A Mapping Path for multi-GPGPU Accelerated Computers from a Portable High Level Programming Abstraction

Allen Leung Nicolas Vasilache Benoît Meister Muthu Baskaran
David Wohlford Cédric Bastoul Richard Lethin
Reservoir Labs
803 Broadway, Suite 803
New York, NY 10012
{leunga,vasilache,meister,baskaran,wohlford,bastoul,lethin}@reservoir.com

ABSTRACT

Programmers for GPGPU face rapidly changing substrate of programming abstractions, execution models, and hardware implementations. It has been established, through numerous demonstrations for particular conjunctions of application kernel, programming languages, and GPU hardware instance, that it is possible to achieve significant improvements in the price/performance and energy/performance over general purpose processors. But these demonstrations are each the result of significant dedicated programmer labor, which is likely to be duplicated for each new GPU hardware architecture to achieve performance portability.

This paper discusses the implementation, in the R-Stream compiler, of a source to source mapping pathway from a high-level, textbook-style algorithm expression method in ANSI C, to multi-GPGPU accelerated computers. The compiler performs hierarchical decomposition and parallelization of the algorithm between and across host, multiple GPGPUs, and within-GPU. The semantic transformations are expressed within the polyhedral model, including optimization of integrated parallelization, locality, and contiguity tradeoffs. Hierarchical tiling is performed. Communication and synchronizations operations at multiple levels are generated automatically. The resulting mapping is currently emitted in the CUDA programming language.

The GPU backend adds to the range of hardware and accelerator targets for R-Stream and indicates the potential for performance portability of single sources across multiple hardware targets.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors – Code Generation, Compilers, Optimization

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GPGPU '10 March 14, 2010, Pittsburgh, PA, USA
Copyright 2010 ACM 978-1-60558-935-0/10/03 ...$10.00.

General Terms

Algorithms, Design, Performance

Keywords

Parallelization, Compiler Optimziation, Polyhedral Model, GPGPU, CUDA, Automatic Translation

1. INTRODUCTION

Modern General Purpose Graphics Processing Units (GPGPUs) are massively parallel multiprocessors capable of high flop performance and bandwidth. While GPGPU programming is eased by the introduction of higher level data parallel languages such as CUDA [16], maximizing the performance of an application still requires the precise balancing of many different types of constraints in the GPU architecture, including the utilization of SIMD and coarse-grained data parallelism, and the management of the memory hierarchy. This problem extends to the system level for anticipated multi-GPGPU accelerated hosts. While programming such systems "by hand" has been demonstrated for a range of applications, this is a difficult and costly endeavor; likely one to be revisited to allow the application to port to rapidly arriving new generations and configurations of GPUs and programming abstractions that change the optimization tradeoffs.

Previously we have demonstrated the R-Streamcompiler [14, 12] and facilities to target processors and accelerators such as SMP, TILEPro, Cell, and ClearSpeed. This paper discusses the adaptation of this compiler to produce CUDA for high-performance execution on GPGPU.

This work is distinguished from other work on mapping to GPGPU as follows. The HMPP compiler from CAPS requires detailed pragma directives, determined by the programmer, to explicitly direct the parallelization of the code and the management of the memory between host and GPU. Our approach is to find these automatically to eliminate this burden on the programmer. The PGI Accelerator performs synthesis and automatic translation of C and FORTRAN into GPU code, but requires the programmer to write kernels in a literal manner where the loop indices are essentially syntactically translated into the threadIdx and blockIdx constructs for the GPU; the approach is limited to kernels where a stride-1 array access is apparent. Our approach uses a more modern semantically based compiler framework that allows for more powerful parallelization and mapping

capability. A paper [4] will appear (co authored by one of the authors of this paper, Muthu Baskaran) describing a mapping process to GPGPU using a modern framework; our work is also targeted additionally to the system level mapping between host/GPU and with multiple GPUs, with significant attention to automatic generation of communication. Other differences with [4] are noted herein.

This paper gives a snapshot of rapidly evolving capability; more details on results and performance will be provided at the workshop.

2. THE R-STREAM COMPILER

The basis of our mapping tool is Reservoir Labs' proprietary compiler, R-Stream. DARPA funded development of R-Stream by Reservoir between 2003 and 2007 in the Polymorphous Computing Architectures (PCA) Program [12].

The flow for the R-Stream high-level compiler is shown in Fig. 1. An EDG-based front end reads the input C program (1), and translates it into a static single assignment intermediate form, where it is subject to scalar optimizations (2) and (3). Care is taken to translate the C types into a sound type system, and to preserve them through the optimizations, for later syntax reconstruction in terms of the original source's types.

The part of the program to be mapped is then "raised" (4) into the geometric form based on parametric multidimensional polyhedra for iteration spaces, array access functions, and dependences.

In this form, the program is represented as a Generalized Dependence Graph (GDG) over statements, where the nodes represent the program's statements and are decorated by the iteration spaces and array access functions (polyhedra), and the inter-statement dependences are represented as labeling of the edges.

Kernels that meet certain criteria, "mappable functions" that can be incorporated into the model, are raised. No new syntactic keywords or forms in the C program are involved in this criteria for raising; R-Stream raises various forms of loops, pointer-based and array based memory references, based on their semantic compatibility with the model.

We note here that the type of C that R-Stream optimizes best is "textbook" sequential C, e.g., like a Givens QR decomposition in a linear algebra text. If the programmer "optimizes" their C (e.g. with user-defined memory management or by linearizing multidimensional arrays), those typically interfere with R-Stream's raising and optimizations.

These mappable functions are then subjected to an optimization procedure which (for example) identifies parallelism, tiles the iteration spaces of the statements into groups called "tasks", places the tasks to processing elements (PEs, which correspond to the cores of a multicore processor or the nodes of a cluster), sets up improvements to locality, generates communications, adds synchronization and organizes transfers of control flow between a master execution thread and the mapped client threads. The GDG is used as the sole representation in all these mapping steps ((5) and (6)).

After this process, the GDG is lowered (7) back to the scalar IR, through a process of "polyhedral scanning" that generates loops corresponding to the transformed GDG. This scalar IR of the mapped program is then subject to further scalar optimization (8) and (9), for example to clean up synthesized array access functions. The resulting scalar IR is then pretty-printed (10) via syntax reconstruction to generate the resulting code in C form, but mapped in the sense that it is parallelized, scheduled, and has explicit communication and memory allocation.

