A New Compilation Path: From Python/NumPy to OpenCL

Xunhao Li
IBM Software Laboratory
Markham, ON, Canada

Rahul Garg
Department of Computer Science,
McGill University,
Montreal, QC, Canada
Email: rahul.garg@mail.mcgill.ca

José Nelson Amaral
Department of Computing Science,
University of Alberta,
Edmonton, AB, Canada
Email: amaral@cs.ualberta.ca

***Abstract*— *Jit4OpenCL* is a new compiler that converts scientific applications written in Python/NumPy into OpenCL code. This compiler is based on *unPython*, an ahead-of-time compiler from Python/Numpy to an intermediate form and OpenMP code, and on *jit4GPU*, a just-in-time compiler that converts that intermediate code into AMD CAL code that is specific for AMD GPUs. The targeting of OpenCL provides a new evaluation point in the tradeoff between performance and portability for this type of application. Adapting the data access analysis to the OpenCL model was critical for the construction of this new compiler. An experimental evaluation compares the execution time of benchmarks compiled with jit4GPU and jit4OpenCL with highly optimized versions of the same benchmarks running on the CPU.**

***Keywords*-OpenCL, GPGPU, Python, Code Generation, Compilers, Just-in-Time Compilation**

I. Introduction

The Imaging Reduction and Analysis Facility (IRAF) is an astronomical analysis system that has been in use at the Space Telescope Science Institute (STScI) since the early 1980s. Greenfield describes how the initial use of Python to develop an alternate scripting environment for IRAF, which led to PyRAF — the Python scripting environment for IRAF, eventually led scientists at the STScI to "develop as many applications in Python as possible" [1]. Their only concern was the efficiency of the code generated. However, between the development of PyRAF and today, Python developers have allayed such concerns through the development of APIs that allow both modules written in other, usually procedural or object-oriented, languages to be called from within Python and the calling of Python modules from within programs written in such languages.

A large number of modules have been developed using such APIs. Of central interest to this paper is a module called NumPy. The development of NumPy was in part motivated by needs identified at the STScI in their attempt to code more applications in Python [2] . NumPy defines a fast multi-dimensional array class that offers rich and versatile arrays, array views, and slicing operators that separate arrays from views. NumPy exposes C arrays as high-level arrays in Python and bound checks subscripts. NumPy, which is implemented in C, endows Python with efficient sequential execution of scientific code.

Thus the scientific community has a productive way to develop scientific code in Python/NumPy to execute efficiently on a single CPU. But how about the utilization of multiple CPUs on a single core, or the exploitation of the cheap computing power made available by modern Graphic Processing Units (GPUs)? There is a class of applications that can be run efficiently in GPUs. Applications in this class include several scientific applications such as the ones that have been created using the combination of Python and NumPy. Thus, it would be nice to have an end-to-end solution in which applications could be translated from Python/NumPy to code that can run efficiently on a GPU.

Garg and Amaral have proposed and implemented such a model [3], [4]. That programming environment requires minimal annotations to the Python/NumPy code. These annotations are necessary to aid with the type inferencing required for the compilation of dynamically typed programs and to signal to the compiler that a given loop nest is parallel. The annotations were carefully designed to enable a normal Python interpreter to run the annotated program without any change. This is accomplished by using decorators to provide type information — all the decorators introduced are imported from the unPython module to be identity decorators when unPython is not being used, thus the interpreter ignores these decorators in that case — and by defining two new `range` functions, called `prange` and `gpurange`, in the programming environment to signal that a loop should be executed in parallel in the CPU or in the GPU. Garg's programming environment is supported by two new compilers.

This research was partially funded by a grant from the Natural Sciences and Engineering Research Council (NSERC) of Canada.

The ahead-of-time compiler, called *unPython*, converts Python/NumPy code with annotations into an intermediate representation and also produces OpenMP code. Then a just-in-time compiler, called *jit4GPU*, generates AMD CAL code and code to manage the transfer of data between the CPU memory space and the GPU memory space. The generated CAL code is then compiled into AMD GPU assembly instructions using an AMD CAL compiler provided by AMD. In addition, a small runtime system takes care of the data transfers between CPU and GPU.

Garg's programming environment is a good first solution for the problem of bridging scripting languages used for scientific computing on one side, and the computing power of GPUs on the other. It was also one of the first compiling solutions that automatically determined the set of memory locations that needs to be transferred to the GPU and generates the data transfers automatically. However, it is bound to GPUs fabricated by AMD because it generates AMD CAL code. This is a reflection of the time in which it was developed — then all GPUs required some form of brand-specific code in order to run programs. However, the release of the OpenCL programming interface, and of compilers to translate OpenCL code to each GPU native code, allows for the targeting of OpenCL as an intermediate representation. In addition, several other computing platforms, including multi-core CPUs, are developing a compilation path from OpenCL.

The contribution in this paper is a compilation path from the annotated Python/NumPy programs in Garg's programming environment to OpenCL. The new compiler, which we call *jit4OpenCL*, works in tandem with Garg's unPython but generates OpenCL code instead of AMD CAL code. Similar to jit4GPU, jit4OpenCL uses Restricted Constant-Stride Linear Memory Access Descriptors (RCSLMAD) to perform the necessary analysis and determine the memory locations that need to be transferred to the device. There is an important difference between the two compilers: jit4GPU optimizes for texture caches while jit4OpenCL optimizes for local memory(which is equivalent to the shared memory in CUDA context). Due to the different performance designs and usage of texture memory and local memory, the optimization phases of the two compilers are very different. The jit4OpenCL optimization involves several code changes that did not exist in jit4GPU because when jit4GPU was written AMD hardware did not have flexible local memory.

