HiDP: A Hierarchical Data Parallel Language *

Yongpeng Zhang, Frank Mueller

North Carolina State University, Raleigh, NC 27695-7534, mueller@cs.ncsu.edu

Abstract

Problem domains are commonly decomposed hierarchically to fully utilize parallel resources in modern microprocessors. Such decompositions can be provided as library routines, written by experienced experts, for general algorithmic patterns. But such APIs tend to be constrained to certain architectures or data sizes. Integrating them with application code is often an unnecessarily daunting task, especially when these routines need to be closely coupled with user code to achieve better performance.

This paper contributes HiDP, a hierarchical data parallel language. The purpose of HiDP is to improve the coding productivity of integrating hierarchical data parallelism without significant loss of performance. HiDP is a source-to-source compiler that converts a very concise data parallel language into CUDA C++ source code. Internally, it performs necessary analysis to compose user code with efficient and architecture-aware code snippets.

This paper discusses various aspects of HiDP systematically: the language, the compiler and the run-time system with built-in tuning capabilities. They enable HiDP users to express algorithms in less code than low-level SDKs require for native platforms. HiDP also exposes abundant computing resources of modern parallel architectures. Improved coding productivity tends to come with a sacrifice in performance. Yet, experimental results show that the generated code delivers performance very close to handcrafted native GPU code.

1. Introduction

Contemporary research seems to indicate that no panacea can seamlessly adapt sequential legacy programs to modern parallel architectures. Simply converting programs into many concurrently executing threads may not necessarily deliver expected levels of performance improvements. This is partly due to today's parallel machines consisting of far more complicated execution and memory hierarchies than the simplistic Von Neumann model, which may be a suitable abstraction for sequential programs — but not so much for today's hierarchical parallelism. Any approaches that ignore such hierarchies are likely to yield performance inferior to they capabilities.

Execution Level	Suitable Parallelism	Synchronization
Kernel	less than a few dozens tasks	kernel boundaries
Block	a few dozens to a few hundreds	$\_syncthreads()$
Warp	more than a few hundreds	Not Necessary
Thread	more than tens of thousands	Not Necessary

Table 1. Execution Hierarchies in Modern GPUs

Consider modern GPU architectures as an example. Table 1 lists the execution hierarchies in Nvidia GPUs. For a single GPU unit, there exist at least four execution levels, each of which features different favorable degrees of parallelism and synchronization methods. *Suppose a problem can be divided by a number of concurrent tasks, which can be realized by fine-grained data-parallel threads cooperatively.* When the number of tasks is small (a few to a few dozens), the top *kernel* level suffices to utilize all GPU computing resources. This is done by assigning multiple blocks to one logic task. But this requires the local barrier synchronization in one task to be replaced by a more expensive global barrier because of a lack of hardware synchronization across multiple blocks. We can also assign just one block to process one task. The benefit of doing this is that task barriers can be implemented locally inside each block. But this is only beneficial when the number of tasks is large enough to exercise all GPU resources. As we delve down to the warp or thread level, synchronization comes at no cost because it is ensured by SIMD and instruction ordering. However, GPUs need substantially more parallelism to reach their peak instruction throughput. There is no clear delimiter to the range of suitable parallelism for each execution level. Effective hierarchical parallelization hence often become dependent on both the hardware and the application. This ambiguous boundary exposes more challenges to languages, as it becomes the task of the compiler and runtime to decouple hierarchical parallelism from the algorithmic expressions.

Hierarchical architectures like GPUs have a substantial influence on the way to solve problems. A common approach to match the hardware structure involves a two-step decomposition. Firstly, a task-level divide-and-conquer approach partitions a large-scale problem into a number of smaller tasks. This step provides opportunities for reducing synchronization overhead and utilizing faster but limited on-chip caches. Secondly, these tasks are executed by a massive

* This work was supported in part by NSF grants 1217748, 1058779, and 0958311.

CGO'13 23-27 February 2013, Shenzhen China.
978-1-4673-5525-4/13/$31.00 ©2013 IEEE. . . $15.00

number of threads cooperatively, demanding exposition of fine-grained data parallelism. As a result, data-parallel primitives, such as segmented parallel scan and reduction (detailed later), play an important role in coding productivity and performance.

Fortunately, many seemingly inherently sequential operations (reduce, scan, partition, etc.) have efficient parallel solutions. A significant effort has been invested by experienced programmers in providing such solutions, often as libraries. In the CUDA ecosystem, libraries like Thrust [14] and CUDPP [15] are widely used by developers to avoid implementing such operations from scratch. However, users often find it difficult to integrate these libraries with their own code for mainly two reasons: (1) User code and libraries are often closely coupled. Users often have to investigate the fine details of the library. (2) Libraries often provide alternate implementation choices of the same functionality due to the hierarchical execution model in today's microprocessors. The best choice is often data dependent, yet not necessarily obvious to programmers. Therefore, it is necessary to try each of them, which can be a daunting task for end users.

Contributions: We propose HiDP, a data-parallel language with hierarchical parallel-for clauses and built-in data-parallel primitives. These language features are equally suitable for describing parallel algorithms and for obtaining high performance in contemporary GPUs. The HiDP compiler judiciously maps parallel for constructs onto the hierarchical execution model of modern GPUs. When multiple mapping choices exist, the compiler generates different versions of code, one for each mapping. It also emits tuning code that aids users in selecting the appropriate version, or the user can manually prune alternatives based on domain knowledge.

HiDP is a machine-independent language. In fact, HiDP encourages users to express algorithms in a general, architecture-neutral fashion. This makes HiDP robust to future architectural advances and extensible for code generation in other formats, such as OpenCL [16]. Like many other high-level languages, HiDP is very concise and easy to learn. More than an order of magnitude of lines of code can be saved compared to native CUDA code. HiDP could also serve as an intermediate language for other high-level languages since its code transformations result in performance that only marginally falls short of hand-written CUDA code.

2. The HiDP Language

A HiDP program is built around a top-level structure specified through the keyword *function* followed by the function name. Other functions can be defined and can be called by the top-level function or by each other. Yet, there is only a single entry to a HiDP program. The compiler will generate a legal C++ function signature and body based on the top-level function.

The header of the function body declares the arguments of this function. The data flow and read/write access properties inside the function are indicated by keywords *input*, *output* or *inout*. All arguments of a function are passed by reference, *i.e.*, a change of an argument in the function will be seen by the function's callee.

In designing the HiDP language, we pursue the following major goals: We intend to: (1) expose low-level data structures for full control over the data layout design; (2) preserve the conciseness of data parallel script languages; (3) provide the ability to customize data-parallel operations, *e.g.*, hierarchical or partial mappings; (4) embed basic data-parallel primitives in the language to improve coding productivity; and (5) keep the language platform independent and only add machine-dependent information at the directive level. In the following subsections, we present key aspects of HiDP and explain how our goals are met by them.

