# Directive-based tile abstraction to distribute loops on accelerators

Tristan Vanderbruggen
University of Delaware
tristan@udel.edu

John Cavazos
University of Delaware
cavazos@udel.edu

Chunhua Liao
Lawrence Livermore National Laboratory
liao6@llnl.gov

Daniel Quinlan
Lawrence Livermore National Laboratory
dquinlan@llnl.gov

## ABSTRACT

Optimizing applications for the next generation of super-computers requires next generation compilers. These compilers need to provide an abstraction for the developer to describe the inner working of applications. And, next generation compilers need to be able to intelligently apply optimizations to a wide variety of algorithms solved by scientific applications. They need to optimize applications for any workload targeting any architecture. In this paper, we present an important component of any next generation super-computer compiler that we call TileK. TileK is a tile abstraction used to generate distributed kernels from nested loops. It provides a high-level abstraction used to decompose the iteration space of loop nests. Its directives-based language enables an effective and efficient placement of multi-dimensional computations on the 3D topology of accelerators (e.g. graphics processing units, GPUs). We implemented both the tile abstraction and the kernel generator in ROSE Compiler. We used TileK to parallelize linear algebra kernels and stencils, targeting multicore CPUs (pThread) and GPUs (OpenCL). TileK enabled us to explore and evaluate a large optimization space of many versions of these kernels for varying input sizes. Our results shows that the selection of a given optimization for a specific input size is a challenging problem.

## CCS CONCEPTS

**•Computer systems organization →Multicore architectures; Heterogeneous (hybrid) systems; •Software and its engineering →Source code generation; •Computing methodologies** →*Parallel programming languages;*

## KEYWORDS

compiler, heterogeneous, source-to-source, GPGPU, language

**ACM Reference format:**
Tristan Vanderbruggen, John Cavazos, Chunhua Liao, and Daniel Quinlan. 2016. Directive-based tile abstraction to distribute loops on accelerators. In *Proceedings of GPGPU-10, Austin, TX, USA, February 04-05, 2017,* 10 pages.


*GPGPU-10, Austin, TX, USA*

DOI: http://dx.doi.org/10.1145/3038228.3038238

DOI: http://dx.doi.org/10.1145/3038228.3038238

## 1 INTRODUCTION

As we progress toward exaflops computing, hardware composing supercomputers have new, less intuitive, execution models. Indeed, super-computer manufacturers often evaluate a wide range of hardware to reach the best performance for their users. The growing number and variety of accelerator-equipped systems in the TOP500 list [16] illustrate this issue. In June 2016, ninety-five systems on this list contained accelerators of five different kinds. These computation accelerators promote the host/kernel paradigm. In this paradigm, an application starts on a host and is programmed in a general purpose language, like C or C++, to be executed by a general purpose processor. The application may spawn kernels from the host. These kernels are written in a language specific to the targeted architecture. In the case of accelerators like GPGPU (General Purpose Graphic Processing Units), kernels can be programmed using OpenCL (Open Compute Language, an extension of C) [12]. In addition, many accelerators are well suited for single-program multiple-data model of parallelism [6]. In SPMD parallelism, the same program is executed by independent processors on different data.

As computing system become more complex new programming models need to be provided [11] that can handle the complexity of the machines. It is the role of programming models to define the abstractions needed to exploit the underlying hardware. Once the programming models are defined, languages or APIs are created to implement them. These languages and APIs will eventually have to be compiled down to parallel codes. An efficient way to accomplish this task is to transcribe the input program into an SPMD problem then generate kernels accordingly. This technique often involves two compiler phases: (1) the programming model's language is compiled to a generic SPMD formulation, and (2) the SPMD formulation is compiled to a kernel specialized to the underlying target architecture.

In this paper, we focused on the second phase where the difficulty is to find a technique to distribute computations across multiple instances of a kernel. As compute intensive codes are generally composed of loops, we focused our attention on methods to distribute nested loops. One such method is to use tiling [20] which subdivides the iteration space of a loop into blocks or tiles. We devised a simple method which decomposes loops into tiles then tiles are mapped to the indexes of the kernel's instances. This method

is generic and we used it to generate kernels for multicore CPUs (pThread) and for accelerator (OpenCL, CUDA).

