \epsilon D$, then $F(m) = \sum_{k=1}^{d} (V(k) * (\prod_{j=k+1}^{d} u_j))$ and range $R = S$.

To argue that the algorithm 1 does produce a correct function F, we observe that given a d-dimensional tuple D, the composite function $F(L)$ is also an RCSLMAD defined over the domain D, and hence each distinct tuple V is mapped to a distinct location in the GPU memory by the composite function $F(L)$. The range R can be directly verified by calculating the maximum and minimum values of $F(L)$ over D.

The problem of multiple RCSLMADs can be simplified in many cases by considering the memory intervals spanned by the RCSLMADs in the list. If the memory intervals spanned by the RCSLMADs in the list are completely disjoint, then each RCSLMAD can be analyzed individually without any necessity to compute the union because there can be no dependencies and no possibilities of multiple copies. If the interval spanned by two or more RCSLMADs overlaps, then we can form a *group* of RCSLMADs. A group of RCSLMADs spans over an interval given as the union of intervals of all members of the group. In general we can partition the list of RCSLMADs into groups of RCSLMADs where the interval represented by each group is disjoint from other groups. An algorithm to compute groups of RCSLMADs is given by algorithm 2.

Algorithm 2 partitions a list of RCSLMADs defined over a domain D into disjoint groups of RCSLMADs.

Inputs: List $\{L_1, L_2, .., L_n\}$ of RCSLMADs defined over a domain D.
Outputs: List $\{G_1, G_2, .., G_n\}$ of groups of LMADs such that the interval spanned by each group is disjoint from the interval spanned by any other group.

1: **for** each RCSLMAD L in $\{L_1, L_2, .., L_n\}$ **do**
2: Find out the start and end addresses of R in memory.
3: **end for**
4: Construct a graph G where each RCSLMAD is a node.
5: **for** each pair of nodes in G **do**
6: Insert an edge if the start-to-end intervals of the two nodes overlap.
7: **end for**
8: Find the connected components in the graph G using a suitable algorithm such as a depth first search. Each connected component represents a group of RCSLMADs.
9: **return** A list of connected components.

If one or more groups contains more than one RCSLMAD, then the problem of computing the union (or the superset) still remains. One general solution is described in algorithm 3.

The algorithm is very conservative and transfers all memory locations in the memory interval spanned by the group. The algorithm is correct but transfers too much data and uses too much GPU on-board memory. Better solutions are needed in most cases.

Algorithm 3 computes a superset of union of arbitrary RCSLMADs

---

Inputs: A group $\{L_1, L_2, .., L_n\}$ of RCSLMADs defined over domain D.
Outputs: Set M, size S of set M and function F for mapping CPU memory locations to GPU memory locations.

1: Compute $m_1 = min(min(L_1(V), V \epsilon D), min(L_2(V), V \epsilon D), .., min(L_n(V), V \epsilon D))$.
2: Compute $m_2 = max(max(L_1(V), V \epsilon D), max(L_2(V), V \epsilon D), .., max(L_n(V), V \epsilon D))$.
3: Compute $M = \{m | m_1 \leq m \leq m_2, m \epsilon Z\}$.
4: Compute $S = m_2 - m_1$.
5: Construct $F(m) = m - m_1$ and $R = \{m | 0 \leq m \leq m_2 - m_1, m \epsilon Z\}$.
6: **return** M, S, and F.

While the algorithms 1, 2 and 3 have only been described for RCSLMADs, they are actually applicable on a slightly wider class of LMADs. For convenience, assume that the ordering function is the identity function and consider the condition $p_k > p_{k+1} - 1 + \sum_{j=k+1}^{d} p_j * (u_j - 1)$ imposed on RCSLMADs. If the condition is instead loosened to $p_k > \sum_{j=k+1}^{d} p_j * (u_j - 1)$, then the algorithms 1, 2 and 3 are still valid (and can be verified by substituting the loosened condition in the corresponding proofs).

5.3 More efficient solutions in specific cases

In some cases, it is possible to derive more efficient solutions that transfer less data than the general algorithm presented earlier. Computing and reasoning with unions of arbitrary RCSLMADs is non-trivial. Consider a group of n RCSLMADs. Let p_{jk} be the stride in the j-th RCSLMAD in k-th dimension. Instead of attempting to compute the union of RCSLMADs in arbitrary cases, this section is limited to the case $p_{t_1 k} = p_{t_2 k}, 1 \leq t_1, t_2 \leq n$, *i.e.* all RCSLMADs have the same stride in any given dimension k. Such cases occur when a programmer is accessing multiple array locations in the loop body with fixed distance between the array accesses. All the RCSLMADs must have the same ordering function because all the RCSLMADs share strides and are defined over the same domain. Without loss of generality, assume that the ordering function is identity throughout this section.

One example of the types of problems being studied in this section is as follows:

$$L_1 = 0 + 20 * i + 3 * j \quad (5.23)$$

$$L_2 = 21 + 20 * i + 3 * j \quad (5.24)$$

$$0 \leq i < 5 \quad (5.25)$$

$$0 \leq j < 5 \tag{5.26}$$

In the example, the two RCSLMADs share the same strides but have different bases.

5.3.1 Representation of union

A suitable representation needs to be chosen to represent the union of RCSLMADs. The representation chosen influences how accurately the union can be computed. As in any other compiler analysis, a tradeoff needs to be made between simplicity and accuracy. One potential choice for representing unions is a single RCSLMAD. However the set of RCSLMADs is not closed over the union operation, *i.e.* the union of two RCSLMADs cannot always be represented exactly using a single RCSLMAD. Instead, I define a new type of set designed to represent a collection of interleaved RCSLMADs with common strides. For brevity, I term such groups as ERL for Extended Restricted CSLMADs since they extend RCSLMADs to multiple bases. Formally an ERL E is defined as a set over a d-dimensional domain D as follows:

Definition 9. *Let there be n RCSLMADs $L_1, L_2, ..., L_n$ defined over the same d-dimensional domain D with the following constraints:*

1. *Let p_{t_1k} and p_{t_2k} represents the k-th strides of L_{t_1} and L_{t_2} respectively. Then the following condition must be satisfied:*

$$\forall(1 \leq t_1, t_2 \leq n, 1 \leq k \leq d)\, p_{t_1k} = p_{t_2k} \tag{5.27}$$

2. *Let b_x be the base of the RCSLMAD L_x. Then*

$$\forall(1 \leq x, y \leq n)\, b_x - b_y < p_d \tag{5.28}$$

Given the above two constraints, ERL E is defined as the union of the RCSLMADs L_1 to L_n. For d-dimensions and n RCSLMADs, an ERL has $2 * d + n$ parameters including d upper bounds, d strides and n bases.

One example of the type of sets represented by ERLs is seen by the following example:

$$L_1 = 0 + 20 * i + 3 * j \tag{5.29}$$

$$L_2 = 1 + 20 * i + 3 * j \tag{5.30}$$

$$0 \leq i < 5 \tag{5.31}$$

$$0 \leq j < 5 \tag{5.32}$$

As can be verified, the above RCSLMADs can be represented by an ERL since all conditions are satisfied. The 2 RCSLMADs are interleaved, *i.e.* they never overlap and have the same

strides. In general, ERL has the useful property that the size of the set being represented can be easily computed. To find an algorithm for finding the size of such a group E, I first prove a theorem.