2.1 Data Types

HiDP's basic structure is an array of any dimension (Scalars have a dimension of 0). There are two ways to declare a variable. One is at the function header (argument), the other is inside the function body (local variable):

```
data_type var([dim0]...[dimn]);
```

A declaration starts with a data type identifier, which can be either a fundamental C/C++ data type (char, float int ...) or a derived (user-defined) one. The number of bracket pairs after the variable name implies the dimension of the variable. The size in each dimension of these arguments is expressed symbolically in terms of either constant or free variables, the values of which must be determined by the HiDP runtime system. An example of declaring a 2D dimensional float array and a 1D dimensional float array is as follows:

```
float my_2d_array[I][J], my_1d_array[K];
```

The other option for declaration is to specify a data type for a scalar integer variable at the beginning of a *map block* (see Section 2.3). Such a scalar integer has to be within a certain range, which is expressed as two tuples enclosed by brackets (inclusive) or parentheses (exclusive). We call this kind of variable a *map iterator*:

```
map_iter := ranges;
```

The range is relaxed to be any arithmetic expression of variables and constants. An example of declaring a map iterator i from 1 to $J-1$ is:

```
i := [1 : J−1];
```

We will discuss the *map iterator* in more detail later.

2.2 Data Parallel Expressions

Like many other data-parallel languages, HiDP allows concise array operations on each element of a structure. This corresponds to the concept of an *apply-to-each* or *map* construct in other languages. Such expressions, together with the *map block*, form the fundamental statements of the HiDP language. Consider the statement

```
A = B ∗ C;
```

All elements of A are updated by the multiplication of elements at the same relative position in B and C. HiDP requires that all variables in a data parallel expression maintain the same shape (same number of dimensions and same size on each dimension) but allows scalar variables to "expand" to the same shape as other multi-dimensional variables in the same expression.

2.3 Hierarchical Map Blocks

The support for data parallel expressions above improves coding productivity by eliminating some *for* loops of languages like C and C++. But the default *apply-to-each* behavior may be too strict to express certain algorithms.

HiDP relaxes its stringent behavior by defining a *map block* following the principle idea of a *parallel for* construct. The number of iterations inside a *map block* is determined by *map iterators*, which must be defined at the beginning of the *map block*, but there may be multiple ones of them defined for each *map block*. In addition, HiDP allows an optional suffix function call to be made at the end of the *map block*. Therefore, a *map block* can be of the following formats (following EBNF notation):

```
map_block: map block
         | map block suffix
block: { statements }
suffix: function_call
```

Map blocks can be hierarchical. A *map block* is called another *map block*'s parent if the former fully encloses the later. Two types of statements can reside in a *map block*: scalar expressions and pre-defined data-parallel primitives (see Section 2.4). The *parallelism* of a *map block* is determined by the product of all *map iterators* of itself and all its parent *map blocks*. The level of *parallelism* is expressed by the number of concurrent scalar expressions executed in this *map block*. HiDP assumes sequential execution of instructions in the *map block* but does not assume any synchronization between different iterators (except for entering and exiting data parallel primitives, see Section 2.4). Therefore, the behavior of any writing to the same memory position from different iterators is undefined.

Many applications have inherent nested parallelism. This fits naturally with HiDP's hierarchical map blocks. Starting from the outermost map block, the compiler's major role is to determine which execution model is best suited for this level, optionally enhanced by programmer hints.

2.4 Data Parallel Primitives

HiDP supports many data-parallel primitives that improve coding productivity. Those primitives can be written either outside any *map block* or inside/appended to a *map block*. If associated with a map block, primitives implicitly call local barriers before entering and after exiting the block. Primitives inside the map block can be regarded as segmented primitives. All operations are performed independently within each segment. Each segment may execute within several blocks cooperatively, within a single block, within a warp or even within a thread. This depends on the number of segments and availability of the primitive's implementation at the execution level. Such choices are ultimately made by the HiDP compiler and runtime. HiDP requires each irregular segmented array to be associated with two index vectors, termed low_range and $high\_range$. These vectors indicate the low and high indexes of each segment, respectively. This representation of a segmented array differs from NESL [4], where an associated boolean array of the same size as the original array is used to infer segment boundaries. For regular segmented arrays (segment sizes are the same), HiDP supports a different interface where only two scalar inputs are used to replace the two low and high index vectors: seg_size and num_seg. This design choice was driven by practical considerations. We find that significant performance benefits can sometimes be achieved if prior knowledge about regularity is available. Table 2 shows the syntax of selected data-parallel primitives in different scenarios.

Irregular Parallel Primitives
$\_min/\_max = min/max(input, low\_range, high\_range)$
$sort(in\_key, out\_key, [in\_value, out\_value], low\_range, high\_range, dir)$
$partition(in, out, [in\_value, out\_value], in\_low\_range, pivots,$ $in\_high\_range, out\_low\_range, out\_high\_range, function)$
$reduce(" + /*", input, output, low\_range, high\_range)$
$scan(in, out, low\_range, high\_range)$
$reverse\_inplace(inout, low\_range, high\_range)$
$reverse(in, out, low\_range, high\_range)$
Regular Parallel Primitives
$\_min/\_max = reg\_min/\_max(in, out, seg\_size, num\_seg)$
$reg\_sort(in\_key, out\_key, [in\_value, out\_value], seg\_size, num\_seg, dir)$
$reg\_reduce(" + /*", in, out, seg\_size, num\_seg)$
$reg\_scan(input, output, seg\_size, num\_seg)$
$reg\_reverse(inout, seg\_size, num\_seg)$
$reg\_reverse(in, out, seg\_size, num\_seg)$
Outside Map Block
$\_min/\_max = min/max(in, size)$
$sort(in, out, size, dir)$
$partition(in\_key, out\_key, [in\_value, out\_value], pivot, size, function)$
$reduce(" + /*", in, out, size)$
$scan(in, out, size)$
$reverse\_inplace(inout, size)$
$reverse(in, out, size)$
Map Suffix Functions
$reduce(" + / * /min/max", output, input, ranges)$

Table 2. Selected Data Parallel Primitives in HiDP

A typical usage of a partition in a map clause is, e.g.:

```
float in[size], out[size], pivots[num_segs];
int low[num_segs], high[num_segs], out_low[num_segs*2],
new_high[num_segs*2];
...
map {
  seg := [0: num_segs);
  partition(in, out, low, pivots, low, high, new_low,
            new_high, MyCompare); }
```

There will be num_segs instances of partition operations on the input array in. The ith instance ($0 <= i < num\_segs$) works on elements between $low[i]$ and $high[i]$. The new index ranges for the two new smaller partitions are created in $new\_low[2 * i], new\_high[2 * i], new\_low[2 * i + 1]$ and $new\_high[2 * i + 1]$. $MyCompare$ can either be a native CUDA device function or an internal HiDP function.