As should be expected, portability has a performance cost. The use of the more general representation in OpenCL does deliver lower performance than the more specific AMD CAL target. The paper summarizes the findings of an extensive performance study that compares the performance of programs generated by jit4GPU and by jit4OpenCL on the same machine. Given the portability goal of jit4OpenCL we also summarize performance results for programs generated using jit4OpenCL for other platforms. These results indicate that there is room both in the implementation of jit4OpenCL and in the OpenCL compilers provided by vendors for performance improvements.

```
1 __kernel void multiply(__global float *A,
      __global float *C, int N){
2     int row = get_global_id(0);
3     int col = get_global_id(1);
4     float sum = 0.0f;
5     for (int i=0; i<N; i++){
6       sum += A[row*N + i]*A[i*N+col];
7     }
8     C[row*N + col] = sum;
9 }
```

Listing 1. Naive OpenCL Matrix Multiplication

II. OpenCL Background

The OpenCL programming model supports parallel computing in an heterogeneous machine formed by a *host* that controls a set of *devices*. A program issues many thread instances, called *work-items*, for execution. Work-items are grouped into *work-groups*. All work-items in a work-group have access to dedicated fast local memory space. The index space that identifies work-items/threads is organized as a grid with each element in the grid representing a work group. Readers are redirected to [5] for details.

A. An OpenCL Example

Consider the computation of the matrix multiplication ($C = A \times A$). For an element in C, say, (i, j), the following computation must be performed:

$$C(i,j) = \sum_{k=1}^{N} A(i,k) \times A(k,j)$$

The host program allocates two memory buffers on the global memory of the device: one for array A and another for array C. For simplicity, assume that A and C can be stored in a device memory at the same time. Then, the host copies array A to a device memory. The host then starts the execution of a kernel function in the device's computing units. These units compute C and store the result into the device memory. The result is then transferred back to host memory.

Listing 1 shows the kernel function when $N \times N$ threads are created in the device to compute C. Lines 2 gets the thread ID for the first and second dimensions. Lines 4-6 accumulate the sum. Line 8 stores the result to global memory. The performance is very low because of the frequent accesses to the high-latency global memory that stores A. In current GPUs, the latency for each global-memory access can be as high as 300 to 500 device cycles. Moreover, the global-memory access of each column is strided, which causes banking conflicts in most hardware architectures, resulting in even longer access latency.

A better strategy is to replace global-memory accesses with local memory accesses to reduce both bandwidth requirements and access latency. In OpenCL, the local memory space is a limited-sized cache that is software-managed by threads. Therefore the optimized kernel must include code that issues loads and stores to the local memory. Multiple threads need to read the same set of array elements. The bandwidth demand can be reduced by moving array elements into local memory by tiles. The bigger the tile is, the less the bandwidth demand will be. But local memory is a scarce resource, thus the tile size has to balance bandwidth and storage space. A CUDA version of the steps to improve a matrix multiplication is presented by Ryoo *et al.* (Fig. 3 in their paper) [6].

Listing 2 shows the kernel that have tiled access for local memory utilization. Two local memory array spaces, `aTile1` and `aTile2`, are used as intermediate fast storage space. Threads interleave the computation with local memory loading operations. The outer loop `tiled_i` holds the code that loads the global memory tile to local memory space and the synchronization that maintains data consistency. The barrier immediately after the inner loop `i` ensures that all threads in a thread group tile have the same computing process. The two barriers guarantee that the inner loop has access to the desired elements in local memory space. Array offsets are computed from the thread local ID and the group ID. A thread loads two elements to local memory in each tile, but accesses $2 \times TILE$ elements in local memory space, thus eliminating global memory accesses. Assume that both the tile size and the thread group size are also $TILE \times TILE$ and the matrix size is $N \times N$. The total bandwidth consumption is only $\frac{1}{TILE}$ of the non-optimized kernel, which results in around $10\times$ speedup in performance.

```
1  // TILE is the constant size of a tile
2  __kernel void multiply(__global float *A,
       _global float *C,
3        int N,
4        __local float aTile1[TILE*TILE],
5        __local float aTile2[TILE*TILE]){
6   float sum = 0.0f;
7   int x = get_local_id(0);
8   int y = get_local_id(1);
9   int gidx = get_group_id(0);
10  int gidy = get_group_id(1);
11  int row = gidx*TILE + x;
12  int col = gidy*TILE + y;
13  int tiled_i, i;
14
15  for(tiled_i=0; tiled_i<N; tile_i+=TILE){
16    aTile1[y*TILE+x] = A[(gidy*TILE*N+
          tiled_i)+y*N+x];
17    aTile2[y*TILE+x] = A[(tiled_i*N+gidx*
          TILE)+y*N+x];
18    barrier(LOCAL_MEM_FENCE);
19    for(i=tiled_i; i<tiled_i +TILE; i++){
20      sum += aTile1[y*TILE+i] *aTile2[i*
            TILE+x];
21   }
22   barrier(LOCAL_MEM_FENCE);
23  }
24  C[(gidy*TILE+y)*N+gidx*TILE+x] = sum;
25 }
```

Listing 2. Tiled OpenCL Matrix Multiplication

III. Data Transfer Analysis

The memory access analysis based on RCSLMADs was developed for the jit4GPU compiler [3]. In jit4GPU the RCSLMAD-based analysis was used to determine the amount of memory accessed by each tile of a loop nest. Jit4GPU used the result of the analysis to determine how each loop nest should be tiled to ensure that the memory accessed by each tile fits into the GPU memory. In contrast, jit4OpenCL uses the same analysis for data transfer, but in addition it also uses the analysis to determine how a loop nest should be tiled so that each tile fits into on-chip local memory. Thus jit4OpenCL uses the result of the analysis for two purposes: (1) to determine the necessary data transfers — the same way that jit4GPU did — and (2) to determine the required on-chip share memory allocations that are necessary and to find opportunities to reduce the allocation requirements through reuse and compactation — on-chip memory allocations were not necessary in jit4GPU. This section starts by briefly recaping the definition of RSCLMADs and their properties.

