A Compilation Framework to Execute Python Numerical-Intensive Applications in a Hybrid CPU-GPU Execution Environment

Rahul Garg

Department of Computer Science, McGill University, Montreal, QC, Canada

José Nelson Amaral

Department of Computing Science, University of Alberta, Edmonton, AB, Canada.

January 16, 2012

Abstract

A new compilation framework enables the execution of numerical-intensive applications in an execution environment that is formed by multi-core Central Processing Units (CPUs) and Graphics Processing Units (GPUs). A critical innovation is the use of a variation of Linear Memory Access Descriptors (LMADs) to analyze loop nests and determine automatically which memory locations must be transferred between the CPU address space and the GPU address space. In this programming model, the application is written in a combination of Python and NumPy, a rich numerical extension for Python. Unobtrusive light annotation is introduced to identify the type of function parameters and return values, and to indicate which loop nests should be parallelized and executed in the GPU. The new compilation system is a combination of an ahead-of-time compiler, unPython, to transform Python/NumPy code into a combination of C++ and an intermediate representation, and a just-in-time compiler, jit4GPU, that converts the intermediate representation into the AMD CAL interface. The experimental evaluation using a collection of well-known benchmarks indicates that there is very significant performance advantages to execute important loops of numerical applications in GPUs.

1 Introduction

The execution of a program in an environment that includes a CPU and a GPU requires: (*i*) identification of important loops that are profitable to execute in the GPU; (*ii*) determination of the data that needs to be transferred for the execution of each loop in the GPU; (*iii*) mapping of the relevant portions of the CPU memory address space to the GPU address space, and vice versa, for the data that needs to be transferred back; and (*iv*) careful transformation of the loop code to be executed in the GPU to adapt it to the constraints in the GPU design. The compilation work flow introduced in this paper assumes that the program has been annotated to indicate which loops are parallel and, among the parallel loops, which ones may be profitable for execution in the GPU.

The program is written in Python with the numerical computation executed by NumPy. Python is a dynamically typed, object-oriented language, which was designed to be interpreted. NumPy is an extension module that defines a fast multi-dimensional array class to enable Python to operate with arrays. The compilation requires that functions to be compiled be annotated with the type of their arguments. These annotations are non-intrusive, thus allowing the same program to be executed by the Python interpreter — a very important feature for development and debugging.[1]

Dynamically typed scripting languages, such as Python, Matlab and Ruby, are attractive for programmers with limited programming-language experience because it is easy to quickly write practically useful programs in such languages. Often the development of a project starts in such a language because of the low entrance barrier and ease of programming. However, when the project grows, the performance of an interpreted environment becomes an issue. Thus, compilation of such languages should be of great interest.

This paper presents a new programming model and compiler framework to increase the productivity of developers that use Python/NumPy and want to extract good performance from machines that feature both GPUs and CPUs. This new compilation framework automatically identifies the data that is necessary to transfer to the GPU for the execution of a loop. At the moment the compiler can handle an important, commonly used, subset of loops with affine array index expressions.

[1]The type of the parameters passed to the interpreter are the same as the types that appears in the type annotations processed by the compiler.

The main contributions of this paper include:

- A complete compilation system — ahead-of-time compiler, just-in-time compiler, and runtime system — that leverages the computational power of GPUs for numerical computations in applications developed in Python (Section 2).
- An adaptation of loop tiling to split the loop into smaller tiles if the data required for the loop execution does not fit in the limited GPU memory (Section 3) .
- An algorithm to determine a reasonable set of memory locations that must be transferred to/from the GPU to enable the execution of each loop, and a mapping from the CPU address space to the GPU address space based on the memory regions that are transferred (Section 4).
- A performance evaluation of the system in AMD GPUs that demonstrates that the generated code running on GPUs outperforms OpenMP generated code by up to 50 times, and either outperforms, or is competitive with, CPU code from the Basic Linear Algebra Subprograms (BLAS) library, which are highly optimized (Section 5).

2 Programming and Compilation Model

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. The glue code exposes the C/C++ functions and data to the Python interpreter as Python functions and objects, and facilitates transferring data back and forth between C/C++ and Python. The glue code is written manually or generated using tools like SWIG or using wrapper APIs like Boost.Python.

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 flexible. 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.

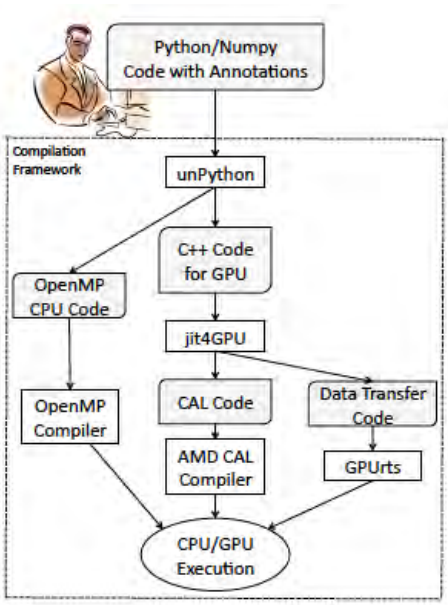


Figure 1: Compilation framework to support the new programming model.

This paper introduces a new compiler system for automatic generation of C++ code for the performance-critical Python sections. Instead of rewriting code in C++, the programmer annotates the Python code with type declarations and parallel loop annotations.

2.1 New compilation work flow

Figure 1 shows the compiler framework that supports this new programming model. UnPython automatically generates the C++ code as well as 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 before compilation. To take advantage of GPU acceleration, the programmer simply annotates functions to be executed in the GPU as GPU accelerated and the compiler handles everything else.

UnPython can only deal with a subset of Python and requires type annotations. However only the small portion of the Python code that will be compiled to C++ needs to conform to these restrictions. The rest of the application remains as is. Furthermore, the annotations are designed

to be compliant with the Python grammar and to be 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.

2.2 Extensions Introduced

In the new programming model presented in this paper 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 type 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. For instance, in the example in Figure 2 the first type is for the parameter `n` and the second type is for the return value. The types of `sum` and `i` are automatically inferred to be `int64`. Local variable type declarations could force them to be `int32`.

```
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 2: Example of decorator used as a type declarator.

The programming model introduces parallel loop annotations. In Python only the `range` and `xrange` iterators are allowed in loops. We defined a special iterator, `prange`. To the Python interpreter, `prange` is identical to `xrange`. However, unPython treats `prange` as a special parallel-loop annotation with the fork-join semantics of OpenMP parallel loops. Furthermore, all variables declared outside the loop are assumed to be shared by all threads. The only synchronization supported is the implicit join point at the end of a parallel loop nest. Parallel loops can be nested

but, in this model, a join point only exists at the end of the outermost parallel loop. UnPython generates OpenMP code for parallel loops that will be executed in multi-core CPUs.[2]

When a decorator is used to specify that a function should be executed in the GPU, the compiler is responsible for generating GPU code as well as for managing data transfers between CPU and GPU automatically. The decorator is ignored in the absence of a GPU or when the compiler fails to generate a GPU version of the code. Thus the source code is portable to multi-core platforms that do not have a GPU.

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 run-time code execution, higher-order functions, generators, meta-classes and special methods such as `setitem`. UnPython supports 16-, 32- and 64-bit integers and 32- and 64-bit floating point numbers, but it does not support arbitrary-precision arithmetic. 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.