Depending on the position of the data parallel primitive, there may be multiple instances of primitive calls. For example, if a data primitive is called inside a *map block*, the number of instances is the *parallelism degree* of the current *map block*. These instances can be executed in parallel without any synchronization. But HiDP assumes local barriers before and after each of them. In other words, the range of the synchronization is constrained to the necessary range to guarantee correctness of each instance. If a primitive is a suffix function call for a *map block*, only a single local barrier constrained by the *ranges* is needed.

2.5 User-Assisted Directives

HiDP supports directives as annotations for map clauses. They are required to assist the compiler in performing the mapping from a hierarchical structure to an execution model. They often require prior knowledge that cannot be deduced by the compiler, and they help reduce the exploration space. Such directives must immediately precede the *map clause* in the program. Their syntax is:

```
#pragma hidp [kernel|block|warp|subwarp|thread]
```

2.6 GEMM in HiDP

As a concrete example, consider the HiDP source code for the level-3 BLAS GEMM routine in Figure 1. Lines 2 to 4 define the function header. The body of the function consists of a single-level map structure with a reduce suffix on temporary variable $\_c0$. Line 8 defines three *map iterators* for the *map block*. The reduction is applied to all k iterators (line 10) for different i and j iterators and is assigned to the 2-D array $C1$. As mentioned above, synchronization is implicitly reinforced before and after the suffix reduction call, but only at a local range (for every k iterators). After $C1$ is updated, C is finally calculated by the GEMM rule (line 11).

```
1  # implementing C = alpha * A * B + beta * C
2  function GEMM
3  input float _alpha, _beta, A[M][K], B[K][N];
4  inout float C[M][N];
5  {
6    float C1[M][N];
7    map{
8      m:=[0:M); n:=[0:N); k:= [0:K);
9      _c0 = a[m][k] * b[k][n];
10   } reduce("+", C1[m][n], _c0, k:=[*]);
11   C = _alpha * C1 + _beta * C; }
```

Figure 1. GEMM in HiDP

HiDP encourages users to express algorithms at the finest data granularity. For the GEMM example, this occurs at line 9 in Figure 1, where the element-wise multiplication over all three dimensions is expressed. This makes HiDP independent of the underling hardware architecture. The decisions on whether or not to fuse them and at which level are left to the compiler backend as it depends on the properties of the targeted hardware.

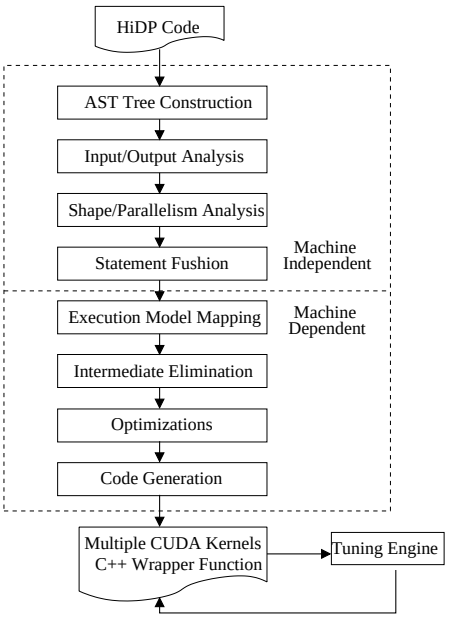


Figure 2. Overview of HiDP Compiler

3. The HiDP Compiler

In this section, we provide an overview of the compilation steps to transform HiDP into a set of CUDA/C++ functions containing both the host and device code. We use the GEMM example in the previous section as a running example.

3.1 Overview

The HiDP compiler consists of a number of phases shown in Figure 2. The input of the framework is a HiDP program with a single function entry. It emits one or even multiple versions of the CUDA kernel code and C++ host code for the same HiDP program. If the output contains multiple versions, a wrapping C++ function is also generated to aid during the tuning process. Users can intercept the tuning process and directly pick the most appropriate version.

3.2 Front End

The HiDP compiler parses each routine to transform a HiDP program into an abstract syntax tree (AST). It detects the top entry-level function and instantiates other internal functions at the top level. There are four types of statements in HiDP: assignment expression, map block, branch expression and function call. They are hierarchical in the sense that a map block can contain multiple assignment expressions inside. All further analysis is performed hierarchically at each statement level. The HiDP front end does not expand data-parallel expressions into for loops throughout the code transformations.

After parsing, the compiler recursively analyzes the input and output set of each statement from the bottom up. Having a complete understanding of this step is important for generating function arguments and recovering inout data in the tuning wrapper.

Execution Level	s_kernel	s_block	s_warp	s_sub-warp	s_sub-warp2	s_thread
1-D Shape	gridDim.x/BLOCK_PER_TASK	gridDim.x	s_block * WARP_PER_BLOCK	s_warp * 4	s_warp * 8	s_block * blockDim.x

Table 3. 1-D Shapes of Execution Model

Level	kernel	block	warp	sub-warp	sub-warp2	thread
s_kernel	1	-	-	-	-	-
s_block	BLOCK_PER_TASK	1	-	-	-	-
s_warp	s_block $*$ WARP_PER_BLOCK	WARP_PER_BLOCK	1	-	-	-
s_sub-warp	s_warp $*$ 4	s_warp $*$ 4	4	1	-	-
s_sub-warp2	s_warp $*$ 8	s_warp $*$ 8	8	4	1	-
s_thread	s_block $*$ blockDim.x	blockDim.x	32	8	4	1