Theorem 3. *Let there be two d-dimensional RCSLMADs L_1 and L_2 defined over the same domain D with equal strides p_k in each dimension k. If the bases b_1 and b_2 of the two RCSLMADs are such that $0 \leq b_2 - b_1 < p_d$, then L_1 and L_2 are completely disjoint.*

Proof. I prove the theorem by contradiction. Let there be two d-dimensional tuples $V_1 \epsilon D$ and $V_2 \epsilon D$ such that $L_1(V_1) = L_2(V_2)$. For the sake of contradiction, I assume that $V_1 \neq V_2$. Let m be the smallest dimension in which V_1 and V_2 differ. The rest of the proof is similar to the proof given in theorem 1. First, consider the case where $V_1(m) < V_2(m)$. We substitute the values of V_1 and V_2 in the definitions of L_1 and L_2. Then we apply the condition $p_m > p_1 - 1 + \sum_{k=m+1}^{d} p_k * (u_k - 1)$. Following the logic of proof of theorem 1, using the condition it can be proven that $L_1(V_1) < L_2(V_2)$. Therefore, we arrive at a contradiction and the theorem is proved for this case. Second, consider the case $V_1(m) > V_2(m)$. In this case it can be proven that $L_1(V_1) > L_2(V_2)$. Again we arrive at a contradiction and the theorem is proved. □

Given the previous theorem, computing the number of distinct elements in an d-dimensional ERL E composed of n RCSLMADs is straightforward.

Theorem 4. *Let E be a d-dimensional ERL composed of n RCSLMADs $L_1, L_2, .., L_n$ with bases $b_1, b_2, ..., b_n$ respectively. Then the number of distinct elements of E is given by $q * \prod_{k=1}^{d} u_k$ where q is the number of distinct elements in the list $\{b_1, b_2, b_3, .., b_n\}$.*

Proof. The theorem is a corollary of the previous theorem. □

5.3.2 One-dimensional RCSLMADs with common stride

Consider two one-dimensional RCSLMADs L_1, L_2 having the same stride m and defined over the domain D.

$$L_1 = b_1 + m * i \tag{5.33}$$

$$L_2 = b_2 + m * i \tag{5.34}$$

$$0 \leq i < u \tag{5.35}$$

Without loss of generality assume that $b_1 \leq b_2$. As was shown earlier, RCSLMADs can be visualized as simple-strided accesses into an imaginary C array. If there is more than one RCSLMAD, we can attempt to place both of them in the same imaginary C array. In this

case, assume we had a C array starting at some arbitrary positive integer $b_0 \leq b_1$.[1] The RCSLMADs can be rewritten as:

$$L_1 = b_0 + m * (i + t_1) + r_1 \tag{5.36}$$

$$t_1 = \lfloor (b_1 - b_0)/m \rfloor \tag{5.37}$$

$$r_1 = (b_1 - b_0)\%m \tag{5.38}$$

$$L_2 = b_0 + m * (i + t_2) + r_2 \tag{5.39}$$

$$t_2 = \lfloor (b_2 - b_0)/m \rfloor \tag{5.40}$$

$$r_2 = (b_2 - b_0)\%m \tag{5.41}$$

$$0 \leq i < u \tag{5.42}$$

Further rewriting:

$$L_1 = b_0 + m * j + r_1 \tag{5.43}$$

$$t_1 \leq j < u + t_1 \tag{5.44}$$

$$L_2 = b_0 + m * k + r_2 \tag{5.45}$$

$$t_2 \leq k < u + t_2 \tag{5.46}$$

We observe that the value t_1 will be less than or equal to t_2 because $b_1 \leq b_2$. We can construct an RCSLMAD L_3 that is a superset of L_1 by increasing the domain of L_1.

$$L_3 = (b_0 + r_1) + m * j \tag{5.47}$$

$$0 \leq j < u + t_2 \tag{5.48}$$

Similarly we can construct an RCSLMAD L_4 as a superset of L_2.

$$L_4 = (b_0 + r_2) + m * j \tag{5.49}$$

$$0 \leq j < u + t_2 \tag{5.50}$$

L_3 and L_4 are defined over the same domain $D' = \{j | 0 \leq j < u + t_2\}$. Further, the difference in bases of L_3 and L_4 is equal to $|r_1 - r_2|$ and $|r_1 - r_2| < m$ because $r_1 < m$ and $r_2 < m$. The union of L_3 and L_4 is therefore a one-dimensional ERL E defined over domain D' with stride m and with bases $b_0 + r_1$ and $b_0 + r_2$. Since we have not yet chosen b_0, we should attempt to choose a b_0 such that the set D' is the smallest. The integer $b_0 = b_1$ gives

[1] The new base b_0 will later be chosen to minimize the domain of an RCSLMAD that combines the memory references of L_1 and L_2.

the smallest set E because any value of b_0 smaller than b_1 will result in either the same or a larger domain D'.

The methodology is generalized by construction in algorithm 4 for n RCSLMADs with $b_0 = b_1$. The idea is that if the number of non-distinct bases in the construct ERL E is q, and if the upper bound of the domain of E is $t_n + u$, then the total number of elements to be transferred to the GPU is $(t_n+u)*q$ since t_n+u elements need to be transferred for each non-distinct RCSLMAD present in the ERL E. Thus on the GPU, we can allocate space equal to $(t_n + u) * q$ elements. For address mapping, consider the interval b_0 to $b_0 + m - 1$ on the system RAM. Out of this interval, q elements accessed by the q non-overlapping RCSLMADs are transferred. This pattern of q out of every m elements is repeated. Therefore if elements not accessed by the ERL were to be discarded, then the pattern is equivalent to q interleaved accesses with stride q.

Algorithm 4 computes the approximate union of n one-dimensional RCSLMADs with the same stride.

Inputs: n one-dimensional RCSLMADs $\{L_1, L_2, .., L_n\}$ with stride m and bases $\{b_1, b_2, .., b_n\}$ such that $b_1 \leq b_2 ... \leq b_n$ defined over domain $D = \{i, 0 \leq i < u\}$.
Outputs: ERL E and a list of n expressions representing the transformed address of each RCSLMAD.

1: **for** each RCSLMAD L_j **do**
2: Compute the term $t_j = \lfloor (b_j - b_1)/m \rfloor$ and $r_j = (b_j - b_1)\%m$.
3: **end for**
4: Construct a list $R = \{b_1 + r_1, b_1 + r_2, .., b_n + r_n\}$.
5: Remove any duplicates from R. Let the number of elements remaining in R be q.
6: Sort R in-place.
7: Construct a one-dimensional ERL E with stride m, bases $R1$ and domain $D' = \{j, 0 \leq j \leq t_n + u\}$.
8: Construct an empty list I.
9: **for** each RCSLMAD L_j **do**
10: Find x such that $R(x) = b_j + r_j$.
11: Append the expression (represented as an AST or other compiler IR) $q * (i + t_j) + x$ to I where i is the original loop counter for which the analysis is being conducted.
12: **end for**
13: **return** E and I.