Code that was not mappable is emitted as part of the master execution thread, thus the R-Stream compiler partitions across the units of a heterogeneous processor target. From this point, the code is compiled by a "low-level compiler" (LLC), which performs relatively conventional steps of scalar code generation for the host and PEs.

R-Stream supports generating code mapped for STI Cell, SMP with OpenMP directives, ClearSpeed, FPGA-accelerated targets, and Tilera. Here we are presenting our first port to GPGPUs, which targets the CUDA execution model.

The breadth of targets R-Stream supports illustrates the benefit of programming in a high-level, architecture-independent manner, and using this compiler technology. The computation choreography needed for efficient execution on GPU can be constructed automatically from the simple sequential C source, and this can be automatically rendered in the CUDA language. The computation choreography for other architecture targets (SMP, etc.) can also be generated from the *same* sequential C source, and automatically rendered in the corresponding target-specific source (OpenMP, etc.). Thus the program abstraction is *portable*.

This mapping process is driven by a declarative machine model for the architectures, using an XML syntax for describing hierarchical heterogeneous accelerated architectures [13]. The contents of the machine model provide the overall architecture topology as well as performance and capacity parameters, a formalization and implementation of a Kuck diagram [11]. For example, it is possible to describe a shared memory parallel computer with many FPGA accelerators, such as SGI Altix 4700 with RASC FPGA boards. It also can describe the complex execution model for GPGPU, in terms of the complex distributed memories and parallel execution engines [10]; it can also describe clusters of traditional MPI connected nodes with attached GPGPU accelerators. While this language (and a graphical machine model browser for it) are human-understandable, they are also tied to the mapping algorithms, providing parameters for different optimizations and driving the mapping tactics. The final code is emitted for the target in terms of the task invocations, explicit communications, and synchronizations among parent and child PEs and among the PEs at any level.

This paper describes our first adaptation of the mapping process to automatically support GPGPUs as a target, with the temporary restriction that data has to be already present in the device memory.

The next section presents details of the mapping process.

3. BASIC MAPPING STRATEGY

There are two key aspects in producing good CUDA mapping: *(i)* assigning the available set of parallelism to blocks and threads, and *(ii)* managing the memory hierarchies efficiently.

The problems are currently handled by our mapper in the following sub-phases:

1. Optimize a weighted tradeoff between parallelism, locality and contiguity of memory references via *affine scheduling*. This step exposes as much parallelism as can be exploited in the target machine model.
2. Use **tiling** (aka **blocking**) to divide the iteration space

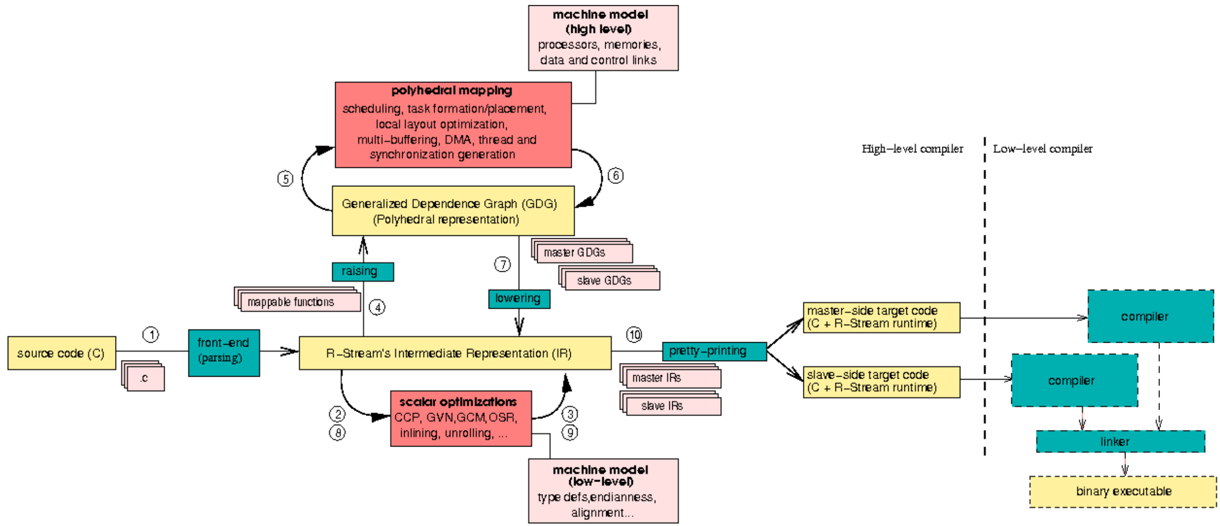

Figure 1: R-Stream**compiler flow**

into tiles, such that each tile fits within the constraints of a thread block.

3. Promote variables in device memory to shared memory via **shared memory promotion**, and produce optimized transfers from global to shared memory.

4. Compute a mapping from any loop iteration to a set of block and thread IDs. We call this **Block and thread placement**. Blocks and thread IDs are distributed across independent iterations, so they can be executed in parallel. The CUDA programing model also imposes that these IDs are constrained to be within a $[0, n)$ interval.

5. Insert synchronizations at block and thread levels. Since CUDA does not natively support block synchronization, we have implemented a software barrier in the style of [4].

6. Apply more optimizations that make the CUDA code that R-Stream produces a better fit with the programming style expected by `nvcc`.

All these steps are carried inside a single powerful framework based on the polyhedral model [1, 9]. The nvcc-specific optimizations include traditional optimizations such as loop unrolling, loop fusion and privatization. In addition to these, two basic transformations may be applied to make the produced code match the CUDA programming model.