Table 4. 1-D Shapes of Execution Model Given its Immediate Upper Layer

3.3 Nested Shape Representation and Analysis

HiDP relies heavily on shape analysis to perform statement fusion and execution model mapping in a safe manner. To achieve that, each statement is analyzed to obtain its shape, which indicates the maximal possible number of data-parallel threads for this statement. We call it the *parallelism degree* of a statement. Parallelism degrees can be multi-level depending on the position of a statement. We use the notation of {[level 0], ..., [level n-1] } to represent an n-level shape. For the GEMM example below, there are four statements in the HiDP source code: a map block (s1), an assignment inside the map (s2), a suffix function call (s3) and another assignment outside the map block (s4). The shape analysis starts from the innermost assignment (s2). Its shape is determined by the range of all map iterators at the same or higher levels. In this case, there is only one map block. Therefore, its shape is a single-level 3D shape of $\{[0:M, 0:N, 0:K]\}$. Next, the reduce call is analyzed. The reduction range is for all ks. Therefore, $[0:K]$ is promoted to the next level making the reduce function's shape a two-level shape of $\{[0:M, 0:N], [0:K]\}$. This is a two-level shape where each instance in the first level shape space ($M \times N$) contains up to K degrees of data parallelism. Its instances need local barrier support (reduction in this case). The shape of the map block is kept consistent with its suffix function call. The shape of the last statement is deduced from the dimension of its operands ($C1$ or C). Hence, it is a single level shape ($\{[0:M, 0:N]\}$). The shape of each statement after shape analysis is shown below:

```
float C1[M][N];
map{ # { [0:M, 0:N], [0:K] } (s1)
    m:=[0:M); n:=[0:N); k:= [0:K);
    _c0 = a[m][k] * b[k][n]; # { [0:M, 0:N, 0:K] } (s2)
}reduce(...); # { [0:M, 0:N], [0:K] } (s3)
C = _alpha * C1 + _beta * C; # { [0:M, 0:N] } (s4)
```

3.4 Statement Fusion

The main motivation behind combining as many operations as possible into a kernel is to save off-chip memory transactions because intermediate variables can be kept in registers, which avoids accesses to global memory. The HiDP compiler tries to merge statements at the top level ((s1) and (s4) in the GEMM example) building on shape analysis. Two statements can be fused if and only if their shapes are compatible with each other. Two shapes are compatible when one is a prefix of the other in a flattened format. In the GEMM example, (s4)'s shape is a prefix of (s1). Therefore, they can be fused into a larger unit. (s4)'s shape extends to the same number of level as (s2), while the parallelism degree in the second level is just 1. After fusion, the GEMM function becomes a single statement, as shown in the following:

```
{ map{ # { [0:M, 0:N], [0:K] }
      ... }
   C = _alpha * C1 + _beta * C; # { [0:M, 0:N], 1 }
} # { [0:M, 0:N], [0:K] }
```

The fused statement does not necessarily correspond to a single kernel at this point. Its transformation also depends on which execution model it is mapped to. For example, if the compiler later decides to assign multiple blocks to execute one instance in the $M \times N$ space at the first level shape, a barrier is needed for the reduction. This results in multiple kernels due to the lack of a global barrier across multiple blocks in CUDA.

3.5 Execution Model Abstraction and Mapping

Starting with this phase, the transformations are machine dependent. First of all, we depict the target machine as a set of hierarchical execution models. HiDP currently only supports CUDA in the back-end. We will thus use the CUDA terminology throughout the rest of the section (even though OpenCL or OpenMP mappings are feasible as well). We add two more execution models to the one mentioned in Section 1. We call them sub-warp (8 thread lanes) and sub-warp2 (4 thread lanes). Similar to statements in HiDP, each level has a physical shape, which corresponds to the number of parallel instances at this level in GPUs. Table 3 lists the one dimensional shapes for all supported execution models. The job of execution model mapping is to associate the hierarchical statement shape into appropriate physical shapes according to their *parallelism degrees*. The physical shape of an execution model also depends on its immediate upper layer during the mapping. The relative shape for each case is shown in Table 4. The lower level shapes always have equal or more parallelism than the upper level shapes.

Take GEMM as the example: The shape of the fused statement is $\{[0:M, 0:N], [0:K]\}$. The first level has $M \times N$ parallelism degrees. Since these are inputs to the function and are not known at compile time, HiDP may select any of the execution models, assuming $M \times N$ ranges from one to arbitrarily large number. The switching point is

marked as a tuning parameter. The second level K is always mapped to the thread level because it is the last level. To conserve space for depicting the code, we prune the tuning space from 6 possibilities to 3 by choosing only block, warp and thread for the first level shape mapping. In fact, this can also be done by inserting a pragma before the map block:

```
#pragma hidp block warp thread
```

The set of valid mappings are shown in the table below. The expressions in parentheses represent the physical parallelism degree at this level.

[0:M, 0:N]	[0:K]
block(gridDim)	thread(blockDim)
warp(gridDim * WARP_PER_BLOCK)	thread(32)
thread(gridDim * blockDim)	thread(1)

After this step, we are able to determine CUDA kernel delimiters for each mapping. Because our run-time supports in-kernel local barriers for the three mappings we choose, a single kernel can implement the fused statement. The scope of the local variable $C1$ is within the kernel and its access pattern is strictly sequential, meaning that each scalar in the array is accessed by the same iterator. Therefore, it can be kept in the register file without any writes to the global memory, obviating its storage allocation.

3.6 Machine Dependent Optimizations

An important optimization strategy for CUDA code generation is to take advantage of the fast on-chip Shared Memory. The HiDP compiler tries to detect shared access patterns between neighboring threads. Again, this depends on the final execution mapping. HiDP searches for arrays whose indices contain only constant or map iterators that are mapped to the thread execution model. HiDP reasons about the shape of thread layout to facilitate the loading of shared data.

3.7 Loop Unrolling and Code Generation

The final code generation step needs to consider the mismatch between the *parallelism degree* of the nested shape (usually data dependent) and the physical parallelism of the corresponding execution model. The former is often greater than the latter. Because the execution order of iterators in the same map block is irrelevant, we generate a for loop with the following template:

```
for (id = iter_start + level_id; id < iter_end; id+=level_stepsize) {
    iterator = id;
    ... (loop body); }
```

where $iter\_start$ and $iter\_end$ are the left and right boundaries of the map iterator. Furthermore, $level_stepsize$s are the same as the values in Table 4 for the case of a one dimensional shape.

Each supported data-parallel primitive has properties like shared memory usage and auxiliary variables. The properties are carried through the compiler framework and are interlaced with other HiDP code. On the host side, all arrays are encapsulated by the HiArray class, which supports an arbitrary number of dimensions and maintains data integrity according to the read/write properties deduced by the compiler. *It is not mandatory to support data-parallel primitives in every level of the execution model.* Restrictions are considered by the compiler to prune the number of possible execution model mappings.