5.3.3 Multidimensional RCSLMADs with common strides

Consider n RCSLMADs $\{L_1, L_2, .., L_n\}$ with the common strides $P = \{p_1, p_2, .., p_d\}$. Let the RCSLMADs be defined over a d dimensional domain $D = \{(i_1, i_2, .., i_d) \mid 0 \leq i_j < u_j, 1 \leq j \leq d\}$ and with the bases $B = \{b_1, b_2, .., b_n\}$. I assume that there are no duplicate RCSLMADs. The objective is to find a d-dimensional ERL E with n-components and with strides P such that E is the approximate union of the given RCSLMADs. Let the base of E be $B' = \{b'_1, b'_2, ..., b'_n\}$ and the domain of E be $D' = \{(i_1, i_2, .., i_d) | 0 \leq i_j < u'_j, 1 \leq j \leq d\}$.

To find the ERL E, the idea is to derive a set of linear integer constraints and then

solve for unknown parameters of E using an integer programming solver. The equations are derived as follows:

1. E can be constrained such that the m-th component of E must be a superset of RCSLMAD L_m. The idea is to assume that for each $V \epsilon D$, there exists a point $V' \epsilon D'$ such that $V' = V + \{t_{m1}, t_{m2}, ..., t_{mn}\}$ and that $E(m)(V') = L_m(V)$. The values t_{mk} are assumed to be unknown integer constants. The following equations can be stated:

$$b_m + \sum_{k=1}^{d} p_k * i_k = b'_m + \sum_{k=1}^{d} p_k * (i_k + t_{mk}) \quad (5.51)$$

$$u'_1 \geq u_k + t_{m1} \quad (5.52)$$

$$u'_2 \geq u_2 + t_{m2} \quad (5.53)$$

$$\vdots$$

$$u'_d \geq u_d + t_{md} \quad (5.54)$$

$$t_{m1} \geq 0 \quad (5.55)$$

$$t_{m2} \geq 0 \quad (5.56)$$

$$\vdots$$

$$t_{mk} \geq 0 \quad (5.57)$$

   If suitable integer values are found for the unknowns t_{mk} and u'_k that satisfy the above constraints, then E is a superset of the union of the RCSLMADs by construction.

2. Each component of E must be an RCSLMAD and must therefore satisfy constraints relating the upper bounds and strides.

   For each integer k such that $2 \leq k \leq d$

$$p'_k \geq p_d + \sum_{j=k+1}^{d} (p_j * (u'_j - 1)) \quad (5.58)$$

3. From the definition of an ERL, the difference between any pair of bases (b'_x, b'_y should be less than stride p_d in the last dimension.

   For $\{(x, y) | 1 \leq x \leq n, 1 \leq y \leq n, x \neq y\}$

$$b_x - b_y \leq p_d - 1 \quad (5.59)$$

   These inequalities forms a set of $(n-1)^2$ constraints necessary for ensuring that E is an ERL.

Thus a total of n equality constraints and $n * d + (d-1) + (n-1)^2$ inequality constraints can be derived for a total of $n + d + n * d$ unknowns. The unknown variables are u'_k, b'_m

and t_{mk} where $1 \leq k \leq d$ and $1 \leq m \leq n$. Ideally we want to minimize the size of E but the size of E is a nonlinear function. To utilize an integer programming solver, I define the following objective function that still attempts to minimize the size of the E. To choose a suitable objective function, the following observations can be made:

1. Let $t_k = max(t_{mk})$. Then $u_k \geq u_k + t_k$. Observing all constraints involving u'_k, if the linear integer solver finds a feasible solution such that $u'_k > u_k + t_k$, then simply reducing the value of u'_k to $t_k + u_k$ will still satisfy all constraints.

2. Let q be the number of distinct entries in B'. Then the size of ERL E is $q * \prod_{k=1}^{d}(u_k + t_k)$ by property of ERLs proved earlier.

Therefore we derive a heuristic objective function that should provide a reasonable approximation to minimizing the number of entries:

$$Minimize \sum_{k=1}^{d} u'_k \tag{5.60}$$

Given the constraints and the objective function, an integer programming solver is utilized to find the value of all unknowns. However, if the number of dimensions or the number of RCSLMADs exceeds a threshold, then the compiler falls back to algorithm 3 because integer programming solvers can take a large amount of time as the number of variables increases.

Consider the following example:

$$L_1 = 0 + 20 * i + 3 * j \tag{5.61}$$

$$L_2 = 21 + 20 * i + 3 * j \tag{5.62}$$

$$0 \leq i < 5 \tag{5.63}$$

$$0 \leq j < 5 \tag{5.64}$$

Then the algorithm will find the following ERL:

$$L'_1 = 0 + 20 * i' + 3 * j' \tag{5.65}$$

$$L'_2 = 1 + 20 * i' + 3 * j' \tag{5.66}$$

$$0 \leq i' < 6 \tag{5.67}$$

$$0 \leq j' < 5 \tag{5.68}$$

If the general form was used in the example instead of the ERL approach, then the number of elements transferred would have been $21 + 20 * 4 + 3 * 4 = 113$. Using the ERL

approach, the number of elements transferred is $6 * 5 * 2 = 60$ elements. The ERL approach can result in significant savings in the number of elements transferred if only a small tile of a large array is accessed in a computation.

If the number of dimensions is greater than one, then a feasible solution does not necessarily exist. The problem is that we are attempting to interpret the n RCSLMADs as n interleaved accesses into a d-dimensional array but such an interpretation does not always exist. If the integer programming solver is called and no feasible solution is found, then the compiler falls back to algorithm 3 for data transfer analysis.

To generate the mapped address of each array access, algorithm 5 is used.

Algorithm 5 computes the mapped address for multidimensional RCSLMAD problems solved using the integer linear programming method

---

Inputs: Specified constants u_k, values t_{mk} and b'_m computed using integer linear programming **Outputs**: A list of n expressions representing the address on the GPU.

1: Construct a vector $R1 = [b'_1, b'_2, ..., b'_n]$.
2: Construct a vector $R2 = R1$.
3: Remove all non-duplicate entries of $R2$.
4: Sort $R2$ in-place.
5: Construct a vector $R3$ such that $R3[i] = x$ where $R2[x] = R1[i]$.
6: Compute $q =$ length of $R2$.
7: **for** each integer k such that $1 \leq k \leq d$ **do**
8: Compute $t_k = max(t_{1k}, t_{2k}, ..., t_{nk})$.
9: **end for**
10: Compute $b_0 = min(R1)$.
11: **for** each RCSLMAD L_m **do**
12: Construct an expression $I_m = (b'_m - b_0) + \sum_{k=1}^{d} q * (i_k + t_{mk}) * \prod_{l=k+1}^{d} (u_l + t_l)$.
13: **end for**
14: **return** list $\{I_1, I_2, ..., I_n\}$.