```
1  @unpython.gpu
2  @unpython.type('ndarray[double,2]',
3                 'ndarray[double,2]',
4                 'ndarray[double,2]',
5                 None)
6  def matrix_mult(a,b,c):
7      m = shape(a)[0]
8      n = shape(b)[1]
9      p = shape(a)[1]
10     for i in prange(m):
11         for j in prange(n):
12             sum = 0.0
13             for k in xrange(p):
14                 sum += a[i,k]*b[k,j]
15             c[i,j] = sum
```

Figure 3: Type annotations for a matrix multiplication program.

[2]UnPython generates sequential code for any parallel lo op that contains a function call.

2.3 Programming Model and Compilation Framework

The compilation framework is divided into three distinct pieces: unPython, an ahead-of-time compiler; jit4GPU (read "Jit-for-GPU"), a JIT compiler exclusively for the generation of GPU code; and GPUrts, a run-time support system to manage GPU resources. The input to unPython is an annotated subset of Python and it generates a C++ and OpenMP code extension module. C++ and OpenMP were chosen as backends for the CPU code because they are portable and they allow this new system to benefit from wisdom accumulated in C++ compilers to generate code for specific CPU platforms. UnPython focuses on compiling performance-critical portions of scientific applications.

The code-generation process for GPUs is more involved. The compiler and run-time system are responsible for automatically managing all GPU code generation as well as data transfers. Therefore, the code generator needs information about strides of NumPy arrays and dependence information. This information cannot be obtained by unPython and hence a purely ahead-of-time approach is infeasible for generating GPU code from Python.

UnPython cannot perform any global analysis because unPython only sees type-annotated portions of the application program. Further, information about strides (and thus the memory layout of the NumPy arrays) is not available to unPython because of the dynamic nature of the language.

The solution was to write a just-in-time compiler, jit4GPU, that is linked with the application. Jit4GPU discovers array layouts before compiling code thus avoiding the limitations faced by unPython. For GPUs, unPython both passes an intermediate program representation to jit4GPU *and* generates a fall-back CPU execution path. At run time jit4GPU determines if GPU code can, and should, be generated. If either execution in the GPU would not be profitable or the jit4GPU fails to generate GPU code, then execution proceeds with the CPU code generated by unPython. Jit4GPU only operates on arrays of numeric types, numeric scalar types, and loops.

For GPGPU programming, jit4GPU produces AMD CAL IL code and utilizes the CAL API to compile and execute the code. The AMD CAL IL compiler is a JIT compiler that compiles proprietary AMD IL code to GPU ISA. There are several reasons for choosing CAL API over

the vendor-agnostic OpenCL. First, at the time that development started, no cross-platform open APIs for GPGPU were available — AMD's OpenCL implementation for GPUs was released later. Second, the GPU code is generated in the JIT compiler and need a fast back-end compiler. Code generated by the AMD OpenCL compiler can be ten times slower than the code from the CAL IL compiler. Finally, the OpenCL compiler had yet to provide support for accessing texture resources in OpenCL.

3 Generating Code for Execution in a GPU

Jit4GPU converts an intermediate code representation, which was generated by unPython from source NumPy code, into a low-level compute-oriented Application Program Interface (API) called Compute Abstraction Layer (CAL). The CAL API enables programming through a pseudo-assembly called the AMD Intermediate Language (IL). Programs written in this IL are compiled to GPU binary code at run time. The AMD IL is similar to the assembly language of Shader Model 4.0 (introduced in DirectX 10) [16], but it has additional features such as double-precision instructions and a global buffer for scatter.

3.1 Highlights of the GPU Architecture

The AMD Radeon 5850 graphics card consists of an array of 18 SIMD units, 18 texture units, a dynamic thread dispatcher, 4 memory controllers and a DMA unit. Radeon 5850 is based upon the AMD Cypress chip. Cypress has 20 SIMD units but 2 of those have been kept disabled in the 5850 to distinguish it from the higher-priced Radeon 5870. Each SIMD unit in Radeon 5850 contains 16 thread processors and each thread processor in turn contains 5 stream processors for a total of 1440 stream processors. Each of the stream processors can execute at most one scalar multiply-and-ADD (MAD) operation in one cycle resulting in a peak performance of 2880 single-precision floating-point operations (FPO) per cycle. Alternatively, the card can perform 576 double-precision FPOs per cycle. Each SIMD array is aligned with a texture unit for memory loads. Each texture unit can execute four physical address computations per cycle for a total of only 72 address computations per cycle for the entire chip. This reveals a considerable asymmetry between ALU and address-

computation capability that is characteristic of most modern GPUs. Each SIMD unit has an associated 8kB L1 texture cache and 32kB Local Data Share (LDS), also called local memory in OpenCL terminology. The texture cache is a read-only cache and can be thought of as a load buffer instead of a true cache. Each SIMD unit has a 256kB register file. This large register file suggests that data should be brought in to registers whenever possible. Each of the four memory controllers have 128kB L2 cache giving a total of 512kB L2 cache for the chip.

The AMD Radeon 5850 GPU runs at a clock rate of 725 *MHz*. Its peak performance is 2000 single-precision Gflop/second (Gflops) or 400 double-precision Gflops. The Radeon 5850 utilizes Graphics Double Data Rate version 5 (GDDR5) memory and has a peak memory bandwidth of 128Gbps.

3.2 AMD Compute Abstraction Layer

From an AMD IL code the CAL compiler generates a module. The IL or ISA function to be executed on the GPU is called a *kernel*. To execute a kernel, the programmer passes a handle to the kernel and the size of the thread-block to a specific function in the CAL API. The beginning of execution of a kernel on the GPU is termed as *launching* a kernel.

AMD IL provides two types of kernels: pixel shaders and compute shaders. Jit4GPU only uses compute shaders. AMD IL compute shaders provide a Single-Program Multiple-Data (SPMD) programming model. Threads are organized into thread groups and thread groups are further arranged into a grid. Each thread knows its absolute thread id as well as its id within the group. The hardware groups threads into batches of 64 threads called wavefronts. A wavefront executes on a Single-Instruction Multiple-Data (SIMD) unit. Each SIMD can execute many wavefronts in parallel depending upon availability of resources such as registers. A thread dispatcher dynamically sends Wavefronts to various SIMD units.

In the CAL API, memory is allocated as a 1D or 2D resource with a maximum size of 16384x16384 elements where an element can be of up to 16 bytes. Henceforth we refer to data types as follows: *float* is a 32-bit floating-point value, *double* is a 64-bit floating-point value, *float2* is a 64-bit segment containing two floats, *float4* is a 128-bit segment containing four floats, and

double2 is a 128-bit segment containing two doubles.

The declaration of a resource specifies the element size for all indexing operations into the resource. Resources can be created on the GPU itself, in a special reserved section of the system RAM that is accessible by the GPU, or from pre-allocated system memory (provided that certain alignment constraints are met). Declarations must specify the location of the resource.

Each resource to be used in the kernel has to be bound, before a kernel is launched, either to a predefined read-only input array (i0 to i7) or to a read-write unordered access view (UAV) resource. Other types of resources are available but are not discussed here.

The GPU can access the buffers allocated in system RAM using DMA accesses within a kernel, or through the CAL API for copying data between system RAM and the GPU. The data copy between resources in system RAM and resources on GPU can be done in parallel with the kernels because the DMA unit is independent of ALUs. Further, most calls of the CAL API (including kernel launch and initiation of DMA transfers) are asynchronous and thus provide the ability to setup pipelines.[3]

3.3 Basic Code Generation for GPUs

The GPU and the CPU are in separate address spaces. Kernel functions executing on the GPU can access resources located in either the on-board GPU RAM or the PCIe buffer space in system RAM. The compiler and run-time system are responsible for ensuring that the data required for computation by a GPU is first copied from system RAM to a GPU-addressable location. While the GPU can read from the PCIe buffers in system memory, reading data within a kernel from PCIe buffer is inefficient because it may require transferring the same data multiple times in small chunks over the PCIe bus. Therefore, data is copied to the GPU on-board RAM before starting computation on the GPU. In general, the amount of on-board memory on a GPU (typically 1GB in current mainstream consumer GPUs) is smaller than the memory requirements of a kernel. Thus loop tiling is often necessary. Finally, memory addresses need to be converted to the GPU address space.

[3] Jit4GPU does not yet take advantage of this parallelism.

Jit4GPU handles all GPU code generation, including analysis for data transfers. Jit4GPU can only generate GPU code for a class of memory access patterns. Loop nests that contain accesses that do not fit this class are executed in the CPU.

3.4 Loop optimizations