First, two parallel thread loops may be merged into one by inserting a thread synchronization, i.e., from

```
doall_threads (j = 0; j < T; j++) {
  S1;
}
doall_threads (j = 0; j < T; j++) {
  S2;
}
```

into

```
doall_threads (j = 0; j < T; j++) {
   S1;
   __syncthreads();
   S2;
}
```

Generalizing the above, we can also interchange a sequential loop with a parallel thread loops, i.e., from

```
for (i = 0; i < N; i++) {
  doall_threads (j = 0; j < T; j++) {
     S;
  }
}
```

to

```
doall_threads (j = 0; j < T; j++) {
  for (i = 0; i < N; i++) {
     S;
     __syncthreads();
  }
}
```

Note the insertion of `__syncthreads` is not always necessary depending whether or not the doall j loop can be legally placed at the outermost level. In our context such considerations disappear as our affine scheduling phase guarantees the only optimal restructuring of the loop is considered. In other words, if it was legal *and profitable* to remove the `__syncthreads`, the schedule considered would have been outer parallel in the first place. It is important to realize optimality is related to a model. Whether the model is derived statically or dynamically and whether it actually translates into performance is a fascinating topic outside the scope of this paper. We now go over the mapper phases involved in more detail in the following sections.

3.1 Affine scheduling

The first step of the mapping process is to expose all the available parallelism in the program, including both coarse-grained and fine-grained parallelism. Generally speaking,

our strategy consists of transforming coarse-grained parallelism into the thread and block parallelism in a CUDA kernel, and fine-grained parallelism into SIMD parallelism within a thread block.

We use an improvement of [7] to perform fusion and parallelism extraction while obtaining contiguous accesses through some of the array references. Important strengths of our algorithm as opposed to [7, 4] include that (1) it balances fusion, parallelism and contiguity of accesses, (2) it ensures that the degree of parallelism is not sacrificed when loops are fused and (3) it is applied as a single mapper phase which makes the algorithm very suitable for iterative optimization and auto-tuning. Our algorithm is based on a complex integer programming problem that is outside the scope of this paper.

In our running example, the available amount of parallelism is easy to expose, and the resulting loop nests are as follows:

```
doall (i = 0; i < 4096; i++)
  doall (j = 0; j < 4096; j++) {
    C[i][j] = 0;
    reduction (k = 0; k < 4096; k++)
      C[i][j] = C[i][j] + A[i][k] * B[k][j];
  }
```

Note that our convention is to use the keywords `doall` and `reduction` to indicate potential parallelism rather than actual usage of parallelism. In this case, `doall` indicates that a loop may be executed in a data parallel manner, while `reduction` indicates that a loop is a reduction, i.e., any sequential order of execution is legal, including tree-based decomposition of the computations.

3.2 Tiling

The next step of our mapping is to perform tiling. The present tiling algorithm is guaranteed to choose tile sizes that satisfy the following criteria:

1. The data footprint of the tile does not exceed the size of the shared memory.
2. The tile size balances the amount of computation and communication (among tiles).

The first constraint ensures that all the memory storage within one tile after tiling fits within the local memory of the GPU. Our algorithm for tile size selection currently maximizes the occupancy of a level of shared memory by maximing the amount of spatial and temporal reuse along the loop dimensions that are marked permutable. The underlying mathematical framework uses Ehrhart polynomial, which are multi-variate periodic polynomials [8, 2, 19, 15]. In general, when targeting architectures with asynchronous memory communications such as Cell, tiling can be tuned to use a fraction of the shared memory. Each subportion of the shared memory is then used to create a small buffer on which we apply multi-buffering to hide communication latencies with computations [14].

For CUDA, we choose not to enable multi-buffering because latency hiding is performed automatically in hardware by the dynamic scheduler. This happens as soon as the number of virtual threads and blocks exceeds the number of physical resources. Compared to sotware managed multi-buffering, hardware manaed latency hiding does not require extra code. Additionally, there is not need to create tile with sizes that fit in a fraction of the memory. One tile can use all the local store. This results in improved reuse inside the single CUDA buffer.

The drawbacks are less immediate as there are multiple tensions between tile sizes, exploitable parallelism, load balancing properties, number of usable registers per thread, possibility or impossibility to privatize arrays into registers, register pressure and spilling to device memory. We are in the process of experimenting and deriving models for these very specific hardware features. One of the difficulties is that tiling needs to communicate with multiple phases of the compiler to make the best decisions. R-Stream has enough flexibility to allow complex decision interplays between phases because every single phase is implemented in the polyhedral model representation.

In our running example, a tile size of $32 \times 32 \times 32$ is computed, and the resulting loop nests loop is:

```
doall (i = 0; i <= 127; i++) {
  doall (j = 0; j <= 127; j++) {
    doall (k = 32 * i; k <= 32 * i + 31; k++)
      doall (l = 32 * j; l <= 32 * j + 31; l++)
        C[k][l] = 0;
    reduction_for (k = 0; k <= 127; k++)
      doall (l = 32 * i; l <= 32 * i + 31; l++)
        doall (m = 32 * j; m <= 32 * j + 31; m++)
          reduction_for (n = 32 * k;
                         n <= 32 * k + 31; n++)
            C[l][m] = C[l][m] + A[l][n] * B[n][m];
  }
}
```

While R-Stream computes tile sizes that respect the criteria above, it is also possible for the user to tune his program by exploring different tile sizes using command-line options.

4. CUDA SPECIFIC MAPPING PHASES

Transforming a standard tiled parallel SMP code into a CUDA kernel generally involves sequences of non-trivial "orthogonal" loop and data transformations, including loop fusion, fission, interchange, stripmining, and data permutation. The first step of this process is block and thread placement, i.e., determining the set of loop dimensions to be used for block and thread dimensions.

4.1 Global memory coalescing

GPUs implement **global memory coalescing**, whereby aligned array accesses from device memory that exhibit a stride-1 contiguity and that are assigned to adjacent threads are merged into a single memory transaction. Taking advantage of this hardware feature is necessary to obtain good performance through improved memory transfer rate.

However, memory coalescing interacts with data layout and thread placement in non-trivial ways. For example, consider the following parallel loop nests with one single reference of interest:

```
doall (i = 0; i < 32; i++)
  doall (j = 0; j < 32; j++)
    ... A[i,j] ...
```

To optimize the reference so that each successive thread accesses adjacent elements of the data array, we have to

interchange loops i and j, and destinate the j loop as the thread dimension. The resulting transformation is shown below:

```
doall_threads (j = 0; j < 32; j++)
  for (i = 0; i < 32; i++)
     ... A[i,j] ...
```

Similarly, consider the following parallel loop nests with also one single reference of interest:

```
doall (i = 0; i < 32; i++)
  doall (j = 0; j < 32; j++)
    ... A[i,2*j] ...
```

Data dimension 1 of A, accessed via a non-stride 1 access (2*j), cannot be optimized via memory coalescing. To ensure global memory coalescing, the only possibility is to transpose the data layout of A to make the loop i as the thread dimension:

```
doall_threads (i = 0; i < 32; i++)
  for (j = 0; j < 32; j++)
    ... A_t[2*j,i] ...
```

Data layout transformations affect the whole layout of the program and must be considered carefully. They are most profitably introduced at communication boundaries between different types of memories. In the current version of R-Stream, we do not perform data layout transformations on an array residing in device memory. *We perform data relayouts on transferred local copies of the arrays, at the time of the transfer.* This mechanism can easily be extended to handle relayouts in device memory at the time of copy from host to device memory.