5.4 Loop tiling for handling large loops

Using previously discussed algorithms 3 and 5, the number of elements to be transferred to the GPU can be computed. However, if the amount of memory to be transferred does not fit on the GPU, then the compiler must attempt to tile the loop. In the implemented compiler, the compiler only considers tiling the parallel loops since tiling parallel loops is always legal. Once an acceptable tile size is found, then the tiles are executed sequentially on the GPU. The execution of the tiles can be pipelined but this possibility was not considered due to time constraints.

In the ideal case, the tiling should be such that the total amount of data transferred between the CPU and GPU should be minimized. However, again the problem is hard and therefore a very basic heuristic was used. The compiler simply reduces the upper bound of

each parallel loop to half its original value. If there are m parallel loops, then 2^m tiles are formed. The compiler then computes the amount of data to be transferred for each tile and if the number of elements is less than the amount of memory available, then the compiler continues tiling the loop. The compiler aborts the attempts to tile the loop if the total number of tiles exceeds 64 and abandons efforts to use the GPU.

5.5 Conclusions

This chapter presented a new heuristic algorithm for array access analysis and for automatically transfering data between the system memory and the GPU memory. The algorithm only handles one class of LMADs but can offer significant space savings on the GPU compared to more naive approaches. This chapter also presented a loop tiling algorithm that can automatically scale parallel loop nests so that the data required for the computation fits in the limited GPU memory. The algorithms have been implemented in jit4GPU that currently only generates code for AMD GPUs, but the algorithms presented in this chapter are equally applicable to any GPGPU system with a separate address space and limited GPU memory.

Chapter 6

Loop transformations for AMD GPUs

To extract the maximum performance out of a chip such as the RV770, extensive code transformations such as loop unrolling and load-coalescing are necessary. Programmers are not expected to do such loop transformations manually for two reasons. First, loop transformations like unrolling reduce programmer productivity and program maintainability. Secondly, low-level details like the memory arrangement of the GPU and information about specific datatypes are not exposed to the programmer. Therefore, it is the responsibility of the compiler to do the necessary transformations. In the implemented compiler framework, jit4GPU is responsible for loop transformations while the AMD CAL IL compiler does low level transformations like instruction scheduling and register allocation.

One important code transformation done by jit4GPU is reduction of the number of memory load instructions (called texturing instructions on the GPU) by coalescing loads into loads of multi-component types such as float2 and float4. Due to the unbalanced ratio of ALU to load unit hardware, the number of load instructions should be reduced so that the load (texturing) unit is not the bottleneck. By default, the compiler uses all the memory resources on the GPU in single-component formats. Single-component formats only allow single-component loads. The compiler tries to identify GPU resources that can be stored in aligned multi-component formats. Resources stored in aligned multi-component formats only allow aligned multi-component loads. Thus, the compiler first analyzes if the loads from a particular resource can always be grouped into aligned multi-component loads. If the grouping is successful, then the compiler uses more efficient formats and reduces the number of load instructions. To find out groups of loads that can be combined, the compiler looks at the addresses accessed in the loop body and attempts to find loads of the form $4 * e + c$ where e is any expression while c is a constant with the value 0,1,2 or 3. If all the accesses from a memory resource match this form, then the memory resource can be stored on the GPU using a aligned four-component format. The compiler then attempts to

identify loads that can be coalesced.

Reducing the number of load instructions is only done by jit4GPU when it can find suitable loads in the loop body to coalesce. Thus, larger loop bodies have more potential for load-coalescing. Before doing load-coalescing transformations, Jit4GPU carries out a loop-unrolling pass. A larger loop body also helps the AMD IL compiler to do better instruction scheduling and register allocation. However, larger loop bodies can cause an increase in register usage per thread that results in a lower number of threads running in parallel on the GPU. Therefore, the compiler first uses an heuristic to determine if the register usage is below a certain threshold. Loop transformations are only done if register usage is below the threshold. Estimating register usage can be time consuming but fortunately need not be done by the JIT compiler. Register usage is estimated by unPython and it passes the information to jit4GPU. If enough registers are available then a loop unroll factor of either four or two is applied. The factor four was chosen because in many cases a factor of four unroll gives good opportunities to coalesce loads into float4, the widest data type available on GPUs. If the loop upper is not divisible by four, then an unroll by two is attempted instead. Inner loops are given higher priority for unrolling.

Under special conditions the jit4GPU also performs loop fusion. Loop fusion is a loop transform where two loops with the same loop bounds and no dependence can be fused into a single loop where the loop bodies of the two loops is concatenated into a single loop body. Thus, loop fusion produces a single larger loop from two smaller loops. However, loop fusion can only be performed when there is no dependence between loops. Further, if there is some code occurring between the two loops, then loop fusion can only be applied if the intervening code can be safely moved to a location before the first loop. I have not implemented dependence analysis in jit4GPU . Therefore, in general jit4GPU cannot perform loop fusion. However, under some circumstances the dependence checking is not required. Let loop L_1 be a parallel loop and let L_2 be a loop contained in the body of L_1. If L_1 is unrolled, then a copy L_2' of L_2 is formed. There cannot be any dependence between different iterations of L_1 because L_1 is parallel. Therefore, there cannot be any dependence between L_2 and L_2' because L_2 and L_2' are in the body of two different iterations of L_1. Therefore, the compiler can easily fuse the loops L_2 and L_2'.

Jit4GPU performs all the loop transformations described before generating AMD IL code. Jit4GPU performs loop unrolling and limited loop fusion followed by load-coalescing as a means to reduce the number of texture unit instructions executed. The overview of the transformations performed by jit4GPU is summarized in algorithm 6.

Algorithm 6 performs loop transformations for GPU code

```
1: for all loops with no child loops do
2:    Mark as candidate for unrolling
3: end for
4: for all parallel loops do
5:    Mark as suitable for unrolling if children loops have no children.
6: end for
7: while register usage is below threshold do
8:    Pick the innermost unrolling candidate loop available and unroll.
9:    Update register usage estimate of parent loops.
10:   Mark the unrolled loop as unsuitable for unrolling.
11: end while
12: Fuse as many loops as possible.
13: Perform load-coalescing transformations.
```

Chapter 7

Experimental evaluation

This chapter presents the performance of the code generated for OpenMP and on several highly parallel kernel benchmarks. Performance was evaluated against the generated serial C++ code. For each kernel benchmark, the execution time of the generated serial code was compared against the total execution time of generated OpenMP and GPU versions. Performance on the GPU was measured with and without loop optimizations. For GPU performance, four numbers are reported:

1. GPU total execution time with GPU-specific loop optimizations enabled in jit4GPU. The time reported includes data transfers and JIT compilation overheads.

2. GPU execution time only with loop optimizations enabled. Only the time taken by the GPU to execute the GPU binary are included and does not include data transfers and JIT compilation times.