While each SIMD unit can complete 80 ALU operations per cycle, the texture unit (TEX) can only perform 4 address computations per cycle. To mitigate this imbalance, jit4GPU performs three transformations: loop unroll-and-jam, coalescing loads into vector types such as float2 or float4, and partial-redundant load elimination. Jit4GPU performs loop unroll-and-jam to expose more opportunities for coalescing and redundant load elimination. Unrolling loops also gives the AMD's CAL IL compiler a larger scope for instruction scheduling. Unroll-and-jam is the same as tiling a loop if more outer loops are unrolled. Tiling can potentially improve the data locality and can improve the texture cache and register usage but the compiler does not take locality into account explicitly while performing the transformation.

Unroll-and-jam is performed in phases. The analysis starts by considering the unroll of the innermost loop and moves outwards in each phase. A loop is considered a candidate for unrolling if it is either the innermost loop, or if it is a parallel loop with constant bounds and all its children also have constant bounds. The loop bound is also required to be a multiple of 2. If a loop satisfies these conditions, then the analysis will consider the register usage. If the estimated register usage after unrolling the loop nest under consideration will be above a set threshold (32 4-wide registers in the current implementation), then jit4GPU does not attempt unrolling of the current loop nest. The number of registers used is restricted by the compiler because the number of GPU threads that can run concurrently is limited by the availability of registers to be distributed amongst the threads. Running many GPU threads is necessary to keep the execution units of the GPU busy and to hide latency of operations such as memory loads.

After the loops are unrolled, the compiler marks groups of array accesses in the loop body that can be coalesced into a vector type. If all address expressions from the same GPU resource can be written as a sum of a multiple of 4 or 2 and a constant offset, then the GPU resource can be

converted into a multi-component resource (such as a resource of float4 components). After unroll-and-jam and coalescing into vector types is complete, the compiler performs partial-redundant load elimination in the loop body. The loads occurring in the loop body that are determined to refer to the same element of the resource are collapsed to a single load.

3.5 Back-end optimizations

Back-end optimizations are done by the AMD CAL compiler. The AMD CAL compiler generates GPU ISA code from the given CAL IL. While the CAL IL is itself register based, the registers in the IL do not correspond to physical registers. The CAL IL compiler does physical register allocation, instruction scheduling, including VLIW packing, dead-code elimination and many other optimizations. While the exact details of all the optimizations performed by the compiler are not public, in our experience the AMD compiler usually does a good job for most back-end optimizations.

4 Automatic Data Transfers and Address Mapping

The proposed shared-memory programming model delivers the abstraction of a single address space shared by the CPU and the GPU. However, the GPU operates in a separate address space from the CPU. Thus, all data needed for computations performed in the GPU must be present in the GPU address space before the computation can be started. The new compilation framework presented here attempts to automatically identify the data to be transferred between the physical CPU and GPU address spaces. The mapping from the single address space in the programming model to the two address spaces in the architecture can be decomposed into two related problems:

- **Data transfer problem:** the compiler must identify the set of memory locations to be copied. This set is a superset of the memory locations actually accessed.
- **Address mapping problem:** for each array access in a computation performed in the GPU, the compiler must replace the address computation done for the array access in the system memory space with an address computation expression in the GPU address space.

4.1 Overview of the approach

In the numerical programs targeted by this compilation framework most memory accesses inside a loop are through array references. The set of memory locations accessed through such a reference usually can be described fully through the memory address expression for one iteration as a function of the loop counters and the iteration domain. Jit4GPU only transforms loops in which array indexes are an affine function of the loop counters.

If these conditions are met, then for each array access the compiler computes a new affine function of the loop counters representing the assigned memory address on the GPU. The compiler also computes the size of the set of memory references accessed by the array access on the CPU. The new function is chosen such that there is a one-to-one correspondence of CPU memory addresses represented by the original expression and the computed GPU memory addresses. If a loop nest contains array accesses for which a new set of affine coefficients cannot be computed, then GPU code is not generated for that nest.

4.2 Representation of array references

To set up an automatic data transfer for a kernel, the compiler needs the loop bounds and the layout of the NumPy arrays referenced. The mapping of array index to memory locations in a NumPy array is more complex than in standard programming languages such as C. Thus, it is best to represent the memory access pattern directly instead of separately representing the subscript expression and strides of arrays. The array-access analysis in jit4GPU is based on the concept of Linear Memory Access Descriptors (LMADs) introduced by Paek *et al.* [25]. Jit4GPU uses a restricted form of LMADs that we call Constant-Stride LMADs or CSLMADs.

Consider an array access occurring inside a loop nest of depth d. Let M be the set of memory locations accessed by the given array access. Let the vector of loop counters be $\vec{i} = (i_1, i_2, ..., i_d)$ and let D be the iteration domain. Assume that D has been normalized so that the lower bound for all loops is zero and the loop stride is one. The upper bound of a loop is assumed to be an affine function of the loop counters of outer loops. The set M is defined to be a CSLMAD if and only if the memory address represented by the array access is an affine function $f(\vec{i})$ of $\vec{i}$. The set M is

completely specified by the function f and the iteration domain D. If the loop-nest is of depth d, then $\vec{i} = (i_1, i_2, ..., i_d)$. Given that f is affine, it can be expressed as:

$$f(\vec{i}\,) = b + \sum_{k=1}^{d} s_k \times i_k \tag{1}$$

$$M = \{f(\vec{i}\,) | \vec{i}\ \in D\} \tag{2}$$

The constants s_k are the strides of the CSLMAD and the integer constant b is the base of the CSLMAD. Both depend on the memory access pattern represented by the array access. The *dimension* of a CSLMAD is the number of loop variables occurring in the memory address expression represented by the CSLMAD. Let the loop variable i_k have the domain $[0, u_k)$. The span for dimension k is $\sigma_k = s_k \times u_k$. The span represents the memory interval traversed by the CSLMAD when the loop counter i_k traverses the domain $[0, u_k)$ while keeping the other loop variables constant.

As an example of a CSLMAD, consider an array in the C programming language declared as `char` $A[P][80]$ where P is a compile-time constant, and an array reference $A[i + j + 1][2 \times i + 2]$ that occurs inside a loop nest with loop indices i and j. Let the iteration domain D be given by $0 \leq i < 100$ and $0 \leq j < i + 5$. In C semantics, the memory address represented by the array access is given by $f(i, j) = \&A[0][0] + 80 \times (i + j + 1) + (2 \times i + 2) = \&A[0][0] + 82 \times i + 80 \times j + 82$. The memory address is an affine function f of i and j. $M = \{f(i, j) | (i, j) \in D\}$ is the set of memory addresses accessed by the loop. The strides are $s_1 = 82$ and $s_2 = 80$, respectively, and the base of the CSLMAD is $b = \&A[0][0] + 82$.

A CSLMAD represents many array accesses that are commonly used in numerical programming. However, it has limitations. For instance, a CSLMAD cannot represent an array access such as $B[i \times i]$.

To completely represent a CSLMAD, a compiler has to store the strides, the base and the iteration domain D. D only needs to be represented once in the compiler for each loop nest. For each loop nest, jit4GPU maintains two tables: one for the strides and base of the CSLMAD, and the other for the loop upper bounds. CSLMADs are represented as affine functions that map a

vector of loop counters to a single integer (a memory address). Thus, a CSLMAD is completely specified by its strides and the base. Lower-case letters, such as f, are used for CSLMADs. The domain is specified separately and is represented by upper-case letters, such as D. The function $mem(f, D)$ returns the set of memory locations accessed by the CSLMAD f when iterating over the domain D. Thus $mem(f, D)$ gives the range of f for the domain D.

4.3 Array access analysis

Given a table of array accesses T_i and a table of loop upper bounds T_u, jit4GPU computes an heuristic solution to two problems. (1) Let C be the set of memory locations accessed by CSLMADs in the loop nest. Find the set G of memory locations to be transferred to the GPU such that $C \subseteq G$. Compute $|G|$. (2) Generate a new table T_i' for array accesses representing the memory addresses on the GPU.

Consider the problem where a loop nest contains two array references A and B. Each reference will be represented by a CSLMAD. For each CSLMAD A, let θ_A be the interval of memory locations that can potentially be accessed by A. If θ_A and θ_B are disjoint, then the problem of data transfer can be decomposed into two separate problems. In general, a set of N array accesses can be split into smaller groups of array accesses such that the interval of memory locations accessed by one group does not overlap with another group. In jit4GPU this splitting consists on splitting the original tables T_i and T_u into smaller tables.