The previous simple examples show that optimizing for memory coalescing is non-trivial, and doing it by hand is a cumbersome task best left to a compiler. To solve this problem, we have devised a combined loop and data-transformation optimization that performs these tasks in a unified manner:

1. Compute a schedule transformation optimizing a weighted cost of the contiguity of the array references accessed by the computation kernel while keeping the same amount of parallelism and locality as originally found by our initial scheduling phase. The details of this integer linear programming based optimization are well beyond the scope of this paper and will be published at a later time.

2. Apply the new schedule to the program that maximizes contiguity along well-chosen dimensions.

3. Assign the proper loop iterations to the thread and block dimensions of a GPU kernel via strip-mining.

4. Transform imperfect loop nests into a perfectly nested loop nests CUDA kernel via strip-mining, fusion and loop interchanges.

5. Arrays whose references cannot be profitably coalesced are flagged for memory promotion in shared memory, if they exhibit enough reuse.

It is important to note that deciding whether an array should be promoted to shared memory is not a trivial process. If some, but not all, of the references to the array can be coalesced, there is a need for a model that analyzes (1) the tradeoff between wasted uncoalesced global memory instructions, (2) the amount of required shared memory to store the promoted copy of the array, (3) the amount and expected benefit of reuse and (4) the impact on register pressure of introducing explicit copy operations.

4.2 Shared memory promotion

The shared memory promotion step promotes the memory used within a tile to shared memory, when our compiler deems it profitable to do so. Copies between the device memory and shared memory are introduced by our communication generation phase. Communication generation is invoked when there is a need (whether it arises from programmability or profitability) to explicitly transfer data between different memories. Current examples when R-Stream invokes communication generation include (1) generation of prefetch instructions or explicit copies that benefit from hardware prefetches on shared memory machines, (2) generation of DMA instructions for the Cell BE architecture which requires SPUs to process data in their local scratchpad memory, (3) copy uncoalesced global arrays that exhibit reuse to shared memory in CUDA.

On a standard matrix-multiply example, the result of memory promotion and communication generation give the following pseudo-code:

```
__shared__ float C_l[32][32];
__shared__ float A_l[32][32];
__shared__ float B_l[32][32];
__device__ float A[4096][4096];
__device__ float B[4096][4096];
__device__ float C[4096][4096];
doall (i = 0; i <= 127; i++) {
  doall (j = 0; j <= 127; j++) {
    doall (k = 0; k <= 31; k++)
      doall (l = 0; l <= 31; l++)
        C_l[k][l] = 0;
    reduction_for (k = 0; k <= 127; k++) {
      doall (l = 0; l <= 31; l++)
        doall (m = 0; m <= 31; m++)
          B_l[l][m] = B[32 * k + l][32 * j + m];
      doall (l = 0; l <= 31; l++)
        doall (m = 0; m <= 31; m++)
          A_l[l][m] = A[32 * i + l][32 * k + m];
      doall (l = 0; l <= 31; l++)
        doall (m = 0; m <= 31; m++)
          reduction_for (n = 0; n <= 31; n++)
            C_l[l][m] += A_l[l][n] * B_l[n][m];
    }
    doall (l = 0; l <= 31; l++)
      doall (m = 0; m <= 31; m++)
        C[32 * i + l][32 * j + m] = C_l[l,m];
  }
}
```

In the simple case of matrix-multiply, R-Stream deems all arrays worthy of being promoted to shared memory. This is due to the large amount of reuse in the matrix-multiply kernel. In our mapping, each transferred memory element is reused 32 times.

4.3 Memory coalescing analysis

We now describe the model we use to derive memory coalescing transformations. Given an array reference $A[f(x)]$, we define a coalescing tuple (A, d, j, w) as follows:

- A is an array,
- d is an array dimension of A, indexed from 0.
- j is a potential thread loop dimension (i.e., it must be a parallel intra-tile loop dimension), and
- w is the weight, which measures how much benefit there is if the given reference is coalesced. Intuitively, a coalescing tuple (A, d, j, w) for a reference $A[f(x)]$ means that if data dimension d of A is made the right-most data dimension [1] and if j is made the outer-most thread dimension, we gain a performance benefit of w in the program.

Currently, the weight w is computed via the following static cost model:

1. An estimate of the number of total iterations in the intra-tile loop is used as the initial estimate of w.
2. A reference from device memory is slower to execute than from shared memory, so we also scale w by the relative speed of the load/store.

Consider this loop nest:

```
doall (i = 0; i < P; i++)
  doall (j = 0; j < Q; j++)
    doall (k = 0; k < R; k++)
      ... A[i,j] + A[i,k] + A[i+j,32*i+j]
```

The tuples that we produce for all the references are as follows: For `A[i,j]`, the tuples are $[(A, 0, i, PQR), (A, 1, j, PQR)]$. For `A[i,k]`, the tuples are $[(A, 0, i, PQR), (A, 1, k, PQR)]$. For `A[i+j,32*i+j]`, the tuples are $[]$, i.e., no memory coalescing is possible. The tuples for the statement S can be computed by merging the tuples for its references. In this case we have: $[(A, 0, i, 2PQR), (A, 1, j, PQR), (A, 0, k, PQR)]$

Our optimizing algorithm, whose description is left for future publication, chooses loop `i` as the most beneficial outermost thread dimension. This results in and transpose `A`. The resulting code is:

```
doall_threads (i = 0; i < P; i++)
  for (j = 0; j < Q; j++)
    for (k = 0; k < R; k++)
      ... A_t[j,i] + A_t[k,i] + A_t[32*i+j,i+j]
```

Note that the first two references have been optimized to take advantage of coalescing.