3. GPU total execution time without any GPU-specific loop optimizations.

4. GPU execution only without any GPU-specific loop optimizations enabled.

The kernels chosen for performance evaluation were matrix multiplication, CP benchmark from the Parboil benchmark suite, Black Scholes option pricing, 5-point stencil code and RPES kernel from the Parboil benchmark suite. The objective of the performance evaluation is to study the performance gains when GPU code generation is enabled against the performance of an OpenMP version of each benchmark. Therefore, benchmarks were all chosen to be highly parallel kernels which are suitable for execution on the GPU. Among the chosen benchmarks, the memory access pattern in four of the benchmarks is describable by RCSLMADs and the compiler is able to generate GPU code. In benchmark RPES, loops are triangular with indirect memory references and the compiler was unable to generate GPU code.

A more comprehensive study of the percentage of cases in which the compiler is able to generate GPU code will require a standard benchmark suite implemented in Python

with suitable GPU code annotations. Implementing such a benchmark suite is left as future work. However, as mentioned in chapter 5, jit4GPU is only able to generate GPU code when the memory access pattern is describable by RCSLMADs. Therefore, some examples for which jit4GPU will be unable to generate a GPU version includes FFT, conjugate gradient algorithms and matrix solvers with triangular loops.

The important results from the experiments are:

1. Using a GPU delivered upto 100 times speedup over generated OpenMP code running on the CPU.

2. Loop optimizations performed by jit4GPU deliver upto four times performance improvement on the GPU.

3. Benchmarks that perform very little computation per data item accessed are not suitable for the GPU because the data transfer overhead is larger than the computation time in such cases.

All experiments were done using a AMD Phenom X4 9550 (2.2ghz quad-core) paired with a Radeon HD 4870 and 4gb of RAM. Frequency scaling was kept disabled on the CPU. The operating system was Ubuntu 8.10 with Linux kernel version 2.6.27-7 and GCC version 4.3.2. For compiling C++ code, the optimization flag `-O3` was passed to the CPU. When compiling for OpenMP, the flag `-fopenmp` was also passed. Several other flags were also tested but they produced no notable performance changes and were exluded in the results presented here.

Each experiment was repeated 5 times and the minimum, maximum and mean execution times are presented.

7.1 Matrix multiplication

Matrix multiplication was implemeneted for 32-bit and 64-bit floating point matrices. A very simple implementation of matrix multiplication was done in Python and the outer two loops were marked as parallel loops for GPU execution. Performance was studied against the matrix sizes. For comparing performance of the generated GPU code against the CPU, the performance results of the generated OpenMP code as well as performance results from ATLAS library are included. ATLAS implements a tiled matrix multiplication algorithm and also autotunes to best fit the system CPU at the time of installation. ATLAS is a high performance library with many years of development effort. By comparision, the Python implementation was a straightforward implementation of matrix multiplication written in under ten lines of Python code. From this Python source, the compiler was able to generate

GPU code that performed twice as fast as ATLAS and over 100 times faster than generated OpenMP code.

The execution times for are presented in Table 7.1 for 32-bit floating point and in Table 7.3 for 64-bit floating point. Each case was repeated 5 times and the minimum, maximum and mean execution time are presented. The speedups obtained using the GPU over ATLAS are presented in Table 7.2 and Table 7.4. The speedups are calculated using the minimum execution time.

Table 7.1: Execution time for matrix multiplication benchmark for 32-bit floating point (seconds)

Problem Size		1024	2048	4096	6120
OpenMP 4 threads	min	7.64	68.35	874.42	1213.7
	max	7.82	68.83	877.98	1219.4
	mean	7.75	68.67	875.64	1215.14
ATLAS BLAS	min	0.125	0.684	5.12	17.87
	max	0.126	0.687	5.14	18.05
	mean	0.125	0.685	5.13	17.91
GPU Total (Opt)	min	0.084	0.39	3.00	8.19
	max	0.087	0.41	3.00	8.34
	mean	0.085	0.40	3.00	8.27
GPU Only (Opt)	min	0.025	0.277	1.78	6.86
	max	0.027	0.288	1.88	7.01
	mean	0.026	0.282	1.82	6.93
GPU Total (No opt)	min	0.159	0.99	8.68	28.16
	max	0.161	1.04	8.84	28.8
	mean	0.160	1.01	8.73	28.39
GPU Only (No opt)	min	0.108	0.90	7.46	26.88
	max	0.110	0.95	7.61	27.52
	mean	0.109	0.916	7.5	27.11

Table 7.2: Speedups for matrix multiplication using GPU for 32-bit floating point over ATLAS

Problem Size	Speedup No Opt	Speedup Opt
1024	0.786	1.488
2048	0.69	1.75
4096	0.58	1.7
6120	0.634	2.18

7.2 CP benchmark

CP benchmark from the Parboil suite was implemented in Python. The benchmark is a simple nested loop and the top two loops are annotated to be parallel for GPU execution. The CP benchmark is representative of some computations done in molecular dynamics. This benchmark computes the columbic potential at each point in a planar grid. The computation of potential at each grid point is independant of other grid points and therefore

Table 7.3: Execution time for matrix multiplication benchmark for 64-bit floating point (seconds)

Problem Size		1024	2048	3072	4096
OpenMP 4 threads	min	8.55	109.5	274.85	1406.82
	max	8.68	110.4	275.8	1409.5
	mean	8.59	109.8	275.1	1408.0
ATLAS BLAS	min	0.244	1.34	4.38	11.62
	max	0.247	1.36	4.47	11.70
	mean	0.245	1.35	4.41	11.64
GPU Total (Opt)	min	0.147	0.679	3.71	5.34
	max	0.16	0.695	3.82	5.53
	mean	0.15	0.685	3.78	5.39
GPU Only (Opt)	min	0.078	0.513	2.34	3.84
	max	0.09	0.543	2.44	4.06
	mean	0.081	0.523	2.38	3.91
GPU Total (No opt)	min	0.227	1.42	6.21	11.26
	max	0.24	1.53	6.52	11.73
	mean	0.232	1.47	6.33	11.4
GPU Only (No opt)	min	0.164	1.249	4.87	10.13
	max	0.177	1.33	5.182	10.66
	mean	0.168	1.27	4.96	10.31

Table 7.4: Speedups for matrix multiplication using GPU for 64-bit floating point over ATLAS

Problem Size	Speedup (No Opt)	Speedup (Opt)
1024	1.07	1.659
2048	0.95	1.97
3072	0.7	1.18
4096	1.03	2.17

CP is a highly parallel kernel suitable for GPUs. At each grid point, the potential is computed as a summation of potentials due to atoms randomly distributed in 3D space.

Jit4GPU was able to achieve a speedup of more than 50 times over generated OpenMP codes. Execution times for the benchmark are presented in Table 7.5. For each case, the minimum, maximum and mean execution time are presented. The speedups are reported in Table 7.6. The speedups are calculated based on minimum execution time.