Jit4GPU creates a memory-reference overlap graph with CSLMADs as vertices. An edge is placed between two vertices if and only if the memory intervals of two CSLMADs overlap. Each connected component in this graph corresponds to a group of possibly overlapping CSLMADs. Each group of CSLMADs is analyzed independently and assigned to a unique GPU memory resource.

For each group of CSLMADs, the objective is to find the set of memory locations potentially accessed by the group. A conservative estimate of this set is to find an interval of memory locations such that it contains the intervals of memory locations accessed by each CSLMAD in the group as a subset. Such an interval can be found simply by taking the minimum of the lower bounds and the maximum of the upper bounds of all intervals in the group. However, such an interval is

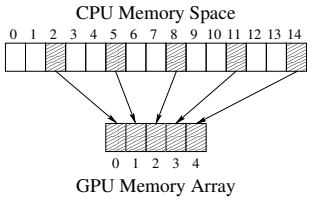


Figure 4: Mapping from CPU address space to a GPU array for a one-dimensional CSLMAD.

a very conservative estimate of the set of memory locations accessed and this general case is not implemented.

In many cases, it is possible to find a more precise estimate of the set of memory locations accessed. Jit4GPU implements the following cases.

4.3.1 One-dimensional CSLMAD

The group consists of a single one-dimensional CSLMAD. The CSLMAD must be of the form $s \times i + b$ where s and b are constant non-negative integers and i is a loop counter that can take any integer value in the interval $[0, u)$. Thus, the domain D of the CSLMAD is $i \in [0, u)$. For a one-dimensional CSLMAD, each possible value of i maps to a unique location in memory. So, the set of memory locations accessed is simply the set containing the u memory locations $b, s+b, 2s+b, ..., (u-1)s+b$. This set can be trivially mapped to a contiguous vector of u unique memory locations on the GPU and it is trivial to compute the number of memory locations accessed and the mapped address. For example, consider the case $f(i) = 3 \times i + 2$ over the domain $[0, 5)$ given in Figure 4, the set of memory locations $2, 5, 8, 11, 14$ in the CPU address space is mapped to the locations $0, 1, 2, 3, 4$ on a GPU memory resource or array.

4.3.2 Multiple one-dimensional CSLMADs with equal strides

Consider the case where a group consists of multiple one-dimensional CSLMADs such that all CSLMADs are defined over the same loop variable i and domain D and have the same stride s. Therefore these CSLMADs differ only in their bases. Then, the sets of memory locations potentially accessed by each of the CSLMADs individually can be computed very easily as described earlier.

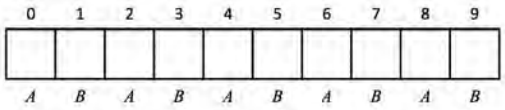


Figure 5: Two CSLMADs A and B with interleaved memory accesses.

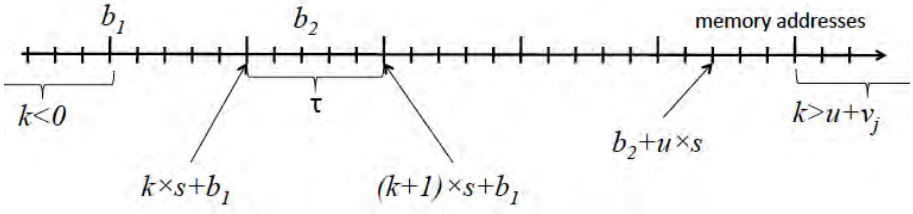


Figure 6: Intervals in which memory accesses from two CSLMADs f_1 and f_j may occur.

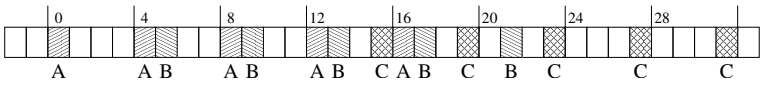
However, the memory locations accessed by the CSLMADs in a group may overlap and therefore cannot be counted separately.

The actual set of memory locations referenced by two CSLMADs A and B may be disjoint even when the intervals θ_A and θ_B overlap. For example, consider the case where CSLMAD A is $2 \times i$ and CSLMAD B is $2 \times i + 1$ over the interval $[0, 5)$. In this case $\theta_A = [0, 8]$ and $\theta_B = [1, 9]$ and they overlap. However, as shown in Figure 5, the actual sets accessed are disjoint.

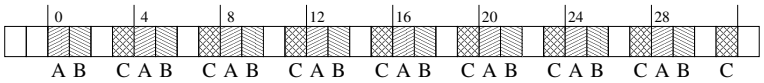
In general, given two CSLMADs A and B with the same stride s and bases b_A and b_B such that $b_B - b_A$ is not a multiple of s, the set of memory locations accessed by A and B is disjoint. Such a group of CSLMADs could be split into smaller groups of disjoint CSLMADs. However, transferring a set of interleaved CSLMADs separately is less efficient than transferring the entire interval as a single contiguous copy. For instance, in the example shown in Figure 5 it is more efficient to do a bulk transfer of the interval $[0, 9]$ to a single GPU resource.

For the general case, let there be n CSLMADs and the j^{th} CSLMAD be given by $f_j(i) = s \times i + b_j$. Without loss of generality, let $b_1 \leq b_2 \leq b_3 \leq \ldots \leq b_n$. Let u be the upper bound of i such that $0 \leq i < u$.

Rewriting the expression for the CSLMADs simplifies the analysis. For all bases $b_j, 2 \leq j \leq n$, the distance $b_j - b_1$ can be computed as two components: a multiple of s and a remainder. Let v_j and r_j be non-negative integers defined as follows:



(a) Original CSLMADs accesses.



(b) Stretched CSLMADs

(c) Compacted CSLMADs

Figure 7: Example that illustrates the analysis and mapping for multiple interleaved one-dimensional CSLMADs.

$$\begin{aligned} v_j &= \left\lfloor \frac{b_j - b_1}{s} \right\rfloor \qquad (3) \\ r_j &= (b_j - b_1) \mod s \end{aligned}$$

(4)

Therefore, replacing equation 3 into equation 4:

$$\begin{aligned} r_j &= b_j - b_1 - s \times v_j \\ b_j - b_1 &= s \times v_j + r_j \qquad (5) \end{aligned}$$

For $j = 1$, $v_1 = 0$ and $r_1 = 0$. Therefore the j^{th} CSLMAD can be expressed as:

$$\begin{aligned} f_j(i) &= s \times i + s \times v_j + r_j + b_1 \\ &= s \times (i + v_j) + r_j + b_1 \qquad (6) \end{aligned}$$

Each CSLMAD can be visualized as a periodically repeated but finite train of unit square waves defined over a particular length of time. The quantity v_j represents the distance between the bases b_j and b_1 in terms of the number of iterations (or number of *periods* in the signal analogy). The quantity r_j is an offset (or *phase* in the signal analogy) representing whether the CSLMADs can potentially overlap. If r_j is non-zero, then the two CSLMADs can be thought of as being out-of-phase and can never reference the same memory location.

Consider two CSLMADs f_1 and f_j as defined above over the domain $i \in [0, u)$. Consider an interval $\tau = [k \times s + b_1, (k + 1) \times s + b_1)$ for any non-negative integer k. Figure 6 illustrates this situation for $s = 5$ and $u = 3$. The interval shown in the figure is for $k = 1$. Each of the CSLMADs f_1 and f_j references at most one memory location in τ because their stride is s. Furthermore, if $k \geq (u + v_j)$ or $k < 0$ then no memory location in τ is referenced by either CSLMAD because there is no overlap between τ and the interval of memory locations accessed by the CSLMADs (see Figure 6). Thus, no more than a total of $2 \times (u + v_j)$ memory locations can be accessed by both CSLMADs combined. If $r_j = 0$ the estimate is refined by recognizing that the two CSLMADs are *in-phase*. In this case the only memory location that might potentially be accessed in τ by either CSLMAD is $k \times s + b_1$ and, therefore, no more than $(u + v_j)$ memory locations are accessed.

The extension of the argument to n one-dimensional CSLMADs with the same stride s needs to determine the maximum number of memory locations that might be accessed in the interval τ by all the CSLMADs combined. Recall that the bases b_1 to b_n are arranged in ascending order. Therefore, given the definition of v_j in equation 3, it must be the case that $v_n = \max(v_1, v_2, \ldots, v_n)$. Some of the values in the multiset $r_1, r_2, ..., r_n$ may be repeated. Let the number of unique values in this multiset be q. Then, no more than q memory locations in τ can be accessed by the n CSLMADs. This result can be proved by contradiction by considering two CSLMADs f_x and f_y such that $r_x = r_y$ and then assuming that they will access different memory locations in the given interval. Given that a maximum of q locations can be accessed in τ and that there are a maximum of $u + v_n$ such intervals that may have accesses, a total of $q \times (u + v_n)$ memory locations can potentially be accessed.

Another way of understanding the process to find the number of memory locations accessed by

multiple overlapping one-dimensional CSLMADs is to consider the example in Figure 7(a). There are three CSLMADs with stride $s = 4$, and bases such that $b_B = b_A + 5$ and $b_C = b_A + 15$. The upper bound for all three CSLMADs is $u = 5$. Thus the accesses fall in the interval from b_A to $b_C + s \times (u - 1)$. This interval is divided into subintervals of s elements each. The pattern of access in an interval that has the most accesses is found. Then, for the purpose of data transfer and mapping, the analysis assumes that this pattern repeats over the entire interval in which accesses may occur as shown in Figure 7(b). This assumption is equivalent to defining new CSLMADs that stretches the original ones by including more accesses than the original ones did.

Next the memory locations accessed by each CSLMAD group must be mapped to GPU addresses. Jit4GPU allocates a vector of $q \times (u + v_n)$ elements on the GPU. Each unique potentially accessed memory location on the CPU must be mapped to a unique memory location on the GPU. Out of every s contiguous locations only q locations may be accessed. The rest can be discarded and the q locations can be stored packed into q contiguous locations on the GPU. Thus, the stride of the CSLMADs changes from s to q.

Before the CSLMADs are compacted in the CPU, the offset r_j of each access is for a stride of s. However compactation changes the stride to a value s', thus all offsets must be recomputed for this new stride. Figure 7(c) illustrates how the compactation affects the offsets. In the implementation of this mapping, jit4GPU constructs a vector of length n with the values $[r_1, r_2, ..., r_n]$, sorts the vector in ascending order and then removes the duplicates resulting in a vector V of length q. The offset of the j-th CSLMAD is given by p_j such that $V[p_j] = r_j$ assuming zero-based indexing. In other words, p_j is the rank of r_j. Thus, the translation of the CSLMAD $f_j(i) = s \times (i + v_j) + r_j + b_1$ to GPU address is given by $g_j(i) = q \times (i + v_j) + p_j + v_0$ where v_0 is the starting address of the allocated vector on the GPU.

4.3.3 RCSLMAD: A special class of multi-dimensional CSLMAD

In general, multidimensional CSLMADs may have complex access patterns. This section deals with a special class of CSLMADs with the property that every access is to a unique memory location. The C-programming-language example in Figure 8 motivates the discussion.