A. Identifying Array Access Elements

Linear Memory Access Descriptors (LMADs) are introduced by Paek *et al.* [7]. Let M be the set of memory locations referenced by an array access. Given a loop nest of depth d, the memory references within the loop nest may be represented by a function f, whose input is a d-component vector $\vec{i} = (i_1, i_2, i_3, ..., i_d)$,

where i_k is the k^{th} loop index in the loop nest. The set of legal loop indices $\vec{i}$ forms a d-dimensional iteration domain D. D is represented in the following format: $D = (l_1..u_1, l_2..u_2, \ldots, l_d..u_d)$, where for any $0 < k \leq d$, l_k and u_k are the lower and upper bounds of loop k, respectively, and $l_k \leq u_k$. For normalized loops, the lower bound of each loop counter is 0.

Rahul and Amaral introduced some variants of LMAD in [4]. A Constant-Stride LMAD (CSLMAD) further constrains D by requiring the upper bound to be an affine function of the outer loop indices [4]. Let $f : \vec{i} \rightarrow \mathbf{N}$ be the following affine function:

$$f(\vec{i}) = b + \sum_{k=1}^{d} s_k \times i_k | \vec{i} \in D \quad (1)$$

where $s_k | k = 1, 2, ..., d$ are the increment strides of each loop counter, b is the base of the array-access descriptor. The function $f(\vec{i})$ is an array descriptor.

For the definition of a Restricted CSLMAD (RCSLMAD), assume that the strides are sorted in decreasing order. Let s_{r_k} be the k^{th} stride in this sorted list, u_{r_k} be the upper bound, and r_k be the position in the sorted list of the loop with stride s_{r_k}. Let $s_{r_m} = \min(s_{r_k}) | s_{r_k} \neq 0$. A CSLMAD is a RCSLMAD if and only if for every s_{r_k} we have:

$$s_{r_k} \geq s_{r_m} - 1 + \sum_{j=k+1}^{m} u_{r_j} \times s_{r_j} \quad (2)$$

This restriction guarantees that in a RCSLMAD the references with small strides do not overlap with references with larger strides. In a RCSLMAD each reference is to a unique memory location, which greatly simplifies the analysis.

A function $f(\vec{i})$ and a loop-iteration domain D define a set of memory locations $L_f(D) = \bigcup_{\vec{j} \in D} f(\vec{j})$. The memory space between $min(f(\vec{j}))$ and $max(f(\vec{j}))$ for all $\vec{j} \in D$ is the *region* defined by f on D, and is denoted as $|D|_f$. Not all locations within this region are necessarily referenced by the loop. The locations that are referenced are called the *effective memory locations* in this analysis. Figure 1 shows an example of a region. Grey blocks are effective memory locations while white blocks are not.

Theorem 1: Given $\vec{i} \in D$ and $\vec{j} \in D$, such that $f(\vec{i})$ and $f(\vec{j})$ are RCSLMADs. If $\vec{i} \neq \vec{j}$ then $f(\vec{i}) \neq f(\vec{j})$. Li gives a detailed proof for the theorems and corollaries [8].

Corollary 1: Given $D_1 \subseteq D$ and $D_2 \subseteq D$ such that $D_1 \cup D_2 = D$, $D_1 \cap D_2 = \emptyset$ then $L_f(D_1) \cup L_f(D_2) = L_f(D)$, $L_f(D_1) \cap L_f(D_2) = \emptyset$.

There may be unnecessary elements (non-effective memory locations) interleaved with effective memory locations, creating "holes" in the memory region. It is a waste to leave these holes in local memory because transferring data between different levels of device memory is expensive. Consider the example shown in Figure 1. The figure represents array accesses of the following RCSLMAD (assuming that the starting address is 0):

$$f((i, j)) = 2 \times i + 12 \times j$$

$$0 \leq i < 3 \text{ and } 0 \leq j < 3$$

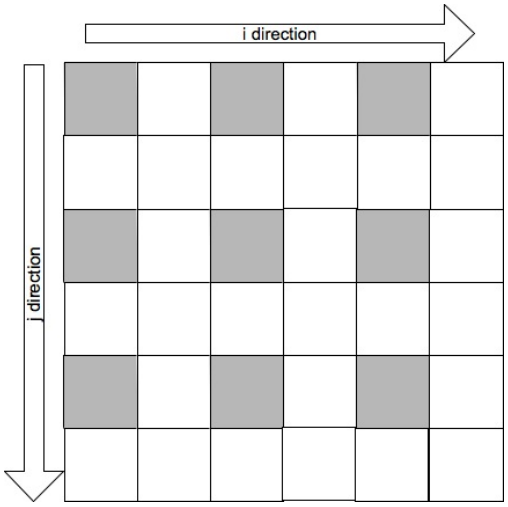

Figure 1. Example of legal RCSLMAD

Although the region of the RCSLMAD covers a large area, there are only 9 effective memory locations. In some cases transferring the whole RCSLMAD region to the device's local memory would be extremely expensive because the effective memory locations only occupy a small portion of the region.