Table 7.5: Execution time for CP benchmark (seconds)

Problem size		128	256	512	1024
Serial Time	min	9.01	36.03	137.99	552.26
	max	9.04	36.48	138.76	555.35
	mean	9.11	36.11	137.19	553.08
OpenMP 1 thread	min	8.78	34.56	138.01	552.71
	max	8.89	35.09	139.52	554.00
	mean	8.82	34.73	138.62	553.28
OpenMP 4 threads	min	2.234	8.692	34.57	138.06
	max	2.297	8.982	35.29	140.06
	mean	2.24	8.925	35.77	138.96
GPU Total (Opt)	min	0.084	0.1979	0.672	2.58
	max	0.085	0.214	0.713	2.85
	mean	0.084	0.204	0.692	2.65
GPU Only (Opt)	min	0.05	0.163	0.635	2.524
	max	0.05	0.182	0.687	2.802
	mean	0.05	0.169	0.655	2.61
GPU Total (No opt)	min	0.161	0.544	2.056	8.129
	max	0.163	0.57	2.235	8.35
	mean	0.162	0.55	2.096	8.4
GPU Only (No opt)	min	0.13	0.51	2.022	8.088
	max	0.15	0.53	2.20	8.30
	mean	0.14	0.52	2.06	8.16

Table 7.6: Speedups for CP using GPU over OpenMP

Problem Size	Speedup (No Opt)	Speedup (Opt)
128	13.81	26.59
256	15.98	43.92
512	16.81	51.44
1024	16.98	53.51

7.3 Black Scholes option pricing

Black Scholes formula is used for option pricing in computational finance. Black Scholes formula is a simple scalar computation that computes two scalar results from five scalar inputs. The benchmark computes multiple Black Scholes options in parallel. The Python implementation is derived from a Brook+ implementation provided by AMD with the Stream SDK. The performance of Black Scholes on the GPU is entirely dominated by the data transfer

time and the execution time on the GPU is negligible. A speedup of about six times is achieved over generated OpenMP code. The detailed results are presented in Table 7.7 and the speedups are listed in Table 7.8. For each case, the minimum, maximum and mean execution times are presented and the speedups are calculated bsaed on minimum execution time.

The benchmark involves functions such as exponential and natural logarithms and these functions were compiled to corresponding hardware instructions on the GPU. However, a significant difference in the calculated results was observed between CPU and GPU. Accurate software implementations of square root, natural log and other functions on the GPU is future work for this thesis.

Table 7.7: Execution time for Black-Scholes benchmark (seconds)

Problem size		512	1024	2048
Serial Time	min	0.36	1.438	5.71
	max	0.363	1.46	5.77
	mean	0.361	1.44	5.73
OpenMP 1 thread	min	0.35	1.46	5.6
	max	0.352	1.478	5.67
	mean	0.351	1.465	5.62
OpenMP 4 threads	min	0.1633	0.394	1.4
	max	0.1633	0.394	1.4
	mean	0.1633	0.394	1.4
GPU Total (Opt)	min	0.0829	0.106	0.224
	max	0.084	0.108	0.227
	mean	0.083	0.1065	0.225
GPU Only (Opt)	min	0.0004	0.0009	0.0020
	max	0.0004	0.0009	0.0020
	mean	0.0004	0.0009	0.0020
GPU Total (No Opt)	min	0.079	0.111	0.22
	max	0.079	0.112	0.23
	mean	0.079	0.111	0.224
GPU Only (No Opt)	min	0.0004	0.0009	0.0025
	max	0.0004	0.0009	0.0025
	mean	0.0004	0.0009	0.0025

Table 7.8: Speedups for Black-Scholes benchmark using GPU over OpenMP

Problem Size	Speedup (No Opt)	Speedup (Opt)
512	2.067	1.96
1024	3.54	3.71
2048	6.36	6.25

7.4 5-point stencil

5-point stencil benchmark computes an out-of-place 5-point stencil over a 2-dimensional matrix. For each element in the input matrix, the code computes a weighted average of

the element and its 4 neighbors and writes the result to the corresponding element in a matrix of the same dimensions. The kernel is highly parallel as each element is processed independently but the amount of computation per point is very small. The compiler was able to successfully carry out the array access analysis and was able to generate the GPU code for this benchmark. However, the data transfer overhead considerably outweighed the computation time savings of the GPU. Therefore, the benchmark ran slower when using a GPU compared to OpenMP. The results are reported in Table 7.9. For each case, the minimum, maximum and mean execution times are presented.

Table 7.9: Execution time for 5-point stencil benchmark (milliseconds)

Problem size		1024	2048	3072	4096
Serial Time	min	10.8	43.3	97.5	175
	max	10.9	43.7	98.2	176.6
	mean	10.8	43.4	97.7	175.8
OpenMP 1 thread	min	10.1	46	70	150
	max	10.26	47	71.24	152
	mean	10.17	46.3	70.4	151
OpenMP 4 threads	min	5	24	47.3	58.1
	max	5	24	47.8	58.5
	mean	5	24	47.5	58.2
GPU Total (Opt)	min	65	98.8	123.1	1010
	max	65.8	99.6	124.9	1016
	mean	65.3	99.1	123.7	1014
GPU Only (Opt)	min	0.39	0.89	2.8	25
	max	0.39	0.89	2.8	26
	mean	0.39	0.89	2.8	25.5
GPU Total (No Opt)	min	35.3	63.1	112.1	1000.3
	max	35.6	63.7	112.7	1006
	mean	35.4	63.25	112.3	1002
GPU Only (No Opt)	min	0.59	2.0	4.2	50
	max	0.59	2.0	4.2	52
	mean	0.59	2.0	4.2	51

7.5 RPES benchmark

RPES benchmark is a Python adaption of the benchmark from Parboil suite. The benchmark involves indirect memory loads and triangular loops. UnPython determined that a GPU version cannot be generated and therefore did not generate calls to the JIT compiler. Therefore, the performance did not change when the GPU was enabled because the GPU was never used and the JIT compiler was not called. The compiler only generated an OpenMP version of the benchmark. This benchmark illustrates that the programmer can safely add GPU parallel annotations without any fear of errors in the case of compiler limitations.

The benchmark was only tested with default parameters provided by the benchmark

suite. The serial version of the benchmark completed in 259 seconds and the parallel version completed in 62 seconds.

7.6 Remarks

The section evaluated the implemented compiler over several kernel benchmarks. The generated GPU code delivered over 2 orders of magnitude performance over OpenMP code for some benchmarks. Jit4GPU was also able to deliver better performance using the GPU than highly tuned CPU libraries such as ATLAS. The loop optimizations done by the compiler were also found to be effective and provided considerable speedups over unoptimized GPU code. The performance of some benchmarks, such as 5-point stencil, were found to be bound by the data transfer overhead making it unsuitable for execution on the GPU.

Chapter 8

Related Work

The most important previous research that relates to the main contributions of this thesis can be divided into three cathegories: the conversion of code originally written to be interpreted in Python to compiled code to run in a CPU, the analysis of memory accesses via arrays and the transfer of relevant memory regions between the CPU main memory and the GPU memory, and the construction of compilers for general-purpose computations that execute in GPUs. This chapter will discuss each category of related work in the subsequent sections.