In this paper, we present TileK: a tiling abstraction embedded in C/C++ using directives. TileK is a set of directives which enables us to perform tiling and loop interchange on nested loops. The strength of TileK is its ability to generate SPMD Kernels by mapping some of the resulting tiles to different processors. These kernels can be distributed with pThreads or OpenCL. TileK was implemented using ROSE Compiler [13], a source-to-source compiler developed at the Lawrence Livermore National Laboratory. First, we present TileK's programming model. Second, we introduce the tiling method and associated transformations applied to the kernel. Third, we show how to use this method to distribute the computation in SPMD Kernels. Finally, we conduct an optimization-space exploration where linear-algebra kernels and stencils are tiled and distributed using TileK. The various versions of these codes are evaluated for many different input sizes. We show that the selection of a given optimized version is non-trivial and that the "one version fit all" approach [8] is sub-optimal when faced with both a large optimization space and a large input space.

## 2 TILEK

TileK is a programming model extending C/C++ through compiler directives. It enables us to define regions of code that have to be offloaded into kernels. In these offloaded regions of code, loops can be decomposed into tiles. These tiles will either appear as loops in the generated kernel or will be distributed across a multicore CPU or accelerators. Figure 1 depicts the compilation flow of TileK.

### 2.1 TileK language

TileK has two constructs: *kernel* and *loop*. Kernel constructs are used to designate a region of code that will be transformed into a kernel. The *kernel* construct is associated with *data* clauses specifying the data used by the kernel. The loop construct is used to mark loops that have to be tiled. The *loop* construct is associated with *tile* clauses. We show an example in Listing 1.

```
#pragma tilek kernel data(A[0:n][0:m],\
                          mode:rw,\
                          live:inout)
{
#pragma tilek loop tile[0](static, 2) \
                   tile[2](dynamic)
for (i = 0; i < n; i++)
  #pragma tilek loop tile[1](static, 3) \
                     tile[3](dynamic)
  for (j = 0; j < m; j++)
    A[i][j] += b;
}
```

**Listing 1: We use this simple kernel to illustrate TileK and associated code transformation. It adds a constant value to every elements of a matrix. The TileK annotations are used to specify a kernel, its data, and the way loops are transformed.**

First, the region of code that has to be transformed into a kernel is marked using the *kernel* construct: `kernel data(A[0:n][0:m], mode:rw, live:inout)`. The associated *data* clause specifies that `A[0:n][0:m]` is read and written by the kernel (`mode:rw`) and lives in and out of the kernel (`live:inout`). `A[0:n][0:m]` represents the array $A$ of dimension $n \times m$. Array dimensions are used for data transfers and for array flattening (linearizing). The additional access mode and liveness information are used for data placement (using eventual constant memory) and data movements. Second, loops in this region of code are annotated with *loop* constructs: `loop tile[0](static, 2) tile[2](dynamic)` and `loop tile[1](static, 3) tile[3](dynamic)`. Loop constructs hold *tile* clauses which are of the form: `tile['order'](static, 'trip-count')` or `tile['order'](dynamic)`. For *static* tiles, the trip-count is known at compile time, but the trip-count of *dynamic* tiles is determine at runtime. The tiles composing a loop are such as the product of their trip-count is equal to the trip-count of the original loop. At this point, the compiler expects evenly divisible loops. Allowing loops that are not evenly divisible is possible, but would require to add guards in the generated code. Adding these guards would add computations and could cause divergence which is costly on some accelerator (e.g. GPGPU). There can only be one dynamic tile else the system cannot be solved. When the kernel is generated, tiles from perfectly nested loops are sorted based on *order*, allowing the compiler to perform interchange.

### 2.2 Iteration Domain

TileK enables to change the order in which the iteration domain of a loop nest is traversed. Figure 2 shows how the iteration domain is separated between tiles. Each cell represents the iteration of coordinates $(i, j)$. For example, cell $(3, 4)$ executes `A[3][4] += b;`. The clauses `tile[0](static, 2)` and `tile[1](static, 3)` split the iteration domain in six subdomains (marked by the bold red lines). And the clauses `tile[2](dynamic)` and `tile[3](dynamic)` iterate over each of these subdomains. The tiles are executed starting with the outer-tiles first (`i` loop then `j` loop), then the inner-tile are executed (same `i` then `j`). The sequence of iterations is $(0, 0)$, $(0, 1)$, $(0, 2)$, $(0, 3)$, $(1, 0)$, ..., $(1, 3)$, $(0, 4)$, ..., $(0, 7)$, $(1, 4)$, ..., $(1, 7)$, $(2, 0)$, ..., $(2, 3)$, $(3, 0)$, ...