The solution is to compress the RCSLMAD to squeeze the holes out. If $f(\vec{i}), \forall \vec{i} \in D$ is a RCSLMAD, where the j^{th} dimension's lower bound is 0, upper bound is u_j, then we construct another RCSLMAD function $g(\vec{i})$ such that its strides are $s_{r_d} = 1, s_{r_k} = \sum_{j=k+1}^{d} u_{r_j} \times s_{r_j}$, where $r_l, l = 1, 2, ..., d$ is the dimension index sorted in stride's decreasing order. This transformation of RCSLMADs eliminates the access holes from the region defined by g on D, $|D|_g \leq |D|_f$ and $|L_g(D)| = \prod_{i=1, u_i \neq 0}^{d} u_i$. Most importantly, the memory space occupied by the effective memory locations in $f(\vec{i})$ is $|D|_g$.

Theorem 2: All memory locations in the region of D on g, which form the memory region from $max(L_g(D))$ to $min(L_g(D))$, are effective memory locations.

For instance, consider the example shown in Figure 1. The constructed RCSLMAD function $g(\vec{i})$ is: $g((i,j)) = 1 \times i + u_1 \times j = i + 3j$, and $|D|_g = 9$, resulting in no waste of local memory.

In a compiler implementation, LMADs are transferred to local memory and are compressed to reduce the usage of scarce local memory using Theorem 2. Thus, given a RCSLMAD $f(\vec{i})$ representing elements in the global memory of a device, the compiler creates a new RCSLMAD $g(\vec{i})$ in the device's local memory, and maps every legal $f(\vec{i})$ to its destination $g(\vec{i})$ before starting the computation. After the computation, $f(\vec{i})$ is updated if $g(\vec{i})$ has changed.

B. Program Transformation and RCSLMAD Decomposition

Some scientific applications process a large amount of input data. The on-chip local memory in current GPUs is limited. Therefore, large RCSLMADs must be decomposed into smaller RCSLMAD tiles such that:

1) each RSCLMAD tile fits in local memory space;
2) a GPU's stream processor can process one RCSLMAD tile at a time;
3) the computation result is the same as with the original RCSLMADs.

To decompose a RCSLMAD $f(\vec{i}), \vec{i} \in D$, split the domain D into disjoint sub domains D_1, D_2 ... D_n such that $\forall i,j, D_i \cap D_j = \emptyset$, $D_1 \cup D_2 \cup \ldots \cup D_3 = D$, to form n RCSLMAD tiles. According to Corollary 1, $\forall i,j$ we have $L_f(D_i) \cap L_f(D_j) = \emptyset$, and $L_f(D_1) \cup L_f(D_2) \cup \ldots \cup L_f(D_n) = L_f(D)$.

RCSLMAD decomposition divides every dimension of D into contiguous regions to form smaller tiles. Each tile represents the region of a sub domain. RCSLMAD decomposition requires transformations to the program source code: for a parallel loop nest that contains sequential loops with array accesses in the body, the compiler tiles the array accesses by strip-mining the sequential loops and dividing parallel loop domains. These transformations split D into many sub-domains. By iterating over a tiled array access, the corresponding iteration sub-domain is traversed. RCSLMADs should be decomposed such that the number of array access elements accessed is equal to the number of threads in the thread block. Currently jit4OpenCL uses thread blocks of size 16*16 and therefore stripmines appropriate loops by 16. Jit4OpenCL only generates OpenCL code if no two RCSLMADs access overlapping memory locations.

The above analysis in jit4OpenCL introduces hazards when RCSLMADs overlap, and at least one contains write operations. This is not an issue with jit4GPU, and thus no data copies are needed, because it does not use local memory space. When the situation happens, jit4OpenCL abandons local memory optimization on the group of RCSLMADs and falls back to global memory accesses.

C. Ordering

Sorting all the elements in a RCSLMAD enables the compiler to generate local memory transfer code through memory-address mapping. Threads have their own, unique IDs. The compiler must know which elements in a RCSLMAD each thread should load to local memory. Creating an order among the memory locations referenced by a RCSLMAD results in a correspondence between threads and the elements of RCSLMADs.

1) Ordering Elements for Different Memory-Layer Mapping: Elements in $f(\vec{i}), \vec{i} \in D$ can be ordered because a RCSLMAD is a one-to-one mapping from vector value to scalar value. The order number of element $f(i_1, i_2, \ldots, i_d)$ is defined below. To simplify the expression, let $t_{r_j} = i_{r_j} - l_{r_j}$ for any r_j, and let $d_{r_k} = u_{r_k} - l_{r_k}$ for any r_k

$$Order(\vec{i}) \quad = \quad \sum_{j=1}^{r_m - 1} \left(t_{r_j} \left[\prod_{k=j+1}^{r_m} d_{r_k} \right] \right) + t_{r_m}$$

Conversely, given $Order(\vec{i})$, $f(\vec{i})$ and D, we can calculate $\vec{i}$. Assume that the bounds and strides of D are $(l_1 \ldots u_1, l_2 \ldots u_2, \ldots, l_d \ldots u_d)$ and $(s_1, s_2, \ldots, s_d)$. Let s_{r_k} be the k^{th} largest stride, then $\vec{i} = (i_1, i_2, \ldots, i_d)$ can be found by:

$$i_{r_m} = \begin{cases} Order(\vec{i}) & \text{if } m = 1 \\ Order(\vec{i}) \pmod{u_{r_m} - l_{r_m}} & \text{if } m \neq 1 \end{cases}$$

2) Ordering Threads: Within a thread group, every thread has its own order. This order is defined as follows: for an n-dimensional thread group, with dimension sizes of $l_i, i = 1, 2, \ldots, n$, then the thread ID in the group is:

$$ID_{thread} = \sum_{i=1}^{n} \left(d_i \times \prod_{j=i-1}^{i-1} l_j \right)$$

Where d_i is the i^{th} component of the thread local ID. The range of thread ID extends from 0 to the size of group minus one.