8.1 Compiling Python for CPUs

UnPython is not the first compiler attempting to compile Python to C/C++ or other lower level language. Various compilers have been developed that compile Python or a closely related dialect. Pyrex [5] is a compiler that compiles the Pyrex language to C/C++. Pyrex language is a statically typed language with many syntactic similarities to Python. Pyrex also has special syntax for interaction with C libraries. The major advantage of Pyrex over unPython is that it allows very easy interaction with C libraries and it has a very simple and efficient way of generating Python bindings for C libraries. However Pyrex language is not Python and will not run on the Python interpreter. Further, Pyrex currently has no support for NumPy arrays and does not support any parallel programming features.

Cython [2] is a fork of Pyrex and has added many features over Pyrex. Cython's syntax is much closer to Python and does support efficient access to NumPy arrays. However, Cython does not support any parallel programming features.

Several compilers have attempted to compile pure Python to lower-level languages using type inference. Shedskin [13] is a compiler that compiles pure, unannotated Python to C++. Shedskin uses type inference and the type inference requires certain implicit typing restrictions on the Python code being compiled. Shedskin uses a much more advanced type inferencing algorithm than unPython and is designed to be a whole-program compiler.

However Shedskin does not support efficiently accessing NumPy arrays and does not have any parallel programming features.

Another compiler that compiles pure Python to lower level language is the RPython [11] compiler present in the PyPy [4] project. PyPy is an attempt to write the Python interpreter in Python itself. To achieve this goal, the interpreter is written in a subset of Python called RPython that includes various implicit type-based restrictions. A compiler then compiles RPython to various backends such as C. Various other backends are also in progress including an LLVM backend and a Java backend. Therefore, in the future PyPy can compile RPython to C, LLVM and Java. However, PyPy does not have support for efficient access of NumPy arrays. To gain support for NumPy arrays, PyPy will require implementation of NumPy array in RPython.

In a very different direction, a new project called Unladen Swallow [7] is a branch of the standard Python interpreter with an LLVM-based JIT compiler. The project aims to be a replacement of the standard interpreter and, as such, requires no changes in Python code but may break C-API compatibility with the standard interpreter. One of the future goals of the project is to have a completely thread-safe Python interpreter that will allow true multithreaded applications in Python. The project is looking at replacing the garbage-collection (GC) mechanism of the standard Python interpreter with an alternate thread-safe scheme because the current reference count based GC is one of the major obstacles to thread safety in the interpreter. Unladen Swallow is still a work in progress and if it succeeds in building a high-performance thread-safe interpreter and offering a thread-safe C API, then it has the potential to be a really good complement to unPython. The current Python interpreter has severely handicapped the parallel programming capability offered by unPython.

8.2 Array access analysis

The primary objective of array access analysis in this thesis is to find out the memory locations to be transfered between CPU and GPU memories. The array access analysis used is based on the concept of LMADs. LMADs were first defined by Paek *et al.* [17]. They proposed LMADs as an efficient way to capture very accurate array access information. They also discussed several operations on LMADs such as a set intersection of LMADs. However they did not describe a method to compute the union of LMADs. In this thesis, I proposed a method to approximately compute and represent a union of RCSLMADs which are a a subclass of LMADs.

In this thesis, array access analysis is done by the JIT compiler jit4GPU and the concept of LMADs is not used by the ahead-of-time (AOT) compiler unPython. One significant advantage of doing array-access analysis just before executing a loop is that the value of

loop-invariant symbolic constants is known just before executing the loop. AOT compilers do not know the value of symbolic constants and thus may have to deal with a very large number of possible scenarios of memory access patterns. JIT compilers can know the value of the symbolic constants and can do a much more precise array-access analysis. Jit4GPU is not the first compiler to do array-access analysis at runtime using LMADs. Rus *et al.* present a runtime representation of LMADs called RT-LMADs [18],[19]. They were interested in applying array-access analysis for parallelization and hence their implementation focused on computing dependence information. They represented a union of LMADs simply as a list of LMADs. However this representation is not useful for this thesis. In this thesis, dependence information is not required and instead an efficient representation of the union that easily maps to a different address space is needed. The algorithm presented in this thesis for computation of union is a unique contribution of this thesis.

8.3 Compilers for GPGPU

The field of general-purpose computations on GPUs started with disguising general-purpose programs as graphic-shader programs. The real emergence of GPGPU started with languages such as Brook [12]. Brook is an extension of the C programming language. Brook provides a stream datatype and introduces kernel functions. Streams are similar to arrays but represent data to be transferred to the GPU. Kernel functions are functions to be executed by each GPU thread and only operate on streams and scalar values. Streams passed as parameters to the kernel functions were marked as input or output streams. At the time the original implementation of Brook was done, GPUs could not do writes to arbitrary locations. Thus, kernel functions could not write to arbitrary locations. To utilize GPUs for general purpose computation, the programmer had to first copy data into streams and then pass these streams to kernel functions. The Brook compiler compiled the kernel functions into OpenGL GLSL or DirectX 9 HLSL shaders. AMD has extended Brook to create a language called Brook+ to expose writes to arbitrary locations and to introduce several new datatypes, but the programming model remains the same. Apart from Brook, several other stream-based programming languages have been proposed for GPGPU such as StreamIt [21][22], Sh [16] and Stream Virtual Machine [14].

Currently, APIs such as OpenCL [3] and Nvidia CUDA [1] are available to program GPUs. Nvidia CUDA is an extension of C/C++. CUDA exposes the GPU as a Single-Program Multiple-Data (SPMD) machine where the same code is executed by many GPU threads. In CUDA, the GPU also has its own separate address space and the programmer is required to manually transfer the data between the CPU and GPU address spaces. CUDA extends C by providing `device` function types that are functions to be executed on the GPU. The code in a device function represents the code to be executed by each GPU

thread. Device functions are written in a subset of C and also can have added syntax for utilizing GPU hardware features such as on-chip shared memory. The calls to device functions must also provide a grid of threads that specifies as a parameter the number and the configuration of GPU threads to be launched. Therefore, overall CUDA provides almost complete control of the GPU to the programmer but is a fairly low-level programming model. If a code is required to be portable between CPU and GPU, then writing code in CUDA can require significant code duplication because two versions (normal C function and a device function) of the same code may need to be implemented. OpenCL is based on concepts similar to CUDA. Compared to CUDA, this thesis provides a simpler alternative to programming GPUs by transferring the burden of generating and optimizing GPU code from the programmer to the compiler. But the compiler framework presented in this thesis is not always successful in generating GPU code and the JIT compilation utilized can also represent a significant performance overhead.

One limitation of APIs such as CUDA and OpenCL is that each performance-critical function is implemented twice: once as a CPU-specific function and then as a GPU-specific device function. One attempt to reduce this duplication is offered by MCUDA [20]. MCUDA compiles CUDA code to multicore x86 code and therefore allows CUDA code to be executed on multicore CPUs. MCUDA is based on the premise that programmers should only write the CUDA version of the performance-critical function and the MCUDA automatically generates a CPU version. Thus, MCUDA is based on a philosophy exactly oppposite of this thesis. This thesis provides the programmer an illusion of executing the program on a symmetric multiprocessor machine while automatically utilizing GPU. MCUDA provides an illusion of executing on a GPU but automatically generates a CPU version if a GPU is not present.