### 2.3 Distributed Kernels

TileK has two distinct extensions to generate SPMD kernels: *Threads* and *Accelerator*. Both introduce a special kind of tile that will be mapped either to software threads or the execution units of an accelerator. We will present the syntax in the TileK language and the associated execution model.

*2.3.1 TileK Threads.* TileK *Threads* extension permits distributing computations across multiple processors or cores. This is done by mapping a tile to the thread dimension, as shown in Listing 2. The clause `num_threads` declare the number of threads that the host will fork. This number of threads does not have to be known at compile time. The computations are distributed based on the `tile(thread)`. Figure 3 shows how the iteration domain is distributed between the different threads.

*2.3.2 TileK Accelerator.* Using the TileK *Accelerator* extension, kernels are offloaded on an accelerator using OpenCL. TileK's Accelerator model uses OpenACC's model for the two levels of parallelism: *gang* and *worker*. *Gang* represents coarse grain parallelism while *worker* represents fine-grain parallelism. Global memory is accessible by all workers from all gangs while workers can only access local memory associated with their gang. Listing 3 shows an example of code annotated with TileK's *Accelerator* extension. The clauses *num_gangs* and *num_workers* declare the size of the abstract

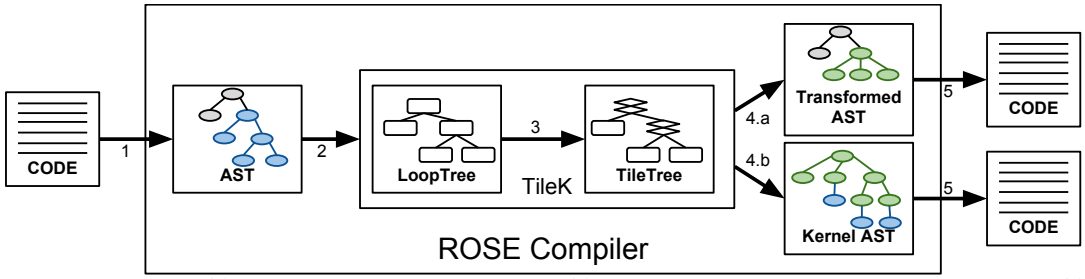



**Figure 1: TileK's compilation flow: (1) ROSE parses the input C/C++ code, (2) TileK extracts a representation of the loop nests, (3) TileK applies tiling based on annotations, (4.a) TileK transforms the application's AST to instantiate the kernel and move data, (4.b) Tilek generates the AST of the kernel for the desired target: pThread, CUDA, or OpenCL, (5) ROSE unparses the AST into C/C++/CUDA/OpenCL code.**

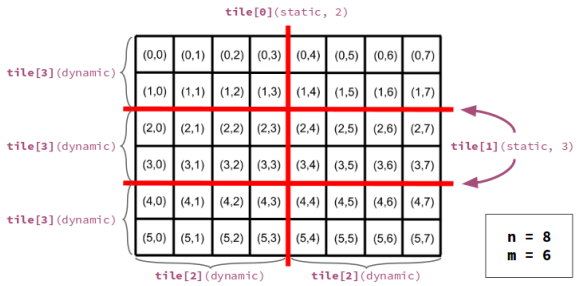



**Figure 2: This figure illustrates the iteration domain for Listing 1 when $n = 8$ and $m = 6$. The static tiles become the outer most loops and iterate over the blocks formed by the dynamic tiles.**

```
#pragma tilek kernel data(...) num_threads(4)
{
#pragma tilek loop tile(thread) \
                   tile[1](dynamic)
for (i = 0; i < n; i++)
  #pragma tilek loop tile[2](dynamic)
  for (j = 0; j < m; j++)
    A[i][j] += b;
}
```