```
1  double A[M][N];
2  double sum = 0.0;
3  for(i=0; i<u1; i++){
4      for(j=0; j<u2; j++){
5          sum += A[i][j];
6      }
7  }
```

Figure 8: A C example to illustrate RCSLMADs

The compiler converts the reference $A[i][j]$ to the CSLMAD $\&A[0][0] + i \times N + j$. However, multidimensional arrays in C are not true multidimensional arrays and are not bounds-checked. Thus, it is perfectly legal for j to be larger than N.

If $u2 \leq N$, then, for a given value of i, all the locations referenced by $A[i][j]$ are in the i-th row. Thus, each point in the iteration space accesses a different array element. However, if $u2 > N$ then there are multiple references to the same location because, for a given value of i, the reference $A[i][j]$ accesses elements in multiple rows.

Given the general form of a two-dimensional CSLMAD $f(i, j) = s_1 \times i + s_2 \times j + c$, each access in this CSLMAD is to a unique memory location if $s_2 \times (u_2 - 1) < s_1$. In other words, the span of j should be contained within the stride of i. To verify this condition the compiler only needs the value of the loop bounds and strides. If the condition is satisfied, the number of memory references is equal to the number of iterations of the loop nest. Jit4GPU always knows the value of loop invariant symbolic constants, such as u_1 and u_2, because each loop is analyzed just before execution.

The example falls in a special class of CSLMADs that can be called Restricted CSLMADs or RCSLMADs. The main motivation for defining RCSLMADs is to ensure a one-to-one mapping from loop iterations to memory locations. This simplifies both the computation of the number of the memory locations accessed as well as the problem of mapping the addresses from CPU to GPU address spaces. Let the loop index dimensions be sorted in descending order of strides i.e. dimensions with larger strides are placed before dimensions with smaller strides. Then RCLMADs are defined as follows.

Definition 1 *Let* $f(i_1, i_2, ..., i_d) = \sum_{k=1}^{d} s_k \times i_k + b$ *be a CSLMAD defined over a domain* $D = (i_1, i_2, ..., i_d)|i_k < u_k$. *If* f *satisfies the condition*

$$\forall k < d, s_k > \sum_{p=k+1}^{d} s_p \times (u_p - 1) \tag{7}$$

then f *is as Restricted CSLMAD (RCSLMAD).*

After stride-based sorting of index dimensions, if the loop indices are ordered in a lexicographic order, then the address of the memory references in the RCSLMAD are ordered in ascending order. The condition for RCSLMADs turns out to be equivalent to the "no overlap test" given by [15]. RCSLMADs have several important properties stated in the following theorems.

Theorem 1 *Let* $f(i_1, i_2, ..., i_d) = \sum_{k=1}^{d} s_k \times i_k + b$ *be a RCSLMAD defined over a domain* $D = (i_1, i_2, ..., i_d)|i_k < u_k$. *For any two iteration vectors* $\vec{i}, \vec{i'} \in D$, $f(\vec{i'}) > f(\vec{i})$ *if* $\vec{i'}$ *is lexicographically larger than* $\vec{i}$.

Proof 1 *Let* m *be the first dimension where* $\vec{i}$ *and* $\vec{i'}$ *differ. Thus,* $i'_m = i_m + t$ *for some positive integer* t. *Let* $\delta_k = i_k - i'_k$ *for all* $0 < k \leq d$. *Therefore:*

$$\begin{aligned} f(\vec{i'}) - f(\vec{i}) &= s_m \times (-\delta_m) + \sum_{k=m+1}^{d} s_k \times (-\delta_k) \\ &= s_m \times t - \sum_{k=m+1}^{d} s_k \times \delta_k \end{aligned} \tag{8}$$

From $\delta_k \leq (u_k - 1)$, *follows:*

$$\sum_{k=m+1}^{d} s_k \times \delta_k \leq \sum_{k=m+1}^{d} s_k \times (u_k - 1) \tag{9}$$

From the defining constraints imposed upon the RCSLMAD:

$$\sum_{k=m+1}^{d} s_k \times (u_k - 1) < s_m \tag{10}$$

Therefore, from inequalities 9 and 10:

$$\sum_{k=m+1}^{d} s_k \times \delta_k < s_m \tag{11}$$

Negating both sides and adding $s_m \times t$ *to both sides:*

$$s_m \times t - \sum_{k=m+1}^{d} s_k \times \delta_k \quad > \quad s_m \times t - s_m \tag{12}$$

But $s_m \times t - s_m \geq 0$. *Therefore, from equation 8 and inequality 12,*

$$f(\vec{i'}) - f(\vec{i}) \quad > \quad 0 \tag{13}$$

Thus the theorem is proved.

Corollary 1 *Let* $f(i_1, i_2, ..., i_d) = \sum_{k=1}^{d} s_k \times i_k + b$ *be a RCSLMAD defined over a domain* $D = (i_1, i_2, ..., i_d) | i_k < u_k$. *All the memory locations accessed by the RCSLMAD are unique.*

From the properties of the RCSLMAD it is obvious that the number of memory locations accessed by an RCSLMAD defined over a domain D is equal to $|D|$.

4.3.4 Multiple CSLMADs with equal strides and different bases

A group containing multiple CSLMADs where all the CSLMADs have the same stride for each dimension often arises in stencil-type computations. The solution to the problem of determining the set of memory locations addressed is presented under two special cases.

First case: *all RCSLMADs are interleaved but definitely access disjoint sets of memory locations* and *the differences between any two bases is smaller than the smallest stride.*

Theorem 2 *Consider* n *RCSLMADs of* d *dimensions with bases* $b_1, b_2, \ldots, b_n$. *For all RCSLMADs in the group, the strides in each dimension are* $s_1, s_2, \ldots, s_d$. *Without loss of generality,* $b_1 < b_2 < b_3 < \ldots < b_n$. *The RCSLMADs are defined over a domain* $D = i_1, i_2, \ldots, i_d | i_k < u_k$. *Let the RCSLMADs satisfy the constraints:*

$$\forall k < d, \qquad s_k > s_d - 1 + \sum_{p=k+1}^{d} s_p \times (u_p - 1) \tag{14}$$

$$\forall m > 1, \qquad b_m - b_1 < s_d \tag{15}$$

Then, the number of memory locations referenced by this group of RCSLMADs is $n \times |D|$.

Proof 2 *Each of the RCSLMADs satisfies the constraints for definition of a RCSLMAD. Therefore, the number of memory locations referenced by each RCSLMAD is* $|D|$. *Moreover, no two RCSLMADs in such a group access the same memory location. This can be proved by contradiction. The assumption for the proof is:*

$$f_p(\vec{i}) = f_q(\vec{i'}) \tag{16}$$

for $\vec{i}, \vec{i'} \in D$ *and* $p < q$. *Three cases must be considered:*

1. $\vec{i}$ *is lexicographically less than* $\vec{i'}$. *However, from theorem 1,* $f_p(\vec{i}) < f_p(\vec{i'})$. *Furthermore,* $f_p(\vec{i'}) < f_q(\vec{i'})$ *because* $b_p < b_q$. *Therefore,* $f_p(\vec{i}) < f_p(\vec{i'})$ *which contradicts the assumption in the proof.*

2. $\vec{i}$ *is lexicographically larger than* $\vec{i'}$ *and the first dimension where they differ is* d *with* $i_d = i'_d + t$ *for some positive integer* t.

$$\begin{aligned} f_p(\vec{i}) - f_q(\vec{i'}) = &\quad b_p - b_q + s_d \times (i_d - i'_d) \\ = &\quad b_p - b_q + s_d \times t \end{aligned} \tag{17}$$

From equations 16 and 17,

$$b_q - b_p = s_d \times t \tag{18}$$

However, from the initial constraints, $b_q - b_p < s_d$. *Therefore,* $b_q - b_p < s_d \times t$ *which is a*

contradiction for this case.

3. *$\vec{i}$ is lexicographically larger than $\vec{i'}$ and the first dimension where they differ is $m < d$ with $i_m = i'_m + t$ for some positive integer t.*

$$f_p(\vec{i}) - f_q(\vec{i'}) = b_p - b_q + s_m \times t + \sum_{k=m+1}^{d} s_k \times (i_k - i'_k) \quad (19)$$

From equations 16 and 19,

$$b_q - b_p + \sum_{k=m+1} s_k \times (i'_k - i_k) = s_m \times t \quad (20)$$

The upper bound on $b_q - b_p$ is $s_d - 1$ and the upper bound on the value of $\sum_{k=m+1} s_k \times (i'_k - i_k)$ is $\sum_{k=m+1} s_k \times (i'_k - i_k)$. Therefore, the upper bound on the LHS of equation 20 is $s_d - 1 + \sum_{k=m+1} s_k \times (i'_k - i_k)$. From the constraint in inequality 14, this upper bound is less than s_m. Therefore $s_m > s_m \times t$ which is a contradiction for this case.

Therefore, no two RCSLMADs in such a group can access the same memory location. Thus, the number of memory locations accessed is $n \times |D|$.

Constraint 14 is tighter than the constraint 7 used in the definition of RCSLMADs. The tighter constraint is necessary to ensure that each RCSLMAD accesses memory locations that are distinct from the locations accessed by other RCSLMADs in the group. For instance, consider two RCSLMADs $f_1(i,j) = 16 \times i + 5 \times j$ and $f_2(i,j) = 16 \times i + 5 \times j + 1$ defined over $D = \{0 \leq i < 8, 0 \leq j < 4\}$. In this case, the RCSLMADs do not obey the constraint 14 because for any i, $f_1(i+1, 0) = f_2(i, 3)$ and thus f_1 and f_2 access overlapping memory locations. The tighter constraint ensures that there are at least u_d intervals of length s_d contained within the stride s_{d-1}. Thus the tighter constraint is necessary to ensure proper interleaving of the accesses.

Second case: *a group G of RCSLMADs obeys constraint 14 but not constraint 15.* Thus, the difference between two bases is not necessarily smaller than the stride. Two RCSLMADs in

the group have similar memory access patterns but the patterns are offset from each other by an arbitrary integer value. The RCSLMADs can therefore potentially overlap. For this case a heuristic solution is implemented. The idea is to find a group G' of n or less CSLMADs satisfying the constraints 14 and 15 with the same strides as G but defined over a possibly larger domain than G such that G' accesses a superset of memory locations accessed by G. For every member f of group G, the idea is to construct a member f' in G' that accesses a superset of memory locations accessed by f. If such a group G' can be successfully constructed, then $|G'|$ can be found using Theorem 2 and can be used as a conservative approximation of $|G|$. This is analogous to the stretching of CSLMADs shown in Figure 7.

Such a solution is constructed by increasing the domain of the RCSLMAD. There are two possibilities: (i) keep the base the same; or (ii) reduce the base of the RCSLMAD. The domain cannot grow unboundedly in dimensions 2 to d because the upper bounds of i_2 to i_d are restricted by the defining conditions on strides of RCSLMADs. If the base is reduced to a value b', the domain must grow sufficiently to include the original RCSLMAD in the new RCSLMAD. Moreover, the difference $b - b'$ must be re-writable as a sum of multiples of strides to ensure that the shifted RCSLMAD does overlap with the original RCSLMAD.