4.4 Integrating synchronization costs

The weight computation we presented above is somewhat inaccurate in the presence of non-parallel loop dimensions. This is because using an inner doall loop as a thread dimension can increase the amount of synchronization that we require in the final CUDA kernel. For example, suppose we have following loop nests with two parallel and one interleaved sequential loop dimensions:

```
doall (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++)
    doall (k = 0; k < 128; k++)
      S;
```

[1] Assuming C-style data layout.

If the dimension `i` is chosen as the thread dimension, then no synchronization code is required:

```
doall_threads (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++)
    for (k = 0; k < 128; k++)
      S;
```

On the other hand, we may prefer to choose loop `k` as the thread dimension, because it allows us to use a higher number of threads. If this is the case, then a `__syncthreads` call must be inserted in the output code to preserve the semantics of the original program. This argument is valid in the same context as the discussion of section 3: it is assumed the loop k cannot be legally used as an outer doall loop. If this assumption was wrong and the k loop was indeed chosen as the outer thread loop, it means the affine scheduling phase should have made the k loop outer parallel. If such a case happens, we see it as a good indication that the cost model used for parallelism maximization should be improved. The resulting pseudocode is:

```
doall_threads (k = 0; k < 128; k++)
  for (i = 0; i < 16; i++)
    for (j = 0; j < 16; j++) {
       S;
       __syncthreads();
    }
```

Furthermore, since the loop `i` was originally a doall loop, we can also sink it below the loop `j`, and hoist the thread synchronization call. This results in the following improved code with fewer number of synchronizations:

```
doall_threads (k = 0; k < 128; k++)
  for (j = 0; j < 16; j++) {
    for (i = 0; i < 16; i++)
      S;
    __syncthreads();
  }
```

In general, a thread synchronization penalty is deducted from the weight of a coalescing tuple. Generally speaking, thread synchronization is required if a loop dimension is nested under a sequential loop within a tile. The total penalty of the synchronization is proportional to the trip count of the sequential loops, which is an estimate of the minimal amount of thread synchronization calls that the program has to execute per block.

4.5 Computing memory coalescing for arbitrary loop nests

We now generalize the previous analysis to imperfectly nested loops with multiple statements. Suppose we have two coalescing tuples (A, d, j, w) and (A', d', j', w') for statements S and S' respectively (S could be equal to S'.) We say that the two tuples are **compatible** if

1. If $A = A'$, then $d = d'$. Intuitively, this means that the two tuples must describe compatible coalescings.
2. If j-loop in S and j'-loop in S' are nested under some common loop nests, and if the j-loop or the j'-loop belongs to this common part, then $j = j'$.

The meaning of the second condition can best be explained via an example. Consider the following loop nests:

```
doall (i = 0; i < 32; i++) {
  doall (j = 0; j < 32; j++)
    ... A[i,j] ...; // S1
  doall (k = 0; k < 32; k++)
    doall (l = 0; l < 32; k++)
      ... A[i,k] + B[k,l] ...; // S2
}
```

Statement S1 and S2 have the i-loop in common. The condition (2) states that if we choose the i-loop for a thread dimension of S1, then we have also use it for the thread dimension of S2. On the other hand, if we choose the j-loop for the thread dimension for S1, then we have the freedom to use the k- or l-loop for the thread dimension of S2.

Given the definition of compatibility, the task of optimizing memory coalescing can be stated simply as follows: given a set of coalescing tuples T, find a compatible subset of T, T_{opt}, such that the weight of T_{opt} is maximized.

4.6 Choosing remaining thread and block dimensions

Figure 2 shows the algorithm to determine which loops should be mapped to the threads and block dimensions. Currently, we simply find one dimension at a time, starting from the first thread dimension. During the selection of the first thread dimension, memory coalescing optimization is applicable.

for i = 1 ... 3 **do**
  if i = 1 **then** enable coalescing optimization **end if**
  Find a new thread dimension from the intra-tile loops
  if too many threads are used **then**
    break;
  end if
end for
for i = 1 ... 3 **do**
  Find a new block dimension from the inter-tile loops
  if too many blocks are used **then**
    break;
  end if
end for

Figure 2: Block/thread selection algorithm.

When choosing other thread and block dimensions (where memory coalescing is no longer a concern), we use the following heuristics instead:

- Choose the loop dimensions with the maximum trip-count. This ensures that we maximize the trip count
- For a thread, the loop dimensions j and j' of any two statements S and S' must be thread-compatible: i.e., either $j = j'$, or else j is not nested under j' or vice versa.
- For a block, the loop dimensions j and j' of any two statements S and S' must be block-compatible: i.e., $j = j'$.

This last restriction for block-compatibility can be clarified by the following example. Suppose we have the following inter-tile loop dimensions:

```
doall (i = ...) {
  doall (j = ...) {
    S1;
  }
  doall (k = ...) {
    S2;
  }
}
```

We can only choose loop-i as the block dimension for S1 and S2, i.e., it is illegal to choose j as the block dimension for S1 and k for S2. This is because the merging transformation cannot be applied to parallel block loops, only to parallel thread loops. In practice, we have a software implementation to synchronize blocks but its cost is usually deemed profitable only when no outermost parallel loops can be exhibited. For instance, this is the case for gauss-seidel stencil iterations.

4.7 An example

The result of block and thread placement on our running example is shown in the following loop nests. The first thread dimension has a trip count of 32. Since we are only allowed a maximum of 512 threads on the 9800GX2, our second thread dimension is limited to 16. The second selected thread dimension has a trip count of 32. To maintain the limit of 512 threads, we stripmine the loop nest by 16 and use the stripmined loop as the second thread dimension. After this, we have exhausted the number of threads. We then proceed to select the block dimensions, which are loops i and j.