**Listing 2: *Threads* extension of TileK applied to the motivating example. The `num_threads` clause was added to the `kernel` construct to determine the number of threads used by the generated kernel. The `tile(thread)` declares a tile that will be mapped to the spawned threads.**

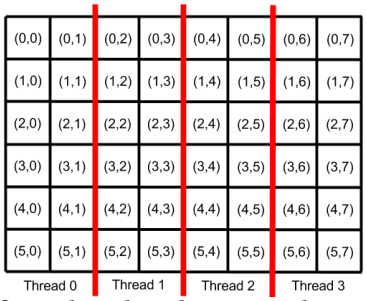



**Figure 3: This figure shows how the iteration domain in Listing 2 is shared among four threads. Each thread computes a block of $2 \times 8$ elements.**

accelerator. Computations are distributed on this abstract machine using *gang* and *worker* tiles. The trip-count for *gang* and *worker* tiles is given by the corresponding *num_gangs* and *num_workers*

```
#pragma tilek kernel data(...) \
        num_gangs[0](2) num_gangs[1](2) \
        num_workers[0](4) num_workers[1](4)
{
#pragma tilek loop tile(gang, 0) \
                   tile(worker, 0) \
                   tile[0](dynamic)
for (i = 0; i < n; i++)
  #pragma tilek loop tile(gang, 1) \
                     tile(worker, 1) \
                     tile[1](dynamic)
  for (j = 0; j < m; j++)
    A[i][j]+=b;
}
```

**Listing 3: This code is the same as Listing 1, but uses the TileK *Accelerator*. This code instantiates an abstract accelerator made of $2 \times 2$ gangs of $4 \times 4$ workers.**

clauses. On both loops, the inner-most tiles are dynamic. So, each worker in the abstract machine will execute a block of the iteration domain.

## 3 GENERATING TILED KERNELS

The first step to generate SPMD kernels is to generate tiled kernels. These kernels correspond to codes like Listing 1. In this case, the computation are not distributed but the loops are tiled and the resulting tiles are reordered. In TileK's compiler, the generation of tiled kernel is done through a two step process: (1) the LoopTree generator converts Rose's IR into the LoopTree IR where tiling is done, and (2) the Kernel generator that converts LoopTrees into optimized ASTs that can be fed back into Rose for further transforming improvements.

We will first present TileK's Internal Representation (IR) called LoopTree. Next, we describe the kernel generated for Listing 1. Finally, we will introduce the algorithm used at runtime to determine the length and stride of each tiles.

### 3.1 LoopTrees

When compiling code annotated with TileK, the region associated to the kernel construct is transformed into a *LoopTree*. They are a simplified Abstract Syntax Trees (ASTs) that can be specifically optimized in our TileK compiler. Any block of code that contains a restricted, but large and common set of for-loops, if-statements, and expression-statements can be model into *LoopTrees*. Here, our restricted set of for-loops corresponds to loops of the form `for (i = lb; i < ub; i += s)` and variations, including: greater-than and non-strict comparison operator, or decrement/increment operators. `lb`, `ub`, and `s` need to be constant across the lifetime of the loop, but do not need

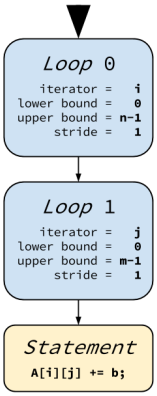



**Figure 4: When directive are not considered, Listing 1 translates into a simple *LoopTrees*. The *i* loop is identified as loop 0 which iterates from 0 to n-1 with a stride of 1. The *j* loop is identified as loop 1 which iterates from 0 to m-1 with a stride of 1 and loop 0 is its parent. The expression statement appears as the leaf of the tree and loop 1 is its parent.**