Construction of the new group G' is done by defining a set of constraints stating that the m^{th} member of G' should be a superset of the m^{th} member of G and then adding the constraints defined in inequalities 14 and 15 for G' which contain unknown parameters. This linear constraint system is solved for a feasible solution for the unknown parameters. If a feasible solution is found, then the G' is successfully constructed. If a feasible solution is not found, then the heuristic returns no solution in this case and the jit4GPU exits passing the control back to the CPU fall-back path.

The number of distinct members of G' is not known at the start of the algorithm. Instead, n suitable RCSLMADs are sought, some of which can be duplicates and can be removed later. The bases of the CSLMADs in G' must be found. Let $b' < min(b_1, b_2, \ldots, b_n)$, b' is a positive integer for which the algorithm seeks a solution. Then, a rewriting of the m^{th} RCSLMAD in G, of the following form, is sought:

$$f_m(\vec{i}) = \sum_{k=1}^{d} s_k \times (i_k + t_{mk}) + r_m + b' \tag{21}$$

$$0 \leq r_m < s_d \tag{22}$$

$$\forall k, 1 \leq k \leq d \quad , \quad 0 \leq t_{mk} \tag{23}$$

$$\forall k, 1 \leq k \leq d \quad , \quad 0 \leq i_k < u_k \tag{24}$$

Values t_{mk} and r_m are also unknowns and will be found later. The above set can be transformed by introducing variables $i'_k = i_k + t_{mk}$.

$$f_m(\vec{i}) = \sum_{k=1}^{d} s_k \times (i'_k) + r_m + b' \tag{25}$$

$$0 \leq r_m < s_d \tag{26}$$

$$\forall k, 1 \leq k \leq d \quad , \quad 0 \leq t_{mk} \tag{27}$$

$$\forall k, 1 \leq k \leq d \quad , \quad t_{mk} \leq i'_k < u_k + t_{mk} \tag{28}$$

Then, it is possible to define a new CSLMAD that covers a superset of memory locations of f_m by simply changing the lower bound of i'_k to 0. This rewriting scheme can be utilized to define the desired set of constraints to obtain G'.

First, let the bases of the n CSLMADs in G' be $b' + r_1, b' + r_2, \ldots, b' + r_n$. Let the upper bounds of the domain of G' be $u'_1, u'_2, \ldots, u'_n$. The first condition imposed is to ensure that the difference of the bases does not exceed s_d. This is indirectly imposed as the following n constraints:

$$\forall m, 1 \leq m \leq n,$$

$$0 \leq r_m < s_d \tag{29}$$

Second, it must be possible to do a rewriting of the m^{th} member of G to the m^{th} member of G' as shown above. It results in the following n constraints:

$$\forall m, 1 \leq m \leq n,$$
$$\sum_{k=1}^{d} s_k \times t_{mk} + r_m + b' = b_n \tag{30}$$

Third, the domain of the rewritten RCSLMAD f_m must be contained within the domain of m^{th} component of G'. This requirement results in the following $m \times n$ constraints:

$$\forall m, \forall k, 1 \leq m \leq n, 1 \leq k \leq d,$$
$$u_k + t_{mk} \leq u'_k \tag{31}$$

Fourth, strides and upper bounds of G' must follow the conditions from Theorem 2 resulting in $d - 1$ conditions

$$\forall k, 1 \leq k < d,$$
$$s_k > s_d - 1 + \sum_{p=m+1}^{d} s_p \times (u'_p - 1) \tag{32}$$

Using the constraints 29, 30, 31 and 32 a feasible solution can be found and can be used to construct the group G'. However, such a group may be defined over a larger domain than necessary. To obtain a good solution in terms of the size of the domain, a linear objective function is introduced

$$Minimize : \sum_{k=1}^{d} u'_k \tag{33}$$

A final constraint restricts the domain of b'. It is necessary to constrain the search of b' to a small domain for an Integer Linear Program (ILP) solver to solve the system in a reasonable amount of time. The group G' is defined such that the pattern of memory accesses repeats after

s_1 addresses. Therefore, the search is constrained to an interval of length s_1.

$$b_1 - s_1 < b' \leq b_1 \tag{34}$$

The above objective function and constraints are given to an integer linear programming (ILP) solver and if a solution is found, then the set G' can be constructed. To constrain the JIT compilation time, the compiler aborts the attempt to solve this case if the number of unknown variables exceeds a threshold. The mapped GPU addresses are computed using Algorithm 1 where I_m is the computed GPU address for the m^{th} RCSLMAD in the group.

4.4 Loop tiling for handling large loops

When the data to be transferred for an RCSLMAD group does not fit on the GPU memory resource, then the compiler must attempt to tile the loop. The compiler only tiles parallel loops because such tiling 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.

Algorithm 1 computes the mapped address for multidimensional RCSLMAD problems solved using the integer linear programming method

Inputs: Specified constants u_k, s_k, b_m. Computed values $t_{m,k}$, u'_k, b' and r_m.
Outputs: A list of n expressions representing the addresses of RCSLMADs on the GPU

1: Construct a vector $R_1 = [b' + r_1, b' + r_2, \ldots, b'_n]$.
2: Remove all duplicate entries from R_1.
3: Compute q = length of R_1.
4: Compute $b_0 = min(R_1)$.
5: **for** each RCSLMAD L_m **do**
6: $\quad I_m = b' + r_m - b_0 + \sum_{k=1}^{d} q * (i_k + t_{m,k}) * \prod_{l=k+1}^{d} (u'_l)$.
7: **end for**
8: Return list $\{I_1, I_2, ..., I_n\}$

Ideally, tiling should minimize the total amount of data transferred between the CPU and GPU. However, currently jit4GPU simply reduces the upper bound of each parallel loop to half of

its original value. If there are m parallel loops, then 2^m tiles are formed. Jit4GPU then computes the amount of data to be transferred for each tile and, if it still exceeds the capacity of a GPU memory resource, continues breaking the loop into smaller tiles. Attempts to tile the loop are aborted if the total number of tiles exceeds a set threshold (64 in this implementation).

4.5 Execution

After determining the properties of all the GPU resources, jit4GPU issues run-time calls to allocate resources and transfer of data. The run-time system uses the fastest method to transfer data depending on strides and alignment of data in CPU memory. The runtime is also responsible for transferring data back from resources to the same RCSLMADs after the computation is complete. To execute the loop nest, jit4GPU assigns one GPU thread to each parallel loop iteration. The code is then compiled to AMD IL by jit4GPU, passed to AMD CAL IL compiler, which compiles the IL to GPU ISA, and is then executed by jit4GPU using the CAL API.

5 Experimental Evaluation

This experimental evaluation compares the performance of the following solutions: Generated OpenMP code, CPU BLAS routines where available, generated GPU code without loop optimizations and generated GPU code with loop optimizations. This evaluation supports the following findings:

- Generated GPU code can reduce execution time by over an order of magnitude compared to generated OpenMP code.
- GPU code generated from naive Python code can outperform very highly tuned CPU libraries such as BLAS.
- Loop optimizations performed by jit4GPU are very effective in reducing the GPU execution time.

5.1 Experimental Platform

Experiments were performed on a Phenom X4 925 2.8 GHz quad-core CPU equipped with a Radeon 5850 GPU clocked at 725 MHz with 1 GB of 1 GHz GDDR5. The software platform used was Python 2.7 with NumPy version 1.4.1 running on 64-bit Linux 2.6.38 kernel. GCC 4.5 was used as the C/C++ compiler with optimization flag -O3. All 4 cores were used when running CPU code. OpenMP code generation used the flag -fopenmp in gcc.

5.2 Experimental results

For each benchmark, we compare the running time of C+OpenMP version with GPU accelerated versions. We also compare with hand-optimized C and with x86 assembler libraries when available. For GPU accelerated versions, we report multiple results. Bars marked as GPU represent the case where the compiler generated CAL code that directly corresponds to the loop body written in Python with no attempt to optimize the code for GPUs. In this version, compiler has not performed transformations outlined in section 3.4 such as unroll-and-jam. load coalescing or redundant load elimination. GPU-Opt represents the case where transformations described in section 3.4 have also been attempted and thus the compiler has generated optimized CAL code. Array access analysis as described in section 4.3 is performed in both cases as analysis is necessary for the compiler to figure out the necessary data transfers.

All experiments were performed 30 times by repeatedly executing the application binary. The arithmetic average of these 30 execution times is reported. A 95% confidence interval was computed but it is not added to the bar graphs because the variations are so small that the confidence intervals would not be visible in the graphs.

Execution times are shown in Figures 9 and 10. The naming convention is: **MmultD** is the double-precision version and **MmultS** is the single-precision version of the benchmark. There are five bars for each benchmark, the first, **CPU**, shows the execution time by a multithreaded implementation on the CPU: either ATLAS (for `MMult`, `rank-K`, and `rank-2K`) or OpenMP (for all other benchmarks). All other bars present the total execution time, including transfer time and JiT overhead for an execution in the GPU: in **GPU-NoOpt** none of the code transformations

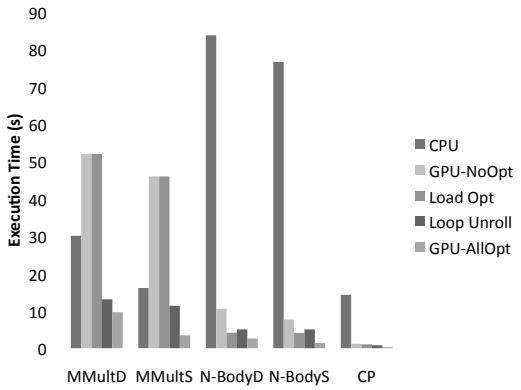


Figure 9: Execution time for matrix multiplication (× ATLAS BLAS), N-body (× OpenMP) and CP (× OpenMP)

described in Sections 3 and 4 are done — the code generated is a straight translation of loop bodies as written by the programmer in Python; **Load Opt** is the execution time when jit4GPU attempts to perform load coalescing and redundant load elimination; in **Loop Unroll** jit4GPU attempts to unroll and jam loops; **GPU-AllOpt** is when all the transformations described in this paper are switched on.

5.2.1 Matrix multiplication

Multiplies two $N \times N$ dense contiguous matrices. The comparison is against the CPU BLAS functions `sgemm` and `dgemm` for single and double-precision respectively. For the dou-ble-precision case, the full matrices do not fit in the GPU memory. Jit4GPU tiles the computation automatically so that the data required for one tile of the computation fits in the GPU memory.