3.8 GEMM CUDA/C++ Output

```
1  // gemm block version
2  __global__ void gemm_block(...) {
3    __shared__ float sum[BLOCK_SIZE]; // used for block reduction
4    int m = blockIdx.x;
5    int n = blockIdx.y;
6    float s = 0.0f;
7    for (k = 0 + threadIdx.x; k < K; k += blockDim.x)
8      s += fetchA(m,k) * fetchB(k,n);
9    reduce_block<ADD, float>(sum, s);
10   if (threadIdx.x == 0)
11     fetchC(m,n) = alpha * sum[0] + beta * fetchC(m,n);
12 }
13 // gemm warp version
14 __global__ void gemm_warp(...){
15   __shared__ float sum[WARPS_PER_BLOCK][WARP_SIZE];
16   int warpId = threadIdx.x/WARP_SIZE;
17   int warpIndex = threadIdx.x & 0x1F;
18   int m = blockIdx.x;
19   int n = blockIdx.y * WARPS_PER_BLOCK + warpId;
20   float s = 0.0f;
21   for (k = 0 + warpIndex.x; k < K; k += WARP_SIZE)
22     s += fetchA(m,k) * fetchB(k,n);
23   reduce_warp<ADD, float>(&sum[warpId][0], s);
24   if (warpIndex == 0)
25     fetchC(m,n) = alpha * sum[warpId][0] + beta * fetchC(m,n);
26 }
27 // gemm thread version, before shared memory optimization
28 __global__ void gemm_thread(...) {
29   int m = blockIdx.x * blockDim.x + threadIdx.x;
30   int n = blockIdx.y * blockDim.y + threadIdx.y;
31   float s = 0.0f;
32   for (k = 0; k < K; k += 1)
33     s += fetchA(m,k) * fetchB(k,n);
34   fetchC(m,n) = alpha * s + beta * fetchC(m,n);
35 }
```

Figure 3. HiDP Emits Different Kernels

Figure 3 lists the emitted GEMM kernel code for the aforementioned three mappings. Depending on the actual execution model mapping, HiDP emits different reduce functions (lines 9 and 23 for block and warp versions, but none for the thread version). The assignment expression inside the map block in HiDP code is converted into for loops (lines 7 to 8, 21 to 22, 32 to 33) according to the template mentioned in Section 3.7. Finally, special care needs to be taken when there is more physical parallelism than the shape parallelism at a certain level. The expression needs to be ensured to only enable the first few threads. This is the case for the (s4) statement because its shape is 1 on the second level, but there are multiple valid degrees of physical parallelism for the block and warp versions. Consequently, lines 10 and 24 are inserted by the compiler to adjust for the parallelism difference.

The host C++ code is generated with necessary branches to choose which version of the kernel to run (lines 11, 14 and

```
1  // generated wrapper code with tuning branches
2  void gemm_wrapper(HiArray<float, 1> &C, ...){
3    int C_dim0 = C.getDim(0); int C_dim1 = C.getDim(1);
4    ...;
5    int M = C_dim0; int N = C_dim1;
6    ...
7    vector<int> degree_0;
8    degree_0.push_back(M);
9    degree_0.push_back(N);
10   dim3 block, grid;
11   if (degree_0 < TUNING_0) {
12       config_block(degree_0, block, grid);
13       gemm_block<<<...>>>(...);
14     } else if (degree_0 < TUNING_1) {
15       config_warp(degree_0, block, grid);
16       gemm_warp<<<...>>>(...);
17     } else {
18       config_thread(degree_0, block, grid);
19       gemm_thread<<<...>>>(...);
20     }
21 }
22 // tuning function
23 void gemm_tuning(HiArray<float, 1> &C, ...){
24   int C_dim0 = C.getDim(0); int C_dim1 = C.getDim(1);
25   ...;
26   int M = C_dim0; int N = C_dim1;
27   ...
28   vector<int> degree_0;
29   degree_0.push_back(M);
30   degree_0.push_back(N);
31   dim3 block, grid;
32   for (int i = 0; i < 3; i++) { // three paths
33         save_inout_arrays();
34         start_timing();
35         gemm_block<<<...>>>(...); // for i == 0
36         gemm_warp<<<....>>>(...); // for i == 1
37         gemm_thread<<<...>>>(...); // for i == 2
38         end_timing();
39         report_timing(degree_0); }
40 }
```

Figure 4. Generated C++ Code by HiDP Compiler (Code is expanded for clarification purpose. Actual code may differ)

17 in Figure 4). The parameters TUNING_0 and TUNING_1 are tuning parameters that need to be determined later.

3.9 Auto-Tuning

If the compiler detects any tuning possibilities, it will also emit tunable code wrapped by profiling code to measure the execution time for each code path. The user can run the executable in the *tuning mode*, where the measured time for each training test case is reported. The second part of Figure 4 illustrates this concept. *After the training phase, the user can then launch an analysis tool* operating on the profiling results to determine appropriate switching conditions for different versions of generated code.

Our analysis tool performs a linear regression match to determine the best time to switch kernels. Occasionally, a switch may not be result in performance benefits because other factors besides the detected *parallelism degree* affect performance but are not factored into decisions. If that was the case, users could always overwrite HiDP's decision by supplying customized code around various kernels to select the best one based on their prior domain knowledge.

4. Experimental Results

In this section, we investigate the performance of HiDP's generated code in several examples. Not only do we compare with parallel implementations on CPUs in some cases, but also with published, hand-optimized CUDA implementations of the same workload. As we will see, even the compiler cannot apply some of the optimizing techniques that an experienced programmer can, while our auto-tuning scheme, an optimization phase that is usually ad-hoc or absent in hand-written code, closes this performance gap.

The experimental platform is a two-socket machine with two AMD Opteron 6128 processors (8 cores each), one Nvidia GTX 480 and 32 GB memory. All experiments are performed using single-precision floating point, unless stated otherwise.

4.1 GEMM

Following the GEMM example in previous sections, we compare with the GEMM of the CUBLAS 4.2 library, a hand-crafted BLAS implementation released by Nvidia.

Let the sizes of the three matrices in $C = alpha \times C + beta \times A \times B$ be $M \times N$ (for C), $M \times K$ (for A) and $K \times N$ (for B). As mentioned in previous sections, our compiler generates several versions of CUDA code depending on the size of $M \times N$, *i.e.*, the parallelism degree detected by the compiler. The auto-tuning engine will find the best switching points after several iterations of off-line training.

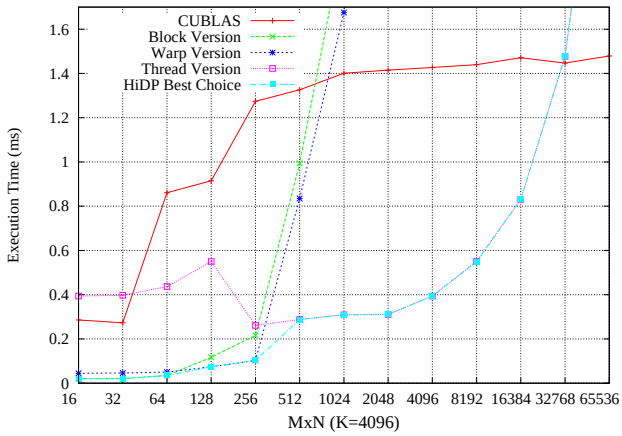


Figure 5. GEMM Results for Small+Medium $M \times N$s

Figure 5 depicts the results for double precision matrix-matrix multiply where K is 4096 while varying $M \times N$ from 16 to 65536. The figure shows the absolute execution time for each case (except some long execution time for block and warp versions at $M \times N >= 4096$). As we can see, when $M \times N <= 64$, the block version outperforms other techniques. This is because assigning an entire block (at this size) to cooperatively compute one element in

C has a better chance of saturating GPU resources than assigning one thread per element. As the parallelism ($M \times N$) increases, the warp version catches up and performs best in the range of $128 <= M \times N <= 256$. For $M \times N$ exceeding 256, HiDP's thread version outperforms the other two versions. In contrast, CUBLAS does not deliver the best performance until $M \times N$ reaches 32768. This implies that the hand-written CUBLAS assigns multiple elements per threads (*a.k.a.* thread fusion), which hurts performance for cases when $M \times N$ is small to medium.