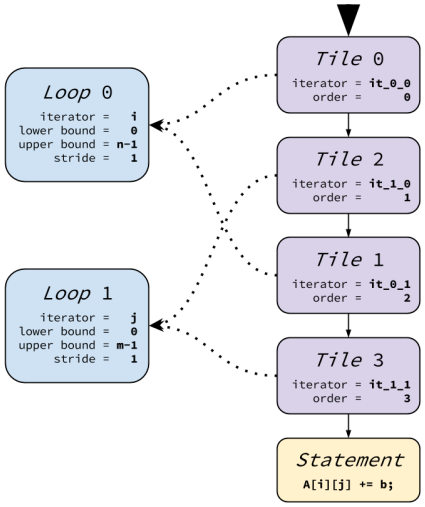



**Figure 5: When the directives are considered, the *LoopTrees* for Listing 1 is more complicated than in figure 4. Both loops are separated into two tiles. Loop 0 becomes tile 0 and tile 1. Loop 1 becomes tile 2 and tile 3. The order of the tiles in the LoopTree depends on the *order* argument (between square brackets).**

to be known at compile time. We note that many popular scientific algorithms and most scientific kernels we have encountered fall into our constrained set of for-loops. The IR uses *fragments* of ROSE's AST to represent variables (symbols), expressions, and statements. Figure 4 shows the *LoopTree* constructed from Listing 1 without considering the annotations. For each loop in the example, the *iterator* was identified alongside the *lower bound*, *upper bound*, and *stride*. In Figure 5, the *LoopTree* was modified according to the TileK annotations. The tile clauses on TileK's loop constructs lead to the creation of tile nodes that take the place of the loop nodes. *Iterators* are constructed for each tile and tiles are sorted depending of their *order* parameter. Each tile is linked to the loop it originates from, providing the bounds and stride. *LoopTrees* are used to generate the kernel, however, we associate to each kernel a copy of its *LoopTree*. This *LoopTree* fully characterizes the kernel and present a few more advantages:

- it uses pieces of the original AST facilitating the generation of correct ASTs
- it has a higher level of abstraction than the original AST simplifying transformations
- it could be used to handle non traditional loops with minimal change to the subsequent transformations

## 3.2 Generated Kernel

```
void klt_kernel_0(
 void ** param, void ** data,
 struct klt_loop_context_t * loop_ctx
) {
 float *A = (float *)data[0];
 float b =  *((float *)param[2]);
 int m =  *((int *)param[1]);
 int n =  *((int *)param[0]);
 int l_0, l_1;
 int t_0, t_1, t_2, t_3;
 for (t_0 = 0;
      t_0 < klt_get_tile_length(loop_ctx,0);
      t_0 += klt_get_tile_stride(loop_ctx,0))
  for (t_2 = 0;
       t_2 < klt_get_tile_length(loop_ctx,2);
       t_2 += klt_get_tile_stride(loop_ctx,2))
   for (t_1 = 0;
        t_1 < klt_get_tile_length(loop_ctx,1);
        t_1 += klt_get_tile_stride(loop_ctx,1))
    for (t_3 = 0;
         t_3 < klt_get_tile_length(loop_ctx,3);
         t_3 += klt_get_tile_stride(loop_ctx,3)) {
     l_0 = klt_get_loop_lower(loop_ctx,0)
                                  + t_1 + t_0;
     l_1 = klt_get_loop_lower(loop_ctx,1)
                                  + t_3 + t_2;
     A[l_0 * m + l_1] += b;
    }
}
```

**Listing 4: Sequential kernel generated for Listing 1. Both loops are separated into two tiles. The resulting tiles are reordered, grouping the outer tiles and inner tiles together.**

Once the *LoopTree* has been tiled, the Kernel generator traverses it and generates a new AST for ROSE. This generated AST includes all the intricacies of the final kernel. It connects to TileK's runtime support and takes care of lowering parameters and data accesses. This AST is finally unparsed using ROSE backend.
An example of the final kernel is shown in Listing 4. It was generated from the code in Listing 1. This kernel receives two arrays of pointers, one for the parameters and one for the data. These pointers are casted to their original types inside the kernel. Each tile from the *LoopTree* leads to the generation of one for-loop. The length and stride of this for loop is obtained from the `loop_ctx` object of type `struct klt_loop_context_t`. Because tiles from multiple loops can be ordered randomly if the loops are perfectly nested, the iteration of the associated loops are determined at the inner-most loop body for this loop nest. At any time, the value of a loop's iterator is the sum of its lower bound and the tiles composing the loop. Any further optimizations are left to the backend compiler.