The bars in Figures 9 indicate that, as expected, loop optimizations are key for the GPU to outperform the highly optimized multithreaded ATLAS implementation for matrix multiplication. Redundant load elimination plays no role in the performance of these benchmarks.

5.2.2 Synthetic N-body kernel

Given points in a 3D space, the kernel computes the sum of distances of each point from every other point. The sum of distances is in itself usually not a useful computation but may be used as part of other computations. The kernel stores the coordinates of the points in three distinct arrays. The kernel performs $9n^4$ floating-point operations (flops) for a problem size of n. Results are shown in Figure 9. The optimized double-precision GPU version produces a throughput of approximately 240 GFlops compared to approximately 8 GFlops on the CPU. Examination of the generated assembly by the OpenMP compiler revealed that SSE instructions were generated. The CPU code is also fully multi-threaded. Thus, the comparison is not unfair to the CPU. CPU reached approximately 17% of its theoretical ALU peak compared to 57% achieved by the GPU. Load and loop optimizations still have a significant impact in the performance `N-Body`. For instance, for `N-BodyS` the GPU-AllOpt is 5.5 times faster than the GPU-NoOpt.

High performance of the GPU compared to the CPU is expected in floating-point intensive benchmarks such as this synthetic N-body kernel. At the time of writing, theoretical ALU peak of even top-end x86 server-class CPUs such as a 12-core Opteron 6180SE is only around 120GFlops for double-precision operations. Thus, even if the CPU version were to obtain a 100% efficiency (an unrealistic scenario) on a top-end CPU, the reported performance in the GPU would still significantly outperform x86 CPUs on this benchmark. Using faster AMD GPUs such as the Radeon 6970 will only widen the GPU lead in this benchmark.

5.2.3 Coloumbic Potential benchmark

CP is a Python implementation of the CP benchmark from the Parboil benchmark suite. CP computes the Coloumbic Potential at each point on a grid. The properties of the point objects are stored in a format emulating an array of structures. This is a single-precision benchmark. This benchmark utilizes a single array of multiple components as opposed to multiple arrays of single components in the nbody kernel. The GPU easily outperforms the CPU as shown in Figure 9. The data-transfer overhead becomes a significant issue for the optimized GPU version and thus further speedups through any other optimizations, or by using faster GPUs, will yield diminishing returns.

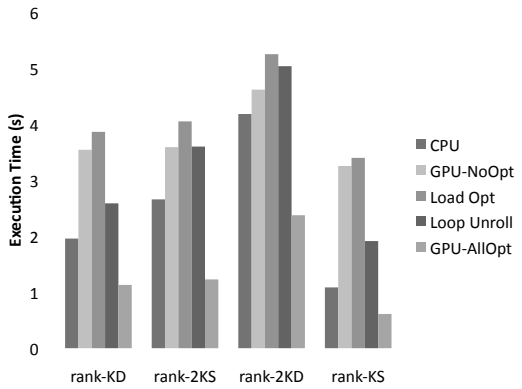


Figure 10: Execution time for rank-k update and rank-2k update comparing ATLAS BLAS and GPU performance

Load and loop optimizations have little influence on the GPU speedup for this benchmark.

5.2.4 Rank-k update and Rank-2k update

Rank-k computes updates of the lower triangular part of a symmetric matrix. The triangular loop structure prevents optimizations of one of the loops in the loop nest but the innermost loop was still optimized. Performance is compared against the CPU BLAS functions `ssyrk` and `dsyrk` for single and double-precision respectively. For rank-2k performance is compared against the CPU BLAS functions `ssyr2k` and `dsyr2k` for single and double-precision respectively. Execution time for both benchmarks are shown in Figure 10.

5.2.5 5-point stencil benchmark

This kernel performs an out-of-place 5-point stencil computation where the new value of each element in a 2-dimensional matrix is a weighted average of five neighboring points including itself. The kernel performs very little computation — (24 *ms* in the CPU, 26 *ms* in the GPU, and 25 *ms* for the optimized GPU code — and the data transfer time and kernel compilation time dominates execution as shown in Figure 11. This benchmark illustrates that the compiler is able to perform accurate array access analysis and to generate GPU code even with relatively complex

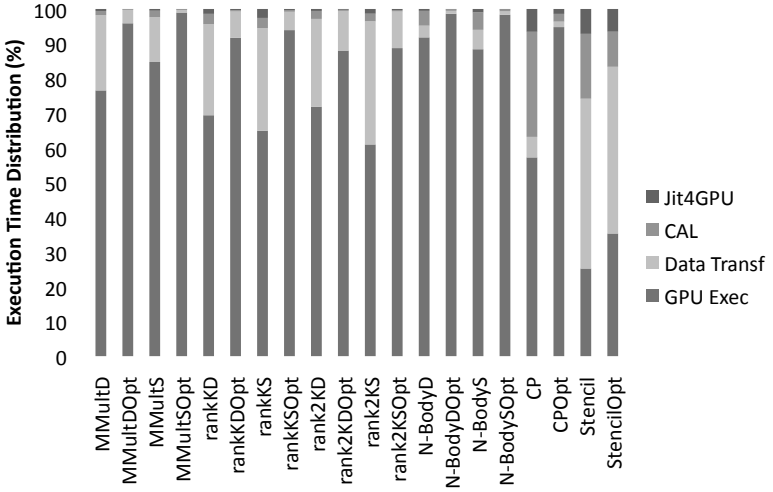


Figure 11: Percentage distribution of GPU execution time for all benchmarks

cases involving multiple references to the same array. This is a double-precision benchmark.

5.3 Time Distribution in the GPU Execution

The lower segment of each bar in Figure 11 shows the percentage of time spent performing computations in the GPU (GPU Exec). The other segments are the fraction of time spent on data transfers, on compilation in the CAL compiler, and on JIT compilation, respectively. Two versions of each benchmark are shown. For instance, MMultD corresponds to the GPU-NoOpt version in Figure 9 while MMultD-Opt corresponds to the GPU-AllOpt version. The most salient effect of the optimization performed by jit4GPU is to reduce the time spent in data transfer and thus to increase the proportion of time spent on execution in the GPU. A comparison between the time distributions for the two versions of each benchmark confirms this expectation.

6 Related Work

Garg first described the combination of an ahead-of-time and a just-in-time compiler to create an end-to-end system for the execution of numerical Python programs on CPU/GPU combinations [14]. A preliminary description of unPython outlined the design strategy [13]. An extension of this compilation framework that targets the generation of OpenCL code trades performance for portability [22]. This archival article has a complete description of the data access analysis based on LMADs and a throughly revised and updated final presentation of the experimental evaluation of the entire compilation system. This section covers work on array access analysis, compilers targeting GPUs, and software-managed data transfers.

6.1 Compiling Python for CPUs

Various compilers have been developed that compile Python, or a closely related dialect, to C/C++ or other lower-level languages. Pyrex [12] compiles the Pyrex language to C/C++. This language is statically typed and has 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, the Pyrex language is not accepted by a Python interpreter. Further, Pyrex does not support NumPy arrays or parallel programming. Cython [4] is a language derived from Pyrex that has a syntax much closer to Python and supports efficient access to NumPy arrays. However, Cython does not support parallel programming.

Several compilers attempt to compile pure Python to lower-level languages using type inference. Shedskin [10] compiles non-annotated Python to C++. In order to implement type inference, Shedskin relies on implicit typing restrictions on the Python code being compiled. Shedskin is a whole-program compiler with a more advanced type inference algorithm than unPython. However Shedskin does not support efficient accesses to NumPy arrays and does not have any parallel-programming features.

Another compiler that compiles non-annotated Python to a lower-level language is RPython [2] from the PyPy [1, 6, 26] 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 including C. PyPy will not support efficient access of NumPy arrays until NumPy arrays are implemented in RPython.

6.2 Array access analysis

The primary objective of array access analysis is to find out the memory locations to be transferred between the CPU and the GPU address spaces. The array access analysis used is based on the concept of LMADs. LMADs were first defined by Paek *et al.* [25]. 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 on the lines of the computation of RCSLMADs described in this paper. Rus *et al.* present a run-time representation of LMADs called RT-LMADs [27]. Their goal is to apply array access analysis to parallelization and hence they focus on computing dependence information. In contrast, the goal of the analysis in this paper is to obtain an efficient representation of the union of all regions accessed with the constraint that this union should easily map to a different address space.

6.3 Compilers for GPGPU

Early use of GPUs for non-graphic computation relied on a stream programming model where the streams represent data to be transferred to the GPU. Brook is an extension of the C programming language that provides a stream data type and kernel functions that run in the GPU and manipulate streams [7]. The programmer has to code the transfer of data into, and out of, the GPU address space. Other proposed stream-based programming languages for GPGPU include StreamIt [31], Sh [23] and Stream Virtual Machine [18].

Current APIs to program GPUs to execute non-graphics code include OpenCL [24] and Nvidia CUDA [9]. Nvidia CUDA extends C/C++ and models the GPU as a Single-Program Multiple-Data (SPMD) machine that executes the same code in many GPU threads. Stratton *et al.*'s 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 [29]. MCUDA compiles CUDA code to multi-core x86 code. MCUDA's philosophy is the opposite of ours. We provide an illusion of a symmetric multiprocessor machine while automatically utilizing GPU.

The model that most closely resembles the programming and compilation model presented in this paper describes a compiler that can automatically generate Nvidia CUDA code from C/C++ programs with OpenMP annotations [19]. But there are three fundamental differences. First, their data-transfer analysis can only deal with arrays but not with pointers — their analysis is based on array names and on finding the declarations of the arrays in the source code. Second, they transfer entire arrays. In contrast the analysis presented in this paper can handle both arrays and pointers, and restrict data transfers to memory regions that contain mostly referenced locations. Third, our JIT-based approach does not require the source code of the entire application. Also, they cannot tile or break the loop if the data does not fit into the GPU. Finally, their loop transformations are different because they target the Nvidia GPU architecture.