Of course, this is by no means to say HiDP can replace the GEMM in CUBLAS. For large-scale GEMM, CUBLAS outperforms auto-generated code in HiDP by a large margin. HiDP does not intend to compete with well-refined numerical libraries. But what HiDP shows is that different code transformation strategies are suited for different data inputs, even on the same machine. It is necessary to emit a complete selection of alternatives for auto-tuning.

4.2 3D Stencil Computation

An interesting group of computations that is well-suited for GPUs are Jacobi stencil computations [9]. In stencil computation, new values of elements are updated based on old values of the local element and their neighbors. There are different neighbor access patterns for different types of stencil computation. HiDP detects such patterns and optimizes them using on-chip Shared Memory to save off-chip memory bandwidth.

We select two stencil computations (7-point and the Himeno benchmark) utilizing double-precision floating-point and compare the performance of HiDP generated code with another adaptive framework for stencil computations [23]. The details of the Himeno computation can be found in [20, 21]. Figure 6 shows how Himeno is expressed in HiDP. The main part is a map clause with a reduction suffix. The map clause gives the user customized control of map iterators, which exclude boundary indices in all three axes here. Array accesses are assumed to be in the order of the declarations of map iterators. Brackets can be omitted if the access is independent per thread. For example, $\_a0$ inside the map clause means $a0[i][j][k]$. We can see that HiDP is almost as concise as the domain specific language in [23]. By adding a reduction using the map suffix, HiDP can even generate reduction code inside the same kernel as the stencil computation, a feature not supported by other frameworks [23].

Because there is only a single-level map in HiDP and the reduction is applied to all map iterators (a global reduction), the compiler selects a kernel-level execution mode where the entire map clause becomes a CUDA kernel. The reduction at this kernel-level mode is a two-phase process involving both the GPU and CPU: each block performs a block-level reduction and then the CPU reduces all local reduction values into a single one.

The difference in performance (in GFlops) is shown in Figure 7. HiDP generates a block size of 16×16 by default.

```
function himeno
input float a0[I][J][K], a1[I][J][K], a3[I][J][K], b0[I][J][K], b1[I][J][K],
      b2[I][J][K], c0[I][J][K], c1[I][J][K], c2[I][J][K], p[I][J][K], wrk1[I][
      J][K], bnd[I][J][K], _omega;
output float _gosa, wrk2[I][J][K];
{
  map{
    i:=[1:I−2]; j:=[1:J−2]; k:=[1:K−2];
    _s0 = _a0∗ p[i+1][j][k] + _a1∗ p[i][j+1][k] + _a2∗ p[i][j][k+1] + _b0∗(
          p[i+1][j+1][k] − p[i+1][j−1][k] − p[i−1][j+1][k] + p[i−1][j
          −1][k]) + _b1∗(p[i][j+1][k+1] − p[i][j+1][k−1] − p[i][j−1][k
          +1] + p[i][j−1][k−1]) + _b2∗(p[i+1][j][k+1] − p[i+1][j][k−1]
          − p[i−1][j][k+1] + p[i−1][j][k−1]) + _c0∗ p[i−1][j][k] + _c1∗
           p[i][j−1][k] + _c2∗ p[i][j][k−1] + _wrk1;
    _ss = (_s0 ∗ _a3 − _p) ∗ _bnd;
    _wrk2 = p + _omega ∗ _ss;
    _ss2 = _ss ∗ _ss;
  }reduce('+', _gosa, _ss2, i:=[∗], j := [∗], k:=[∗]);
}
```

Figure 6. Himeno Benchmark in HiDP

In contrast, [23] uses off-line tuning to search for the best parameters for a stencil. This difference contributes to the performance difference between them. However, HiDP still manages to reach at least 70% of the GFlops performance.

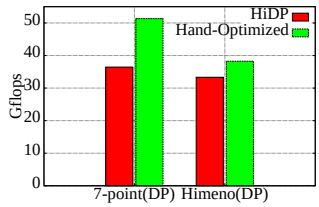
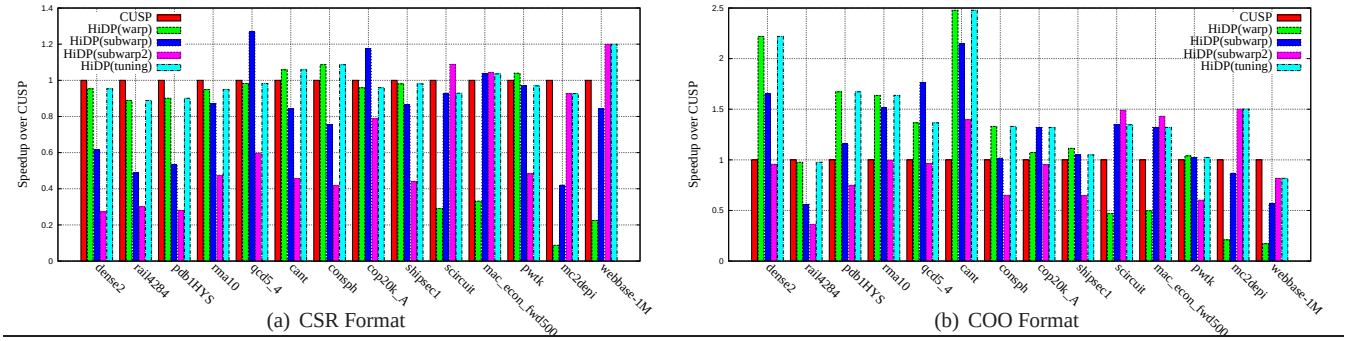


Figure 7. Comparing HiDP with Auto-Tuned Code in Stencil Computations

4.3 Sparse Matrix Vector Multiplication

A typical sparse matrix vector multiplication routine has a two-level map block where the outer level iterates over each row of the sparse matrix and the inner level iterates over each element in the same row and then performs a reduction on this row. The shape analysis generates a two-level nested shape $\{[0 : row), *\}$ for the second-level map clause. The $*$ indicates that the parallelism degree is data dependent. This uncertainty, if not further constrained by the user, leads the HiDP compiler to try several options to determine which execution mode to choose at the second level. The number of rows is used by HiDP as the parallelism degree, *i.e.*, it determines which mapping has a better chance to utilize all GPU computing resources. In the following, we show results obtained by three decisions where the inner map is executed (1) by an entire warp, (2) by a subwarp of 8 threads (subwarp) and (3) by a subwarp of 4 threads (subwarp2). As the number of rows for the sparse matrix increases, HiDP tends to use less threads per inner map.