## 3.3 From Loop Bounds to Tile Bounds

We cannot determine the length and stride of the tiles at compile time. We need to wait until the bounds and strides of the loops

**Algorithm 1** Given a loop's bounds and stride, its number of tiles, and the kind and trip-count of each tile, this algorithm computes the length and stride for each tile.

$L \leftarrow upper^{in} - lower^{in}$
$i \leftarrow 0$
**while** $i < num\_tile^{in}$ **and** $kind_i^{in} = static$ **do**
  $length_i^{out} \leftarrow L$
  $stride_i^{out} \leftarrow L / trip\_count_i^{in}$
  $L \leftarrow stride_i^{out}$
**end while**
$S \leftarrow stride^{in}$
$j \leftarrow num\_tile^{in} - 1$
**while** $j >= i$ **and** $kind_j^{in} = static$ **do**
  $length_j^{out} \leftarrow S * trip\_count_j^{in}$
  $stride_j^{out} \leftarrow S$
  $S \leftarrow length_j^{out}$
**end while**
**if** $kind_j^{in} = dynamic$ **then**
  $length_j^{out} \leftarrow L$
  $stride_j^{out} \leftarrow S$
**end if**

are known at runtime. Before launching a kernel, Algorithm 1 determines the length and stride of all the tiles of one loop. For this algorithm, tiles are sorted following their declaration order (not the *order* parameter). The inputs of the algorithm are: (1) bounds and stride of the loop ($upper^{in}$, $lower^{in}$, and $stride^{in}$), (2) number of tiles ($num\_tile^{in}$), and (3) kind and trip-count for each tile (arrays $kind^{in}$ and $trip\_count^{in}$). The outputs are the length and stride for each tile (arrays $length^{out}$ and $stride^{out}$). The algorithm goes from outer-most tile to the inner-most. It propagates the loop length dividing it by the trip-count of each tiles. It stops when/if it encounters a dynamic tile. Then, if there was a dynamic tile, it goes from inner-most tile to the outer-most. It propagates the loops stride multiplying it by the trip-count of each tile.

## 4 GENERATING SPMD KERNELS

Section 3 presented the generation of a tiled kernel. In this section, we explain how this process can be used to build SPMD kernels. These kernels distribute computations across threads or an accelerator.

### 4.1 Kernel Index to Tile Iteration

In Section 2.3, we presented the machine models associated with TileK's *Thread* and *Accelerator* extensions. In both of these extensions, the machine is composed of multiple execution units. Each of these execution units is identified by a unique index. In the case of the *Thread* extension, the indexes are made of one integer between zero and the number of spawned threads. For the *Accelerator* extension, the dimension of the index varies depending on the gang and worker dimensions. This index space can have two, four, or six dimensions ; from one gang and one worker dimension to three gang and three worker dimensions.
For a given kernel, there will be as many distributed tiles as there is dimensions in the index. Kernels are associated with a single index (one instance of the kernel per execution unit), and each of the distributed tiles take only one value per execution of the kernel. Hence, distributed tiles are not loops in the generated kernel. Instead, the current iteration of a distributed tile is determined using its stride and the kernel's index.

### 4.2 Threads

```
void klt_kernel_0(
 int tid, void ** param, void ** data,
 struct klt_loop_context_t * loop_ctx
) {
 float *A = (float *)data[0];
 float b =  *((float *)param[2]);
 int m =  *((int *)param[1]);
 int n =  *((int *)param[0]);
 int l_0, l_1;
 int t_0 = tid * klt_get_tile_stride(loop_ctx,0);
 int t_1, t_2;
 for (t_1 = 0;
      t_1 < klt_get_tile_length(loop_ctx,1);
      t_1 += klt_get_tile_stride(loop_ctx,1) ) {
   for (t_2 = 0;
        t_2 < klt_get_tile_length(loop_ctx,2);
        t_2 += klt_get_tile_stride(loop_ctx,2) ) {
     l_0 = klt_get_loop_lower(loop_ctx,0) + t_0 + t_1;
     l_1 = klt_get_loop_lower(loop_ctx,1) + t_2;
     A[l_0 * m + l_1] += b;
 }
}
```

**Listing 5: Kernel generated when distributing the code from the motivating example using TileK's Thread extension (Listing 2). The variable `tid` represents the Thread ID, and it is used to determine the current iteration of the thread tile.**

In TileK's *Threads* extension, generated kernels are distributed between a group threads. The kernel is launched once for each threads and receives a *thread ID* (`tid`). The iteration of the thread tile is equal to the product of `tid` and the tile's stride. Listing 5 shows the kernel generated for the annotated code in Listing 2. The distributed tile iteration `t_0` is computed at line 10.