In the following automatically generated code, both block dimensions have trip counts of 128:

```
__shared__ float C_l[32][32];
__shared__ float A_l[32][32];
__shared__ float B_l[32][32];
__device__ float A[4096][4096];
__device__ float B[4096][4096];
__device__ float C[4096][4096];
doall_block (i = 0; i <= 127; i++) { // bl.x
 doall_block (j = 0; j <= 127; j++) { // bl.y
  doall (k = 0; k <= 1; k++)
   doall_threads (l = 0; l <= 15; l++) // th.y
    doall_threads (m = 0; m <= 31; m++) // th.x
     C_l[16 * k + l][m] = 0;
  reduction_for (k = 0; k <= 127; k++) {
   doall (l = 0; l <= 1; l++)
    doall_threads (m = 0; m <= 15; m++) // th.y
     doall_threads (n = 0; n <= 31; n++) // th.x
      B_l[16 * l + m][n] =
       B[32 * k + 16 * l + m][32 * j + n];
    doall (l = 0; l <= 1; l++)
     doall_threads (m = 0; m <= 15; m++) // th.y
      doall_threads (n = 0; n <= 31; n++) // th.x
       A_l[16 * l + m,n] =
        A[32 * i + 16 * l + m][32 * k + n];
    doall (l = 0; l <= 1; l++)
     doall_threads (m = 0; m <= 15; m++) // th.y
      doall_threads (n = 0; n <= 31; n++) // th.x
       reduction_for (o = 0; o <= 31; o++)
        C_l[16 * l + m][n] +=
         A_l[16 * l + m][o] * B_l[o][n];
  }
```

```
  doall (l = 0; l <= 1; l++)
   doall_threads (m = 0; m <= 15; m++) // th.y
    doall_threads (n = 0; n <= 31; n++) // th.x
     C[32 * i + 16 * l + m][32 * j + n] =
      C_l[16 * l + m][n];
 }
}
```

4.8 CUDA placement and synchronization generation

The above set of loop nests is still not in the standard CUDA kernel form. The following merging heuristics can be used to transform arbitrary loop nests into the standard form:

1. Loop dimensions that are assigned to a block or thread should be made implicit.
2. Loop dimensions that are below block dimensions can be sunken into the CUDA kernel and executed sequentially. Note that doing so may require addition synchronizations to be inserted.

In our running example, the loop dimensions `i` and `j` are used as the block dimensions. Since there are no loop dimensions above `i` in this example, the entire loop nests may be executed in the CUDA kernel, and the host-side code contains only a kernel launch. The reduction loop dimension `k` can be sunken into the CUDA kernel; doing so requires the introduction of `__syncthreads()` calls to sequentializes the execution within this loop. The resulting transformed loop nests are as follows:[2]

```
__shared__ float C_l[32][32];
__shared__ float A_l[32][32];
__shared__ float B_l[32][32];
__device__ float A[4096][4096];
__device__ float B[4096][4096];
__device__ float C[4096][4096];
doall (i = 0; i <= 1; i++)
  C_l[16 * i + th.y][th.x] = 0;
__syncthreads();
reduction_for (i = 0; i <= 127; i++) {
  doall (j = 0; j <= 1; j++)
    B_l[16 * j + th.y][th.x] =
       B[32 * i + 16 * j + th.y,32 * bl.y + th.x];
  doall (j = 0; j <= 1; j++)
    A_l[16 * j + th.y][th.x] =
       A[16 * j + 32 * bl.x + th.y,32 * i + th.x];
  __syncthreads();
  doall (j = 0; j <= 1; j++)
    reduction_for (k = 0; k <= 31; k++)
      C_l[16 * j + th.y][th.x] +=
         A_l[16 * j + th.y][k] * B_l[k][th.x];
  __syncthreads();
}
doall (j = 0; j <= 1; j++)
  C[16 * j + 32 * bl.x + th.y][32 * bl.y + th.x]
   = C_l[16 * j + th.y][th.x];
```

4.9 Privatization

Memory utilitization of the above memory can be further improved by recognizing that each thread writes to its own disjoint set of locations in `C_l`. Thus the following transformation on references is possible:

```
C_l[16 * j + th.y][th.x] ⟶ C_l[i]
C_l[16 * j + th.y][th.x] ⟶ C_l[j]
```

The resulting loop nests after privatization are as follows. In this example, each thread keeps around two running sums for inside the local array `C_l`.

```
float C_l[2]; // local memory
__shared__ float A_l[32][32];
__shared__ float B_l[32][32];
__device__ float A[4096][4096];
__device__ float B[4096][4096];
__device__ float C[4096][4096];
doall (i = 0; i <= 1; i++)
  C_l[i] = 0;
__syncthreads();
reduction_for (i = 0; i <= 127; i++) {
  doall (j = 0; j <= 1; j++)
    B_l[16 * j + th.y][th.x] =
       B[32 * i + 16 * j + th.y,32 * bl.y + th.x];
  doall (j = 0; j <= 1; j++)
    A_l[16 * j + th.y][th.x] =
       A[16 * j + 32 * bl.x + th.y,32 * i + th.x];
  __syncthreads();
  doall (j = 0; j <= 1; j++)
    reduction_for (k = 0; k <= 31; k++)
      C_l[j] += A_l[16 * j + th.y][k] * B_l[k][th.x];
}
__syncthreads();
doall (j = 0; j <= 1; j++)
  C[16 * j + 32 * bl.x + th.y][32 * bl.y + th.x] =
   C_l[j];
```

4.10 Loop fusion

To reduce control overhead, adjacent loops with compatible trip counts may be fused. In our example, we may fuse the loop nests:

```
doall (j = 0; j <= 1; j++)
  B_l[16 * j + th.y][th.x] =
     B[32 * i + 16 * j + th.y,32 * bl.y + th.x];
doall (j = 0; j <= 1; j++)
  A_l[16 * j + th.y][th.x] =
     A[16 * j + 32 * bl.x + th.y,32 * i + th.x];
```

into:

```
doall (j = 0; j <= 1; j++)  {
  B_l[16 * j + th.y][th.x] =
     B[32 * i + 16 * j + th.y][32 * bl.y + th.x];
  A_l[16 * j + th.y][th.x] =
     A[16 * j + 32 * bl.x + th.y][32 * i + th.x];
}
```

Note that in general, this loop fusion requires adding a `__syncthreads` inside the parallel loop as has been described in section 3. In the particular case of communications between memories, the lack of dependence allows us to omit this spurious synchronization. This is equivalent to saying that memory transfers are fully parallel across imperfectly nested loops.

[2]We use `th` for `threadIdx` and `bl` for `blockIdx` to reduce clutter in the pseudo code.