Threads are responsible for loading and storing RCSLMAD elements to local memory space. To simplify the analysis, jit4OpenCL creates a 1-to-1 mapping between each element in a RCSLMAD tile and each thread in a thread group. Therefore, the size and shape of a RCSLMAD tile is exactly the same as for a thread group. At runtime each thread loads/stores one element of the RCSLMAD tile leading the group of threads to collaborate to copy the RCSLMAD tile from global memory to local memory.

IV. The New Just-in-Time Compiler

The compilation framework uses unPython, a compiler that converts Python programs with annotations into either OpenMP code or an intermediate representation with calls to a JiT compiler for the generation of code to be executed in a GPU [4].

```
1 from unPython import prange
2 @type('ndarray[int32 1]','ndarray[int32 1]','int32',
      'int32')
3 def f(A,B,C,n):
4     sum = 0
5     for i in prange(N):
6         C[i] = A[i] + B[i]
7         sum += C[i]
8     return sum
```

Listing 3. Example of unPython Program Extension

Listing 3 shows a function that is written with the annotations required by unPython. The first three parameters — `A`, `B` and `C` — are 1-dimensional arrays with element data type `int32`; the last parameter is a variable of type `int32`; the return type is also `int32` (indicated by the last element in the annotation list). The for loop starting on line 5 is parallel, this is indicated by the use of a `prange` iterator instead of the usual `range` iterator for sequential loops.

Jit4OpenCL functionally replaces jit4GPU in the compilation framework to allow the generation of OpenCL code. Two important modifications in the original design of jit4GPU are the treatment of potentially overlapping RCSLMADs and the generation of a configuration grid. To prevent write-after-write hazards, jit4OpenCL identifies overlapping RCSLMADs and check if they contain at least one array write reference. If that is the case, jit4OpenCL generates code that accesses global memory, instead of local memory, to prevent a hazard. This is a coarse analysis because it is possible that two overlapping RCSLMADs that have array write references do not actually write to the same locations. However given that whole RCSLMADs are transferred between the different memory spaces, a finer-grain analysis might not yield much performance improvement because of the higher cost of transferring data at finer granularity.

A. Copying from Host Memory to Device Memory

A data transfer in OpenCL requires the initialization of device memory objects with specific sizes. The transfer is requested via the OpenCL API and is handled by OpenCL. Jit4OpenCL uses an analysis that was created for jit4GPU to first calculate the size of the memory region to be transferred, then it creates a device memory buffer object with that size. The memory buffer is contiguous on the device memory, therefore jit4OpenCL changes the strides in the LMAD description as needed.

B. Grid Configuration Generation

An OpenCL grid configuration determines the number, the geometry, and the size of each thread group prior to kernel execution. The size and geometry of a thread group may affect performance. Too few threads in a group leads to under-utilization of local memory space. Too many threads in a group would probably overflow the register file or would not execute because of insufficient local memory space to hold all the data. The grid configuration must be determined according to the program's array access pattern.

In jit4OpenCL the dimension for each thread group was determined empirically to be 16×16. If the group size requires more local memory space than the GPU can offer, a smaller groups are formed.

C. Kernel Code Generation

UnPython passes to jit4OpenCL: (a) a serialized abstract syntax tree (AST) of the source code in the form of a string; and (b) a set of LMADs describing the array-access references in the source code. Jit4OpenCL traverses the tree to generate target OpenCL kernel code.

Assume that there are p levels of parallel loops in a perfect loop nest with loop indexes $l_1, l_2, \ldots l_p$, where l_1 is the index of the outermost loop. The domains of the indexes are $d_1, d_2, \ldots d_p$ respectively. Jit4OpenCL generates a p-dimensional thread grid configuration (OpenCL and CUDA only support $p \leq 3$), and the i^{th} dimension's iteration domain starts from 0 to $d_i - 1$.

```
1 for x in gpurange(n):
2     for y in gpurange(n):
3         for k in range(n):
4         B[x,y] = A[x,k]*B[k,y]
```

Listing 4. Python Matrix Multiplication Computing Code

The Matrix Multiplication computation in Listing 4 demonstrates how jit4OpenCL generates an OpenCL grid configuration. There is a perfect parallel loop with depth 2, with both upper bounds equal n. The compiler

generates a 2-dimensional grid, with the domain of the first dimension (the x dimension) and the second dimension (the y dimension) both stretching from 0 to $n-1$, as shown in Figure 2, where a small square represents an issued thread. In this example, $n \times n$ threads are issued, and each thread has its own loop index. For example, the upper-left thread computes `B[0,0]`.