### 4.3 Accelerator

```
__kernel void klt_kernel_0(
  int n, int m, float b, __global float *A,
  __constant struct klt_loop_context_t *loop_ctx
) {
  int l_0, l_1, t_2, t_5;
  int t_0 = get_group_id(0)
                  * klt_get_tile_stride(loop_ctx,0);
  int t_1 = get_local_id(0)
                  * klt_get_tile_stride(loop_ctx,1);
  int t_3 = get_group_id(1)
                  * klt_get_tile_stride(loop_ctx,3);
  int t_4 = get_local_id(1)
                  * klt_get_tile_stride(loop_ctx,4);
  for (t_2 = 0;
       t_2 < klt_get_tile_length(loop_ctx,2);
       t_2 += klt_get_tile_stride(loop_ctx,2))
    for (t_5 = 0;
         t_5 < klt_get_tile_length(loop_ctx,5);
         t_5 += klt_get_tile_stride(loop_ctx,5)) {
      l_0 = klt_get_loop_lower(loop_ctx,0)
                                  + t_1 + t_0 + t_2;
      l_1 = klt_get_loop_lower(loop_ctx,1)
                                  + t_4 + t_3 + t_5;
      A[l_0 * m + l_1] += b;
    }
}
```

**Listing 6: Code generated for Listing 3 using TileK *Accelerator* with OpenCL backend.**

In TileK's *Accelerator* extension, the kernel is offloaded to an accelerator using OpenCL. Depending on the number of gangs and workers required by the kernel, a specialized kernel using 1D, 2D, or 3D arrays is launched.. TileK Accelerator maps gangs to work-groups and workers to work-items of the OpenCL NDRange. Given this mapping, the index used to calculate the iteration of `tile(gang,1)` is OpenCL's `get_group_id(1)`. Similarly, `tile(worker,0)` maps to `get_local_id(0)`. In Listing 6, we show the kernel generated for the annotated code in Listing 3. The distributed tiles from the outer loop of the original code (`i` loop in Listing 3) are associated to variables `t_0` and `t_1`. These variable are assigned on lines 6 and 8 using OpenCL's group ID and OpenCL's local ID for the gang tile and the worker tile, respectively. Similarly, `t_3` and `t_4` are the tiles from the `j` loop in Listing 3 and are assigned at lines 10 and 12.

TileK's *Accelerator* extension can also be used to generate CUDA code. The CUDA code generated by TileK follows the same structure as the genrated OpenCL. Given that TileK was design with flexibility in mind, switching from OpenCL to CUDA is only a matter of using a different object. These objects represent the *target language*, including C, OpenCL, and CUDA. Since these languages are based on C, code generation only differ by a few line of code ; e.g. selecting the correct qualifier for a constant variable, global memory, or kernel function.

## 5 OPTIMIZATION SPACE EXPLORATION

TileK allows one to describe many different versions of any given loop nest. This section motivates the need for predictive modeling to generate optimized kernels. The goal of this modeling would be to accelerate an iterative compilation process [2, 4]. We also make a hypothesis that such iterative compilation will have to be input aware [7, 10]. A predictive model that is input aware uses features of the input, especially array sizes, to generate the best version of a given code. To illustrate our point, we evaluate the performance of many versions of four simple computation kernels:

- **2d-conv**: 2D convolution (Listing 7a)
- **sgemm**: single precision general matrix multiply (Listing 7b)
- **syr2k**: symmetric rank-2k operations (Listing 7c)
- **doitgen**: multiresolution analysis kernel (MADNESS) (Listing 7d)