Figure 8 depicts the speedups achieved for each choice using a hand-written sparse matrix vector library (CUSP) as the baseline. We show two representations of the sparse matrix here: CSR (Compressed Sparse Row) and COO (Coordi-



(a) CSR Format (b) COO Format

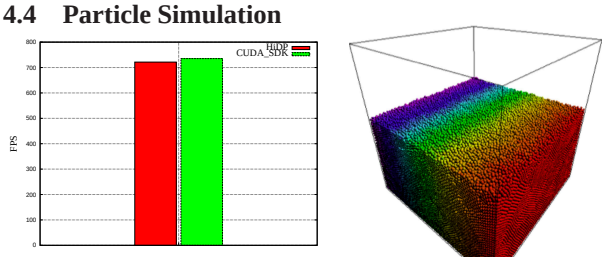
Figure 8. Sparse Matrix Vector Multiply

nate List). On the X axis, matrices are ordered by increasing number of rows from left to right. We see a clear performance benefit of using fewer threads per row as the number of rows increases, with only a few exceptions near the switching point. (In COO format, HiDP still chooses subwarp as the parallelization alternative for the scircuit matrix — even though subwarp2 is slightly faster. This is due to a significant performance loss of subwarp2 for the pwtk matrix.) With our tuning capability, HiDP delivers very close or even better performance than hand-written CUDA code.

Another observation is that HiDP performs better in COO format than CSR format in general. This is due to implementation differences between our HiDP code and CUSP for the COO format: HiDP uses an auxiliary array to convert the COO format into the CSR format and reuses the CSR kernels. In contrast, CUSP performs segmented reduction for the COO format, which turns out to be slower.

Just using the number of rows to determine the switching point is by no means optimal. The distribution characteristics (min, max and average etc.) of the number of non-zero entries in each row of the sparse matrix should affect the decision, too. The HiDP compiler, at this point, does not consider these aspects for more advanced decisions. If equipped with prior knowledge, the user has to manually choose the appropriate implementation.

4.4 Particle Simulation



(a) FPS Comparison (b) Particle Simulation

Figure 9. Particle Simulation

As a demonstration of a pipeline of kernels, we choose the particle simulation example of the CUDA SDK. We simulate collisions of 128K particles in a cube (Figure 9(b)). The core of the simulation consists of a sequence of steps: an update of particle velocities and positions, hashing, sorting and collision detection. After rewriting the algorithm into a much more concise HiDP code, the compiler emits several kernels similar to the hand-written code of the CUDA SDK. As a result, the FPS (frame per second) metric shows little difference (Figure 9(a)).

4.5 Quicksort

It is very easy to express quicksort in HiDP because HiDP supports segmented *partition* and *sort* as built-in parallel primitives (see Table 2). In HiDP, quicksort performs a few iterations of partitioning with carefully chosen pivots. (We use the average of the min and max.) In the beginning, there is only one segment. The number of segments doubles after each iteration. After the number of segments is large enough (64 or more), we finish with segmented sort in a separate map clause, which internally uses a merge sort implementation.

We compare HiDP's code with GPU-Quicksort, a hand-written CUDA sorting library using quicksort [6]. GPU-Quicksort also starts with partitioning an array into smaller segments but then switches to bitonic sort. To the best of our knowledge, it is the fastest open-source GPU implementation utilizing quicksort.

Figure 10 depicts the execution time of each implementation for 4 to 32 million unsigned integers. The input distribution is uniform. (We observed similar patterns for other input distributions.) HiDP is able to keep up with GPU-Quicksort in terms of performance. But in contrast to GPU-Quicksort, HiDP shines in coding productivity: the total number of source code lines for GPU-Quicksort, including both host-side C++ and CUDA, adds up to about 900 lines. The equivalent HiDP code is just short of 50 lines, more than an order of magnitude less.

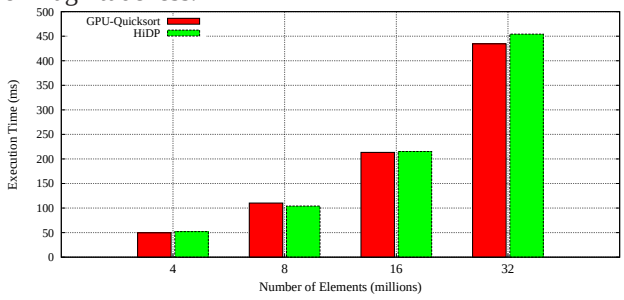


Figure 10. Quicksort

4.6 Bitonic Sort

Bitonic sort is a good show case for HiDP's support of a regular interface for parallel primitives because it always works on segments of the same size and the total size has to be a power of two. Here, we compare the performance with that of the same algorithm released by Nvidia in the CUDA SDK. In this example, auto-generated HiDP code achieves up to 80% of the performance of hand-written code (Figure 11). Similar to quicksort, bitonic sort in HiDP requires only ≈50 lines of code. In contrast, the hand-written CUDA SDK requires more than 250 lines of code.

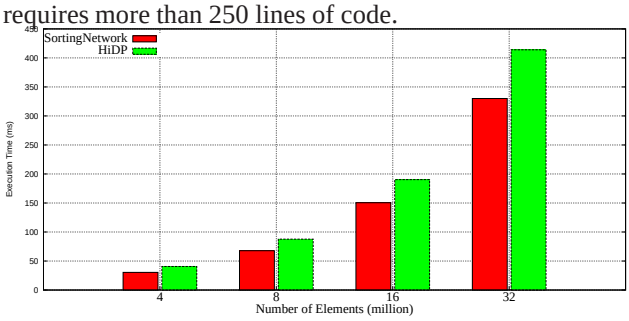


Figure 11. Bitonic Sort

5. Related Work

There are numerous propositions to extend existing languages with directional annotations. Lee *et al.* were the first to support CUDA code generation with OpenMP annotations ([18, 19]). The StarSs programming model represents a group of variants (OmpSs [10], GpuSs [3]) under a common theme: exploiting task-level parallelism via compiler pragmas on task arguments. It relies on the read-/write properties of task arguments to build a task dependency graph and creates necessary memory copies. StarPU [2] offers a unified task abstraction. Tasks can be implemented by "codelet", which targets different architectures. Both StarSs and StarPU focus on run-time scheduling of tasks and do not alleviate users from writing low-level kernels. HMPP [22] and OpenACC [17] are recent approaches to utilize OpenMP-like pragmas on parallelizable code sections, which are often do/for loops. Their optimization scope is limited to block level. Both lack auto-tuning capabilities.