The work closest to this thesis is the work by Lee *et al.* on compiling OpenMP code to Nvidia CUDA [15]. They describe a compiler that can automatically generate Nvidia CUDA code from C/C++ programs with OpenMP annotations. There are three primary differences between this thesis and the work done by Lee *et al.* First, for analyzing which data to transfer to the GPU, they can only deal with arrays but not with pointers. They simply transfer the entire array without analyzing what specific memory locations within the array are accessed in the loop. The approach in this thesis is based upon analyzing the memory locations accessed and this approach, if used within a C/C++ setting, can be applied to both pointers and arrays. Second, they cannot tile or break the loop if the data does not fit into the GPU. If one of the arrays referenced in the loop is too big to fit on the GPU, they cannot generate GPU code whereas this thesis can tile the loop in some cases and can thus divide the computation into smaller pieces. Finally, they describe loop optimizations that are suitable for Nvidia GPU architecture while this thesis is concerned

with AMD architecture. Their loop optimizations are completely different than the loop optimizations described in this thesis because the architectures are very different.

Another work that extends OpenMP for GPGPU programming is EXOCHI framework by Wang *et al* [23]. EXOCHI is an extension of OpenMP for C/C++ for heterogenous systems. Their implementation is for a multicore x86 CPU and for an integrated Intel graphics chipset. Unlike the discrete GPUs considered in this thesis, such as the Radeon 4870, Intel graphics chipsets are integrated into the northbridge of the CPU and do not sit on a PCIe bus. These integrated chips also do not have a separate onboard memory and can access the system RAM. EXOCHI therefore does not need to copy data and instead only needs to remap the memory address translation table from CPU to the GPU. The address translation remapping is handled by EXOCHI's runtime. To program the GPU, EXOCHI requires the programmer to write GPU code but does not require the programmer to do any data transfers because data transfers are not necessary. Instead, the GPU code can directly access any data in system RAM thereby simplifying the programming. EXOCHI is only suitable for systems where both the CPU and the accelerator (such as the GPU) can access the system RAM directly and where the address translation table can be simply remapped. Therefore EXOCHI is not applicable to current generation discrete GPUs.

8.4 Conclusions

This thesis describes the first Python compiler to provide simple parallel programming support for numerical applications. The implented compiler is also one of the first to automatically map a shared-memory parallel programming model to a GPGPU system. This thesis describes a new algorithm to automatically transfer relevant data between a CPU and a GPU. The implemeneted compiler provides a programming model that is simpler to program than current GPGPU APIs such as CUDA and that relies on compiler analysis and optimization to automatically generate GPU code.

Chapter 9

Conclusions

This thesis introduced a new programming model for more efficient programming of numerical programs in Python for execution on GPUs. The thesis also described the design and implementation of a compiling system to convert numerical Python programs annotated with type and parallel loop annotations to multi-cores and GPUs. In this new programming model, a programmer writes code for a simple shared-memory abstraction and the compiler automatically converts the program to use a GPU as an accelerator. The program remains portable to multicores and GPUs with no code changes.

The compiler system consists of unPython, an ahead-of-time compiler and jit4GPU, a just-in-time compiler. Jit4GPU implements a new algorithm to analyze the regions of memory accessed by an array reference in a loop nest. The algorithm is restricted to a class of affine accesses termed as RCSLMADs. Jit4GPU automatically transfers the required data for the computation between the CPU and the GPU based on the results of the array access algorithms. Jit4GPU is not a general-purpose JIT compiler and only works on numerical programs represented as parallel loop nests with array accesses representable as RCSLMADs. Jit4GPU generates GPU code from a typed abstract-syntax-tree (AST) representation of the Python program generated by unPython. Jit4GPU also performs several loop optimizations such as loop unrolling and memory load coalescing.

The performance evaluation used several numerical kernels. On some kernels, Jit4GPU performs over 100 times faster than OpenMP code generated by unPython. Jit4GPU also delivers better performance than some highly tuned CPU libraries, such as ATLAS, without requiring the programmer to do any optimizations such as unrolling or tiling in the original Python source. Compilers, such as Jit4GPU, allow the programmer to easily utilize the computational power of modern GPUs for general purpose computation.

Bibliography

[1] Nvidia CUDA (2009-09-30). `http://www.nvidia.com/cuda`.

[2] Cython:C-Extensions for Python (2009-09-30). `http://www.cython.org`.

[3] OpenCL - The open standard for parallel programming of heterogeneous systems (2009-09-30). `http://www.khronos.org/opencl/`.

[4] PyPy project (2009-09-30). `http://codespeak.net/pypy/dist/pypy/doc/`.

[5] Pyrex - a Language for Writing Python Extension Modules (2009-09-30). `http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/`.

[6] Python/C API Reference Manual (2009-09-30). `http://www.python.org/doc/2.5/api/api.html`.

[7] Unladen Swallow: A faster implementation of Python (2009-09-30). `http://code.google.com/p/unladen-swallow`.

[8] AMD. *ATI Stream Computing User Guide*, February 2009.

[9] AMD. *Compute Abstraction Layer (CAL) Intermediate Language (IL) Reference Guide*, February 2009.

[10] AMD. *R700-Family Instruction Set Architecture*, March 2009.

[11] D. Ancona, M. Ancona, A Cuni, and N. Matsakis. RPython: A Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *OOPSLA 2007 Proceedings and Companion, DLS'07: Proceedings of the 2007 Symposium on Dynamic Languages*, pages 53–64. ACM, 2007.

[12] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[13] Mark Dufour. Shed Skin - An experimental (restricted) Python to C++ compiler (2009-09-30). `http://code.google.com/p/shedskin/`.

[14] Francois Labonte, Peter Mattson, William Thies, Ian Buck, Christos Kozyrakis, and Mark Horowitz. The Stream Virtual Machine. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 267–277. IEEE Computer Society, 2004.

[15] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 101–110. ACM, 2009.

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

[17] Yunheung Paek, Jay Hoeflinger, and David Padua. Efficient and precise array access analysis. *ACM Trans. Program. Lang. Syst.*, 24(1):65–109, 2002.

[18] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. Run-time Assisted Interprocedural Analysis of Memory Access Patterns. Technical report, Department of Computer Science, Texas A&M University, 2001.

[19] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. *International Journal of Parallel Programming*, 31(4):251–283, 2003.

[20] John Stratton, Sam Stone, and Wen mei Hwu. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs. In *21st Annual Workshop on Languages and Compilers for Parallel Computing (LCPC)*, July 2008.

[21] William Thies, Michael Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the International Conference of Compiler Construction*, 2002.

[22] A. Udupa, R. Govindarajan, and M.J Thazhuthaveetil. Software Pipelined Execution of Stream Programs on GPUs. In *International Symposium on Code Generation and Optimization (CGO)*, pages 200–209, 2009.

[23] Perry H. Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 156–166. ACM, 2007.