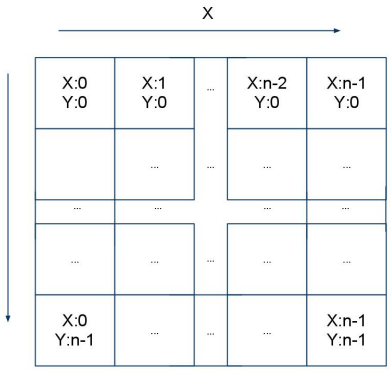

Figure 2. Matrix-multiplication Program Thread Grid Example

For serial loops that are in a loop nest body, jit4OpenCL performs the following transformations:

- strip mine the loop according to the RCSLMAD analysis for tiled array accesses;
- insert code, immediately before the execution of loop body, to transfer the tile accessed in the stripmined loop from off-chip global memory onto on-chip local memory space;
- replace global array accesses in the loop body with local memory accesses;
- insert code, immediately after the loop body, to transfer modified array tiles from local memory back to global memory;
- insert synchronization barriers right after the data transfer between local memory and global memory for data consistency.

Jit4OpenCL breaks an imperfect parallel loop nest into two or more smaller perfect parallel loop nests to enable the compiler to handle each of them. The transformation ensures the correctness of the program by guaranteeing that before the execution of an instruction, all its predecessor instructions are executed.

1) Local Memory Loading/Storing Code Generation: Given a set of RCSLMADs R that will be used in the loop body, the thread's global and local IDs, ID_g and ID_l respectively, the group size and the grid size in each dimension x (S_x^{group} and S_x^{grid}, respectively), the loop stripmining length L_{sm}, and the stripmined outer loop counter c, jit4OpenCL calculates the data elements referenced within the inner loop by changing the RCSLMAD's domain.

For each RCSLMAD $r \in R$, $r : f(\vec{i}), \vec{i} \in D$, the following address will be referred in the inner loop:

$$f(\vec{i}), \vec{i} \in D'$$

where for each dimension j in D', if j is a parallel dimension, then:

$$ID_g \times S_j^{group} \leq j < ID_g \times (S_j^{group} + 1) - 1$$

otherwise

$$L_{sm} \times c \leq j < L_{sm} \times (c+1) - 1$$

It is obvious that $f(\vec{i}), \vec{i} \in D'$ is also a RCSLMAD. To verify that, just replace D with D', and the RCSLMAD restrictions still stand. Moreover, D' covers the loop index domain of the inner loop for all the threads in the group. To utilize local memory, $f(\vec{i}), \vec{i} \in D'$ is transferred to local memory, where it gets compressed using the method stated in Theorem 2.

In the host code jit4OpenCL declares a dedicated, independent, local memory space for each $r \in R$ that is accessed in a serial loop in the code. The size of this space is exactly $L_f(D')$ where $r : f(\vec{i}), \vec{i} \in D$. Therefore, jit4OpenCL can generate local memory loading/storing code using the following method:

1) assign a unique order ID to the thread (see Section III-C);
2) Each thread i: calculates the global memory location of the i^{th} element in the processing RCSLMAD tile and the corresponding local memory location;
3) load/store the RCSLMAD tile element to/from the corresponding location of local memory space.

2) Redirecting Array Access References: After data is loaded into local memory, array accesses must be redirected to their copies in local memory space. Two changes must be made:

1) change of the array head pointers.
2) change of the subscripts.

To change the array pointer simply replace the array name with the one declared in local memory space. Changing of the subscripts requires jit4OpenCL to construct a new normalized RCSLMAD for the copy in local memory.

Given a RCSLMAD $r : f(\vec{i}), \vec{i} \in D'$ referring to an array access in global memory, where D' is the tiled domain that will be accessed within the stripmined serial loop by all the threads in the group, and the local memory with size larger or equal to

$L_f(D)$, the compiler constructs another RCSLMAD $g(\vec{i}), \vec{i} \in D''$, where D'' is a normalized representation of D', and $g(\vec{i}), \vec{i} \in D''$ points to memory locations in the local memory space. Normalization shifts the region of each dimension to start from 0 and leaves the region size unchanged. The compiler redirects the array access references by replacing the access to the first RCSLMAD with the new one.

V. Experimental Evaluation

This evaluation compares jit4OpenCL-generated code, code generated by jit4GPU. and highly optimized CPU code (for matrix multiplication) or OpenMP code generated by unPython for the other benchmarks. The main findings from this experimental evaluation are:

- As expected, the portability of jit4OpenCL comes with a performance cost: overall code generated by jit4GPU is faster on an AMD platform than code generated by jit4OpenCL.
- Jit4OpenCL also delivers performance improvements on a NVidia GPU.
- There is still room to improve the code generated by the combination of jit4OpenCL and the OpenCL compiler provided by vendors: currently they produce code twice as slow as optimized BLAS code on a CPU for matrix multiplication.
- For some benchmarks (`BS`, `MB`, and `MM`), the time to transfer data between CPU and GPU is a very significant fraction of the total execution time.
- Most of the benchmarks deliver slower than peak performance on GPU, which suggests that the generated code is bandwidth-bounded rather than computing-bounded. Newer GPU architectures, like NVidia Fermi, has increased memory bandwidth and a global memory cache, which will have a boost on the performance of the code generated by jit4OpenCL.

A. Experimental Platforms and Methodology

Experiments used a Phenom II X4 925 2.8 *GHz* with a Radeon 5850 GPU with 1GB of 1 *GHz* GDDR5. Catalyst 10.7 with OpenCL 1.1 update drivers were used. The software platform was Python 2.6 with NumPy version 1.2 running on 64-bit Linux 2.6.32 kernel. GCC 4.4 was used as C/C++ compiler with optimization flag -O2. All 4 cores were used when running CPU code. For OpenMP generation we used the flags -fopenmp in gcc. This machine is refereed to as the AMD machine.