On the language side, Sequoia [12] adds memory hierarchy as a first class feature in its language with task parallelism. It captures the importance of utilizing the memory hierarchy of modern architectures, which is also part of HiDP's optimization strategies. Chapel [7] and UPC [11] are parallel languages for a Single Program Multiple Data (SPMD) model of parallelism. They provide high degree of programmability with a global address space. Chapel also supports nested parallelism with mixed task and data parallelism. C++ AMP [8] extends C++ to support data-parallel accelerators. Nested parallel for loops are absent from C++ AMP, for it treats the underling accelerator as a flat architecture — unless the user uses language extensions to write kernels in a similar manner to CUDA or OpenCL. None of the above languages exploit the hierarchical execution model of GPUs to the extend that HiDP does.

The Petabricks compiler [1] provides an encapsulation of a function body that is similar to HiDP's approach. This modular design is convenient for compilers with auto-tuning capabilities. In contrast, Petabricks's tuning is for algorithmic choices. Users need to provide native code for each algorithm.

An active research topic is source-to-source compiler framework that translates well-established high-level languages (data-parallel Haskel, Python) to CUDA. Garg and Amaral [13] propose compiling techniques to convert Python loop structures and array operations to CUDA code. But to stay efficient, they require the programmer to conform to the style of the targeted language (similar to C++ AMP). Copperhead [5] conforms to Python's syntax as much as possible without introducing codelets. It advocates the mapping of nested parallel structures into a hierarchical execution model. Though this mapping can be directed by the end-user, it is static and lacks the tuning capabilities that are essential for performance, as shown in our work. CuNesl [24] is a recent compiler framework to directly convert NESL [4], a very concise data-parallel language supporting nested parallelism. Similar to HiDP, it generates CUDA/C++ code from high-level functional languages. This work identifies the necessity to convert recursive calls into iterative functions to better match today's GPU architecture. But it suffers from non-negligible performance loss and memory footprint increases. This is because the standard flattening method to convert nested parallelism into its segmented counterpart is too general and fails to efficiently utilize hierarchical resources of GPUs.

Overall, HiDP provides a low-level, hierarchical STL-like interface with data-parallel language features. The user can concentrate on algorithmic design while benefiting from the hand-crafted common primitives developed by architecture experts. The single-entry function design helps to integrate HiDP with an existing mixed language code base.

6. Conclusion

Inserting directional annotations to legacy code may be a desirable method to take advantage of new compiling and architecture features, yet such annotations limit the optimization space, and their applicability is often restricted to only selected algorithms. In practice, data structures and algorithms tend to require changes to better utilize computational resources of modern parallel architectures. High-level languages that embrace performance efficiency and coding productivity seem to be a more promising solution to improve performance.

This paper presents HiDP, a hierarchical data-parallel language designed for efficient execution on today's SIMT architectures. HiDP allows users to express algorithms as a mixture of both task-level and data-level parallelism. HiDP's

compiler performs kernel fusion based on symbolic shape analysis and integrates with common handwritten data-parallel primitives. HiDP explores various execution mappings according to the application structure and searches for appropriate dynamic switching points via auto-tuning.

HiDP's motivation reaches beyond coding productivity. Our experimental results show that HiDP is capable of achieving good performance for many application types compared to their hand-written counterparts. HiDP is an active project with a forthcoming open-source release.

References

[1] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. PetaBricks: A Language and Compiler for Algorithmic Choice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, Jun 2009.

[2] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 863–874, Berlin, Heidelberg, 2009. Springer-Verlag.

[3] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 851–862, Berlin, Heidelberg, 2009. Springer-Verlag.

[4] Guy E Blelloch and Parallel Ram Model. NESL: A Nested Data-Parallel Language. Technical report, 1993.

[5] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: Compiling an Embedded Data Parallel Language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 47–56, New York, NY, USA, 2011. ACM.

[6] Daniel Cederman and Philippas Tsigas. A Practical Quicksort Algorithm for Graphics Processors. In *Proceedings of the 16th annual European symposium on Algorithms*, ESA '08, pages 246–258, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.

[8] Microsoft Cooperation. C++ AMP : Language and Programming Model. 2012.

[9] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil Computation Optimization and Auto-tuning on State-of-the-art Multicore Architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[10] Alejandro Duran, Eduard Ayguad, Rosa M. Badia, Jess Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, pages 173–193, 2011.

[11] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, 2003.

[12] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[13] Rahul Garg and José Nelson Amaral. Compiling Python to a Hybrid Execution Environment. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 19–30, New York, NY, USA, 2010. ACM.

[14] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2010. Version 1.3.0.

[15] http://code.google.com/p/cudpp/. CUDPP.

[16] http://www.khronos.org/opencl. OpenCL.

[17] Axel Koehler. GPU Programming with CUDA and OpenACC. 2012.

[18] Seyong Lee and Rudolf Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[19] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a Compiler Framework for Automatic Translation and Optimization. *SIGPLAN Not.*, 44:101–110, February 2009.

[20] Satoshi Matsuoka, Takayuki Aoki, Toshio Endo, Akira Nukada, Toshihiro Kato, and Atushi Hasegawa. GPU Accelerated Computing from Hype to Mainstream, the Rebirth of Vector Computing. In *Journal of Physics: Conference Series 180*, 2009.

[21] E.H. Phillips and M. Fatica. Implementing the Himeno benchmark with CUDA on GPU clusters. In *International Parallel and Distributed Processing Symposium(IPDPS)*, Apr 2010.

[22] S. Bihan R. Dolbeau and F. Bodin. HMPP: A Hybrid Multicore Parallel Programming Environment. In *Workshop on General Purpose Processing on Graphics Processing Units*, Boston, MA, Oct 2007.

[23] Yongpeng Zhang and Frank Mueller. Auto-Generation and Auto-Tuning of 3D Stencil Codes on GPU Clusters. *In International Symposium on Code Generation and Optimization (CGO)*, April 2012.

[24] Yongpeng Zhang and Frank Mueller. CuNesl: Compiling Nested Data-Parallel Languages for SIMT Architectures. *In International Conference on Parallel Processing (ICPP)*, Sep 2012.