4.11 Loop unrolling

A complementary transformation that we can do to further reduce control overhead is to unroll loops with small trip counts. Our current heuristic is to unroll all loops with trip count of 4 or less. With this in place, the resulting code is follows:

```
float C_l[2];
__shared__ float A_l[32][32];
__shared__ float B_l[32][32];
__device__ float A[4096][4096];
__device__ float B[4096][4096];
__device__ float C[4096][4096];

C_l[0] = 0;
C_l[1] = 0;
__syncthreads();
reduction_for (i = 0; i <= 127; i++) {
   B_l[th.y][th.x] =
      B[32 * i + th.y][32 * bl.y + th.x];
   B_l[16 + th.y][th.x] =
      B[16 + 32 * i + th.y][32 * bl.y + th.x];
   A_l[th.y][th.x] =
      A[16 + 32 * bl.x + th.y][32 * i + th.x];
   A_l[16 + th.y][th.x] =
      A[16 + 32 * bl.x + th.y][32 * i + th.x];
   __syncthreads();
   reduction_for (k = 0; k <= 31; k++) {
      C_l[0] += A_l[th.y][k] * B_l[k][th.x];
      C_l[1] += A_l[16 + th.y][k] * B_l[k][th.x];
   }
   __syncthreads();
}
C[32 * bl.x + th.y][32 * bl.y + th.x] = C_l[0];
C[16 + 32 * bl.x + th.y][32 * bl.y + th.x] = C_l[1];
```

4.12 Performance discussion

Our matrix multiply kernel runs at 116 GFlops/s on one GPU core of 9800GX2 chip,[3] versus 178 GFlops/s when using the CUBLAS v2.0 `sgemm` routine. It achieves 232 GFlops/s on GTX 285 chip compared to 395 GFlops/s from the CUBLAS v2.0 routine. This is using version 2.0 of the *nvcc* compiler.

Our mapped routine does not yet perform as well as the library. Still, our work shows the feasibility of automating the difficult transformations from C source to CUDA and achieving a reasonable percentage of peak performance, using the advanced mapping capabilities of R-Stream.

We take a step back to compare these results to Volkov's results [20]. In this work, a careful examination of the table on page 7 shows R-Stream automatically derives the same mapping features as the reported CUBLAS v1.1 implementation (32x32x32 tile sizes and array C privatized). The fraction of performance peak of 36-44% reported by Volkov for CUBLAS v1.1 is close to what R-Stream obtains automatically. We speculate the roughly 5% performance difference remaining between R-Stream and CUBLAS v1.1 might be linked to the performance of the low-level single thread code that we did not optimize. For the case of matrix multiply, Volkov shows a fraction of 58-60% of the hardware peak can be reached. This corresponds to our measurements with CUBLAS v2.0. The necessary steps needed to generate a version with performance comparable to CUBLAS v2.0 are well suited to optimization in the polyhedral model. Our prototype implementation shows promising preliminary results that should help close this gap. It will be the subject of a future publication.

[3]The 9800GX2 has two GPU cores and our current mapping only utilitizes one.

4.13 Differences with existing approaches

This short section aims at clearly pointing out the differences between the R-Stream implementation of a polyhedral model based restructuring compiler and existing approaches. To the best of our knowledge, there are currently 3 other major active projects using the polyhedral model: the GCC compiler, IBM's XLC compiler and the contributions from Ohio State University [6, 3, 4]. We are not aware of recent developments inside GCC and XLC to specifically target GPGPUs. Our comparison will therefore be limited to recent contributions from Ohio State University.

The first major difference resides in the the implementation of the various mapping phases we described. R-Stream implements all these phases in the polyhedral model. Information such as: which dimensions correspond to which threadId or blockId dimension are available to any mapping phase in the compiler. This is important for future optimizations such as removing redundant communications for instance. In contrast, recent contributions [3, 4] tend to defer the mapping to a later syntactic phase outside of the polyhedral model. Although contributions exist to restructure a program represented as a tree back into the polyhedral model [18], they do not discuss the case of explicit processor dimensions and they are very cumbersome.

A second difference comes from the way we perform affine scheduling. We optimize a weighted tradeoff between parallelism, locality and contiguity of memory references via *affine scheduling*. This step exposes as much parallelism as can be exploited in the target machine model. In particular it does not sacrifice parallelism for locality or contiguity. Other existing approaches tend to search for a schedule with maximal parallelism given stringent constraints on the fusion-distribution structure [6] or first select a fusion-distribution structure and only then optimize for maximal parallelism [17]. Our approach is able to optimize all these metrics in a single unified problem.

A third difference concerns the explicit generation of communications and the associated memory promotion phase. R-Stream performs data layout transformations and local storage size optimization by solving optimization problems not described in this paper. R-Stream also allows the automatic genneration of multi-buffered code, although we have not found this step to be beneficial in the context of automatic CUDA generation. Other approaches we are aware of do not perform such advanced optimizations.

Lastly, the final difference resides in the selection of tile sizes, grid sizes and thread block sizes. R-Stream computes these mapping parameters automatically by using advanced counting algorithms based on Ehrhart polynomials to optimize various metrics decribed in section 3.2. R-Stream also allows the user to specify the dimensioning of its mapping by command line flags to experiment with various sizes. In contrast, recent contributions [4] require the tile sizes, grid sizes and thread block sizes to be set explicitely or computed by other means.

4.14 Beyond embarrassing parallelism : Block-level synchronization

Performing synchronization at the host side is costly in terms of overhead due to kernel launch and possible data transfer between host and GPU. To avoid this, we implement a `__syncblocks` primitive using atomic intrinsics in the deveice memory space. To make sure that the program is free from race conditions and potential deadlock, we restrict the number of thread blocks used for executing a GPU kernel to be equal to the number of streaming multiprocessors in the GPU chip. This restriction ensures that there is no more than one thread block per streaming multiprocessor and all thread blocks are active at any point. We have seen in practice that performing synchronization across thread blocks on the GPU side rather than the host side results in a huge performance improvement.

With the `__syncblocks` primitive in place, we mapped an interesting and non-trivial kernel, namely Gauss Seidel 2D stencil, using R-Stream and the automatically generated CUDA code achieved around 16 GFlops/s and 21 GFlops/s for the 5 point stencil and 9 point stencil, respectively, on a GTX 285 GPU.