To show that the OpenCL code generated by jit4OpenCL is portable, experiments are also reported for a desktop computer equipped with an NVidia GTX260 GPU, an Intel Core 2 Duo E5200 (2.5GHz), and 6GB DDR3 of memory. The machine is running on Ubuntu 9.10 32-bit using GCC 4.4 and NVidia Computing Software Development Kit (SDK) 3.2 Beta. This is the NVidia machine. For each result presented in the tables, twenty runs of the benchmark were performed. The mean and a 95% confidence interval are presented.

B. Benchmarks

Five benchmarks are used in this evaluation. `CP` computes of the Columbic potential at each point in a planar grid. `CP` is inspired by the CP benchmark in the Parboil suite and is a highly parallel kernel. `NB` is a kernel of an N-body simulation. It computes the distances between a point and every other point in a 3D space. `NB` performs $9n^4$ floating-point operations for a problem of size n. `BS` computes multiple Black Sholes formulas, used for option pricing in computational finance, in parallel. `BS` is derived from a Brook+ implementation distributed by AMD with the Stream SDK. `MB` computes an approximation of the Mandelbrot set of points in the complex plane. For each point in a 2D grid, `MB` checks if the point belongs to the Mandelbrot set or not. `MM` is the multiplication of 2 $N \times N$ single-precision dense contiguous matrices.

C. Results

The top rows in Table I show the execution times (in seconds) and the bottom half shows the relative speedups between each compiler and the CPU and the speedup of jit4GPU over jit4OpenCL. For `MM` the comparison is with the highly optimized BLAS implementation of matrix multiplication. Both JIT compilers outperform the OpenMP code running in the multi-core CPUs, but the performance of the code generated by jit4GPU is significantly better, except for `MB` where the jit4OpenCL performance is almost double that of the jit4GPU performance. Because of the intermediate translation to OpenCL and the additional compilation step to convert the generated OpenCL code into native code, jit4OpenCL has significant additional overhead in comparison with jit4GPU. In short-running benchmarks this overhead impacts the performance comparison. For instance, the kernel (GPU execution only) for `MB` produced by jit4OpenCL is six times faster than the kernel generated by jit4GPU.

The main goal for the re-targeting of the compiler to generate OpenCL code is to deliver a compilation infrastructure that can generate code for multiple platforms. Table II reports the performance of the code generated by jit4OpenCL in the NVidia GPU described above. Given that this is a 2-core CPU (versus the quad-core CPU in the AMD machine), it is not surprising

Benchmark	CP	NB	BS	MB	MM
Input Size	512	768	4096	4096	4096
jit4GPU	**0.56** ± 0	**6.27** ± 0.01	**0.52** ± 0.01	**0.72** ± 0.01	**0.58** ± 0.01
jit4OpenCL	**2.32** ± 0	**9.16** ± 0.01	**3.94** ± 0.50	**0.41** ± 0	**14.30** ± 0.01
CPU	**69.13** ± 0.07	**401.49** ± 0.71	**4.16** ± 0.01	**2.22** ± 0	**2.01** ± 0.01
jit4GPU × CPU	**124**	**64**	**8.1**	**3.1**	**3.4**
jit4OpenCL × CPU	**30**	**44**	**1.1**	**5.4**	**0.14**
jit4GPU × jit4OpenCL	**4.2**	**1.5**	**7.6**	**0.57**	**24**

Table I
COMPARISON BETWEEN JIT4GPU, JIT4OPENCL AND CPU IN THE AMD MACHINE

Benchmark	CP	NB	BS	MB	MM
jit4OpenCL	**1.19** ± 0	**12.49** ± 0.03	**1.35** ± 0.01	**0.67** ± 0	**2.49** ± 0.01
CPU	**196** ± 0	**672** ± 0	**7.99** ± 0.07	**5.05** ± 0	**4.64** ± 0
jit4OpenCL × CPU	**166**	**54**	**5.9**	**7.6**	**1.87**

Table II
COMPARISON BETWEEN JIT4OPENCL AND CPU CODE IN THE NVIDIA MACHINE

that the speedups are higher. Nonetheless, these results provides indications that the generated OpenCL code delivers performance in multiple platforms.

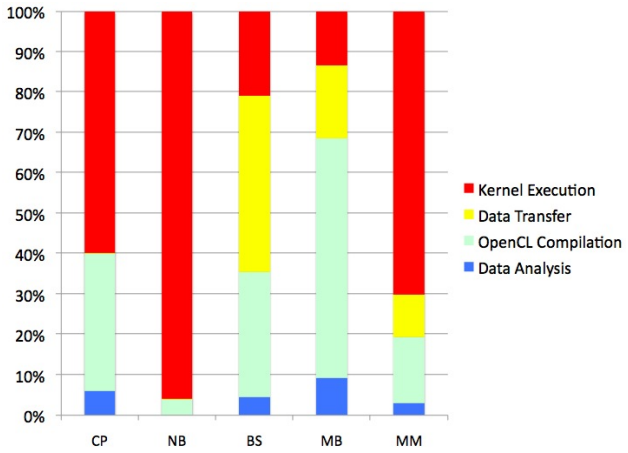

Figure 3. Breakdown of Execution Time in NVidia machine.

The graph in Figure 3 shows the contribution of the different portions of the execution to the execution time of the benchmarks in the NVidia machine. The graph shows that both data transfer and the OpenCL compilation are significant contributors to the execution time in this machine.