Wang *et al.* introduces EXOCHI, an extension of C/C++ OpenMP for heterogeneous systems [32]. Their implementation is for a multi-core x86 CPU and for an integrated Intel graphics chip set. These chip sets are integrated into the north bridge of the CPU and can access the system RAM. Therefore EXOCHI does not need to handle data movements between different address spaces: a simple remapping of the address translation table enables the GPU to reference the data. Such a technique cannot be used with discrete AMD GPUs. This remapping is handled by EXOCHI's runtime. In EXOCHI the programmer must write GPU code but does not need to implement data transfers. EXOCHI is only suitable to accelerators that can access system RAM directly. Therefore it is not applicable to the current generation of discrete GPUs.

Saha *et al.* present a shared-memory programming model for a simulation of discrete Larrabee GPUs based on the x86 instruction set [28]. However, they require the programmer to annotate some data structures with declarations indicating which data structures are shared between CPU and GPU while our compiler requires no such annotation.

Yang *et al.*'s source-to-source compiler takes as input a naive kernel consisting of code for a

single thread and generates an optimized CUDA kernel [33]. They analyze the memory access pattern of array references and attempt to perform memory load coalescing, automatic utilization of on-chip shared memory and unrolling and tiling of parallel loops. Their analysis is limited to single induction variable in each dimension of the subscript.

Theano is a Python library that compiles expression graphs into either C code or Nvidia CUDA code [5]. For GPU code, Theano automatically transfers all the input and output arrays in an expression to and from the GPU. Unlike jit4GPU, Theano does not perform an array access analysis and instead transfers whole arrays. Copperhead, by Catanzaro *et al.*, is a compiler from Python to CUDA that only compiles functions with statements involving only a predefined library of parallel operations on entire vectors such as replicate, map and reduce [8]. It does not handle loops.

Baskaran *et al.*'s compiler automatically generate CUDA kernels from C code using a polyhedral representation of loop nests [3]. However, they do not discuss how to generate code to copy data between the CPU address space and the GPU address space.

6.4 Software Managed Data Transfers

The automatic management of data transfers between CPU and GPU address spaces is analogous to the management of data in Scratch-Pad Memories (SPMs) that are prevalent in embedded systems [20]. Kandemir *et al.* propose a method to combine loop tiling with a cost analysis to dynamically manage the transfer of data into SPMs [17]. Their "extreme values of affine functions" method appears to have similar elements to our data access analysis, but they do not have a similar address-mapping issue. Moreover, instead of using LMADs to describe the loop accesses they offer an informal description of the array analysis. Li *et al.* also use loop tiling to manage SPMs but their tiling requires an Integer Linear Programming (ILP) solver that is too slow to be used in a JIT compiler [21]. Udayakumaran and Barua's approach is less restrictive on the constraints imposed on the loops containing the references to be transferred to SPM, but it only transfers entire arrays and will not work for programs containing recursion or function pointers [30].

Another architecture that requires a compiler to manage data transfers across different address spaces is the CELL processor. The compiler for the CELL processor developed by Eichenberger *et*

al. also uses loop tiling, which they call *blocking*, to generate the SIMD code required for execution in a Synergistic Processor Element (SPE) in the CELL architecture [11]. Similar to the compilation system presented in this paper, their compiler provides a single shared-memory abstraction through the use of compiler-inserted Direct Memory Access (DMA) transfers.

7 Conclusion

The key for building a compilation framework for the new programming model presented in this paper was to split the compilation into two steps: an ahead-of-time compiler (unPython) and a just-in-time compiler (jit4GPU). To implement the illusion of a shared address space, the compiler performs an analysis to determine the memory locations that must be transferred to the GPU. The compilation system does automatic data transfer, based on a detailed analysis of data usage, between GPU and CPU. The paper formally describe an analysis based on integer constraints to enable a reduction in the number of memory locations transferred between the CPU and the GPU address spaces. Moreover, some traditional loop optimizations were successfully adapted to this environment to overcome important imbalances in the design of the GPU. Only unobtrusive light annotation is needed in the Python code to assist with type inference and to indicate which loops may be profitable to execute in the GPU. This compilation framework was implemented for an AMD GPU. It delivers significant speedup over generated OpenMP code, and it either outperforms or is very competitive with highly optimized multi-threaded CPU BLAS code for the same benchmarks.

Acknowledgements

This work was supported by Natural Science and Engineering Council (NSERC) of Canada. We would also like to thank Micah Villmoh, Michael Chu and others at AMD Stream Computing group who provided many clarifications about the CAL SDK.

References

[1] Pypy project (2009-09-30). http://codespeak.net/pypy/dist/pypy/doc.

[2] D. Ancona, M. Ancona, A Cuni, and N. Matsakis. "RPython: A step towards reconciling dynamically and statically typed OO languages". In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 53–64, Montreal, QC, Canada, 2007.

[3] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Compiler Construction (CC)*, pages 244–263, Paphos, Cyprus, 2010.

[4] S. Behnel, R. Bradshaw, and D. S. Seljebotn. Cython: C-Extensions for Python (2009-09-30). http://www.cython.org.

[5] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, and Y. Bengio. Theano: a CPU and GPU math expression compiler. In *Python for Scientific Computing Conference (SciPy)*, Austin, TX, June 2010.

[6] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Progr. Systems*, pages 18–25, Genova, Italy, 2009.

[7] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics (TG)*, 23(3):777–786, 2004.

[8] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 47–56, San Antonio, TX, USA, 2011.

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

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

[11] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing compiler

for the CELL processor. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 161–172, St. Louis, MO, USA, 2005.

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

[13] R. Garg and J. N. Amaral. unPython: Converting python numerical programs to C. In *7th Python in Science Conference (ScyPy)*, Pasadena, CA, August 2008. http://conference.scipy.org/proceedings/SciPy2008.

[14] R. Garg and J. N. Amaral. Compiling Python to a hybrid execution environment. In *3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 19–30. ACM, 2010.

[15] Jay Philip Hoeflinger. *Interprocedural parallelization using memory classification analysis.* PhD thesis, Champaign, IL, USA, 1998.

[16] W. Jones. *Beginning DirectX 10 Game Programming.* Course Technology Press, Boston, MA, USA, 1st edition, 2007.

[17] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In *Design Automation Conference (DAC)*, pages 690–695, 2001.

[18] F. Labonte, P. Mattson, W. Thies, I. Buck, C. Kozyrakis, and M. Horowitz. The Stream Virtual Machine. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 267–277, Antibes Juan-les-Pins, France, 2004.

[19] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 101–110, Raleigh, NC, USA, 2009.

[20] L. Li, L. Gao, and J. Xue. Memory coloring: A compiler approach for scratchpad memory management. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 329–338, St. Louis, MO, USA, 2005.

[21] L. Li, H. Wu, H. Feng, and J. Xue. Towards data tiling for whole programs in scratchpad memory allocation. In *Asia-Pacific Conference Advances in Computer Systems Architecture (ACSAC)*, pages 23–25, Seoul, Korea, August 2007. Springer.

[22] X. Li, R. Garg, and J. N. Amaral. A new compilation path: From python/numpy to opencl. In *Workshop on Python for High Performance and Scientific Computing (PyHPC)*, Seattle, WA, USA, November 2011.

[23] M. D. McCool, Z. Qin, and T. S. Popa. Shader metaprogramming. In *ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 57–68, Saarbrucken, Germany, 2002. Eurographics Association.

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

[25] Y. Paek, J. Hoeflinger, and D. Padua. Efficient and precise array access analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(1):65–109, 2002.

[26] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Workshop Companion to Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 944–953, Portland, OR, USA, 2006.

[27] S. Rus, L. Rauchwerger, and J. Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. *International Journal of Parallel Programming*, 31(4):251–283, 2003.

[28] B. Saha, X. Zhou, H. Chen, Y. Gao, S. Yan, M. Rajagopalan, J. Fang, P. Zhang, R. Ronen, and A. Mendelson. Programming model for a heterogeneous x86 platform. In *Programming Language Design and Implementation (PLDI)*, pages 431–440, Dublin, Ireland, 2009.

[29] J. Stratton, S. S. Stone, and W.-M. W. Hwu. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In *Workshop on Languages and Compilers and Parallel Computing (LCPC)*, pages 16–30, Edmonton, AB, Canada, August 2008.

[30] S. Udayakumaran and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 276–286, San Jose, California, USA, 2003.

[31] 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, Seattle, WA, USA, 2009.

[32] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Programming Language Design and Implementation (PLDI)*, pages 156–166, San Diego, CA, USA, 2007. ACM.

[33] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU compiler for memory optimization and parallelism management. In *Programming Language Design and Implementation (PLDI)*, pages 86–97, Toronto, ON, Canada, 2010.