To give a feel of the order of magnitude of host-based synchronizations, 1024 iterations of the 5 point Gauss Seidel 2D stencil on a 4096x4096 grid take about 10.5 seconds on a GTX 285 GPU using `__syncblocks`. This is likely to be much improved with a dynamic distribution of the computations as has already been demonstreted [5]. The performance increase comes from reducing the very poor load balancing properties of pipelined parallel stencil programs. On the other hand the time required to perform host based synchronizations is around 8 seconds. This timing is without any computation and is already worse than what we expact to reach on a single GPU.

4.15 System level mapping

The R-Stream implementation has facilities for hiearchical and heterogeneous mapping. The machine model input file describes architectures in a recursive manner. Host-GPU partitioning and parallelization are a generalization of the host to processing element parallelization and partitioning used for targeting the Cell architecture. After the first level of partitioning is performed, a software pipeline for operand and result communication is created (using the techniques from [14]; and then the compiler recursively "pushes in" to parallelize for the individual accelerator. Implementation of this feature is in-progress for several architecture targets. In the particular case of multi-GPUs, we are developing additional models to account for the costly host-based synchronization. From the point of view of the R-Stream mapper, this is just another level of memory and synchronization hierarchy. While it is too early to present results, the

5. CONCLUSION

We have presented our port of R-Stream to the CUDA target. While we have not nearly tuned (or auto-tuned) it, performance results are already encouraging: (1) automatically reaching performance close to CUBLAS1.1, and (2) achieving a significant fraction - automatically - of the performance provided by current libraries, from ANSI C. That the input C is not geared toward a particular architecture or execution model is a factor in increasing portability; the input has less binding and more options for mapping. The pathway to portability and productivity in code is through encouraging expression of algorithms at the highest semantic level and using compiler building blocks to automatically explore (implicitly or explicitly) different code configurations to find one with good performance.

We coined the "textbook C" to emphasize that to use R-Stream, the programmer must currently write in a simple manner, fewer lines and with fewer hand optimizations. This makes the semantics of the program clear and means that fewer things (programmer memory management, etc.) have to be undone by the compiler in raising stages to get at the semantics to be optimized by the mapper. Typically, though, programmers have not been writing in this style, perhaps not because they wanted to but because previous compilers were weak. But also, they have not been writing in CUDA or OpenCL, either (until recently). What we are showing is a way of getting the performance potential of GPGPU with a much, much, simpler programming abstraction, and one that is portable to other advanced parallel and accelerator architectures. The project of extending the definition of "textbook C" to a broader set of C idioms is to a large extent orthogonal to the mapping considerations.

We will present more results at the workshop. In particular, we will discuss in more detail the mapping of the complex Gauss-Siedel kernel, showing the complex code forms that the compiler can produce, and the relationship between the performance achieved by GPGPU and for General Purpose processors.

An aspect of the Gauss Siedel kernel that is interesting is that it is certainly not trivially vectorizable. In many instances, the most impressive results of hand-coding for GPGPU come from building on the vector primitive [21]. While the polyhedral framework on which we are building can certainly be used to explore different vectorization decompositions, it has a broader repertoire of transformations and in particular locality enhancing fusions that can potentially significantly improve the performance of non-trivially vectorizable codes, including in particular dense matrix iterative algorithms.

6. REFERENCES

[1] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, Williamsburg, VA, April 1991.

[2] A. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Mathematics of Operations Research*, 19:769–779, 1994.

[3] M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for gpgpus. In *ACM International Conference on Supercomputing (ICS)*, Jun 2008.

[4] M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'10)*, lncs, Cyprus, March 2010. Springer-Verlag.

[5] M. Manikandan Baskaran, N. Vydyanathan, U. Bondhugula, J. Ramanujam, A. Rountev, and

P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In *PPOPP*, pages 219–228, 2009.

[6] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Affine transformation for communication minimal parallelization and locality optimization of arbitrarily nested loop sequences. Technical Report OSU-CISRC-5/07-TR43, The Ohio State University, May 2007.

[7] U. Bondhugula, A. Hartono, J. Ramanujan, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Programming Languages Design and Implementation (PLDI '08)*, Tucson, Arizona, June 2008.

[8] E. Ehrhart. Polynomes arithmetiques et methode des polyedres en combinatoire. *International Series of Numerical Mathematics*, 35, 1977.

[9] P. Feautrier. Some efficient solutions to the affine scheduling problem. part I. One-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, October 1992.

[10] Khronos OpenCL Working Group. The openCL specification (version 1.0). Technical report, 2009.

[11] D. J. Kuck. *High Performance Computing*. Oxford University Press, 1996.

[12] R. Lethin, A. Leung, B. Meister, P. Szilagyi, N. Vasilache, and D. Wohlford. Final report on the R-Stream 3.0 compiler DARPA/AFRL Contract # F03602-03-C-0033, DTIC AFRL-RI-RS-TR-2008-160. Technical report, Reservoir Labs, Inc., May 2008.

[13] R. Lethin, A. Leung, B. Meister, P. Szilagyi, N. Vasilache, and D. Wohlford. Mapper machine model for the R-Stream compiler. Technical report, Reservoir Labs, Inc., Nov 2008.

[14] B. Meister, A. Leung, N. Vasilache, D. Wohlford, C. Bastoul, and R. Lethin. Productivity via automatic code generation for pgas platforms with the r-stream compiler. In *Workshop on Asynchrony in the PGAS Programming Model*, Jun 2009.

[15] B. Meister and S. Verdoolaege. Polynomial approximations in the polytope model: Bringing the power of quasi-polynomials to the masses. In *ODES-6: 6th Workshop on Optimizations for DSP and Embedded Systems*, Apr 2008.

[16] NVIDIA. Cuda zone `http://www.nvidia.com/cuda`, 2008.

[17] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Hybrid iterative and model-driven optimization in the polyhedral model. Technical Report 6962, INRIA Research Report, June 2009.

[18] N. Vasilache. *Scalable Program Optimization Techniques In the Polyhedral Model*. PhD thesis, University of Paris-Sud, September 2007.

[19] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and Maurice Bruynooghe. Analytical computation of Ehrhart polynomials: enabling more compiler analyses and optimizations. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 248–258. ACM Press, 2004.

[20] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[21] V. Volkov and J. W. Demmel. LU, QR and Cholesky factorizations using vector capabilities of GPUs. Technical Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley, May 2008.