D. Comparing with Hand-Written OpenCL Code

We also want to see how the code generated automatically by jit4GPU and by jit4OpenCL from Python/Numpy code perform compared with these hand-written versions of the benchmarks. As the run-time results shown in Figure 4 indicate, there is still significant room for improvement in the speed of the code generated by both of these compilers. For this comparison the OpenCL versions for `nbody` and the columbic potential (`cp`) benchmarks were extracted from the parboil OpenCL suite, matrix multiplication is from the AMD SDK, and Mandelbrot `mandel` and Black Sholes (`BS`) were hand-written by us. For this comparison, all three versions of the code were run on an a machine equiped with two Intel Xeon E5405, with 16G of memory, and with a GPU AMD FirePro 9800v and with AMD SDK 2.3.

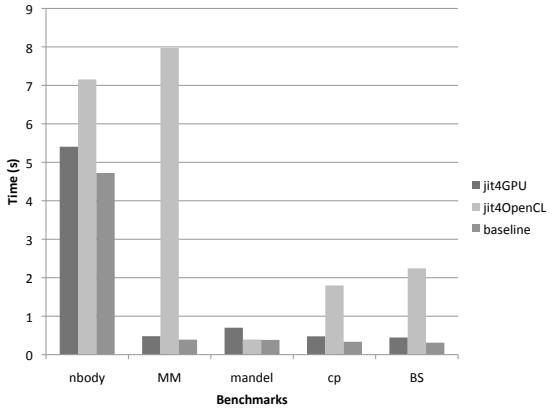

Figure 4. Comparison with Hand-Written OpenCL Code.

VI. RELATED WORK

Paek *et al.* proposed LMADs as a linear model that describes the memory locations accessed by an array in a loop nest [7]. LMADs are used to identify coalescing accesses, interleaving accesses, contiguous

accesses, and to privatize arrays for parallel shared-memory machines. We use a subset of LMAD, the RCSLMAD, to discover and identify memory locations of accessed array elements, to identify memory locations accessed in a thread group, and to do the array privatization on on-chip shared memory. [3].

The LMAD-based analysis extended in this paper was created for jit4GPU prior to the description of related, and more limited, analyses by Yang *et al.* [9]. Jablin *et al.*'s automatic communication management specifically targets NVidia GPUs and the CUDA intermediate representation [10].

Ryoo *et al.* discussed the possible optimization space for CUDA programs on NVidia's Tesla Architecture [6], [11], [12]. They argued that to achieve good performance using CUDA, programmers have to strike the right balance between the number of simultaneously active threads, and thread resource usage, including register usage, shared memory usage, and the global memory bandwidth usage. One of their important discoveries is that, as long as register files are not overflown, increasing the amount of simultaneously active threads will lead to better performance. In our work, the automatic grid configuration generation module follows the guidelines described in [6].

Volkov *et al.* analyzed the access latency to each different level of memory and the PTX code (the intermediate byte code format for CUDA programs) generated by NVCC compiler, to devise an implementation pattern that unrolls loops aggressively and interleaves memory transferring with computing [13]. They argue that fewer active threads leads to more register file space available for each thread, and less thread-scheduling overhead. Their tuning of the code is most suitable for manual optimization and is difficult to integrate in a compiler design because it requires detailed analysis of the cost of accessing the various levels of the memory architecture. Moreover different GPUs may result in different performance.

VII. Conclusion

The combination of Python and NumPy is an increasingly popular choice of language for numerical intensive applications. This compiler-construction paper describes the re-targeting of jit4GPU to generate code for OpenCL to create a compilation path for the execution of these applications on any architecture that implements the OpenCL open standard. The experimental evaluation demonstrated the use of this compilation framework in two different architectures. As expected, the portability pays a performance cost, but this initial evaluation indicates that this cost is reasonable and that we still observe significant performance improvement in the GPU-equipped machines for certain applications. Moreover, although the automatically generated code boosts programmer's productivity, it still cannot compete with hand-written OpenCL codes.

References

[1] P. Greenfield, "Reaching for the stars with Python," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 38–40, May/June 2007.

[2] T. E. Oliphant, "Python for scientific computing," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 10–20, May/June 2007.

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

[4] R. Garg, "A compiler for parallel execution of numerical Python programms on graphic processing units," Master's thesis, University of Alberta, 2009.

[5] A. Munshi, "The OpenCL specification version 1.1," *Khronos OpenCL Working Group*, 2010.

[6] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, and W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Symposium on Principles and practice of parallel programming PPoPP*, 2008, pp. 73–82.

[7] Y. Paek, J. Hoeflinger, and D. Padua, "Simplification of array access patterns for compiler optimizations," in *Programming Language Design and Implementation (PLDI)*, Montreal, QC, Canada, 1998, p. 71.

[8] X. Li, "JiT4OpenCL: A compiler from python to opencl," Master's thesis, University of Alberta, Edmonton, AB, Canada, August 2010.

[9] 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)*, Toronto, ON, Canada, 2010, pp. 86–97.

[10] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic cpu-gpu communucation management and optimization," in *PLDI*, San Jose, CA, June 2011, pp. 142–151.

[11] S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S. Ueng, and W. Hwu, "Program optimization study on a 128-core GPU," in *The First Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[12] S. Ryoo, C. Rodrigues, S. Stone, J. Stratton, S. Ueng, S. Baghsorkhi, and W. Hwu, "Program optimization carving for GPU computing," *Journal of Parallel and Distr. Computing*, vol. 68, no. 10, pp. 1389–1401, 2008.

[13] V. Volkov and J. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Conference on Supercomputing*, Austin, TX, USA, 2008, pp. 1–11.