These four kernels all have two parallel outer loops, as can be seen in Listing 7. We distributed these two loops with the *Threads* and the *Accelerator* extensions of TileK. For each kernel, we evaluated eight *threads* versions and forty *accelerator* versions. For both the thread and accelerator experiments, we only measure the time it takes for the kernel to run, ignoring initialization and data-transfers, which are not optimized. Each kernel is executed five times and the average of the five measurements is used. We compare the different optimized versions by looking at the number of floating point operations per second (flops) that they compute. Given that we are comparing four different computations for many different inputs, flops is the best measurement when optimizing runtime.
Varying the inputs for these codes is simple: none of the controls are data-dependent. As there is no loop or conditional depending on the values of the inputs, only the input's size influences the runtime of a given version. For each version, we evaluated a few hundred input sizes depending on the kernel. Table 1 summarizes the input space for each of the four codes used in the experiments.

| | parameters | float operations | # inputs |
|---|---|---|---|
| **2d-conv** | n,m | $17.n.m$ | 225 |
| **sgemm** | n,m,p | $n.m.(6.p+1)$ | 1152 |
| **syr2k** | n,m | $n^2.(6.m+1)$ | 252 |
| **doitgen** | r,q,p | $2.r.q.p^2$ | 250 |

**Table 1: Summary of the input space for each of the four codes used in the experiments. It shows the parameters (positive integers) that define the input sizes of the different problems, the formula which gives the number of floating point operations as a function of the parameters, and the number of inputs (combinations of parameters) that was evaluated.**

These experiments were conducted using Elastic Cloud Compute (EC2) from Amazon Web Services (AWS). We used compute instances with 16 VCores (c4.4xlarge) to evaluate the *Threads* extension and GPU instances (g2.2xlarge) for the *Accelerator* extension. The c4.4xlarge instances are equipped with Intel Xeon E5-2666 v3 (Haswell) processors and 30 GiB of RAM. The g2.2xlarge instances are equipped with NVIDIA Grid K520 GPU (Kepler architecture with 1,536 CUDA cores and 4GB). OpenCL was used to target the GPU, using both our OpenCL implementation of the TileK runtime and the corresponding OpenCL target. The g2.2xlarge instances used OpenCL 1.2 CUDA with the NVIDIA driver version 352.79. All codes were compiled using gcc 4.8.3 and the -O3 optimization flag.

### 5.1 Thread Experiments

For each of the four codes, we generated eight threaded versions of the computation kernel. The loops `i` and `j` are annotated with one dynamic tile each. Then a *thread* tile is placed either before or after the dynamic tile of one of the loops. Finally the dynamic tiles can be interchanged using the order parameter (square bracket on the tile clause). Table 2 summarizes the eight tile configurations used to parallelize the two outer loops of each of the four codes. For example, the tiling in Listing 2 corresponds to the first tile configuration in Table 2. The results of our experiments with the thread versions of these four codes are presented in the top row of Figure 6. These graphs present the percentage of peak performance of four versions of each code for various inputs sizes. The percentage of peak performance compare a version performance for one input to the performance of the best version for the same input. We show versions 1, 3, 7, and 8 from Table 2. These versions are the two best (1 and 3) and the two worst (7 and 8). On the X-axis inputs are sorted based on increases in performance, independently for each version.
We can draw a few observations from these graphs:

- No version is the best for all inputs: version 3 for **2d-conv** performs approximately 65% worse than another version for at least one input
- The best version depends on the code being optimized: version 3 is generally the best for **2d-conv** while it is version 2 for **sgemm**

```
void 2dconv(
  int n, int m,
  float ** A, float ** B
) {
  int i, j;

  for (i = 1; i < n-1; i++) {
    for (j = 1; j < m-1; j++) {
      B[i][j] = 2.34 * A[i-1][j-1]
              + 4.64 * A[i+1][j+1]
              - 7.93 * A[i-1][j  ]
              + 1.26 * A[i  ][j-1]
              + 0.64 * A[i+1][j  ]
              + 1.34 * A[i  ][j+1]
              - 2.17 * A[i-1][j+1]
              + 7.22 * A[i+1][j-1]
              - 9.59 * A[i  ][j  ];
    }
  }
}
```

**(a) 2d-conv: 2D Convolution with random, hard-coded, floating coefficients**

```
void sgemm(
  int n, int m, int p,
  float ** A, float ** B, float ** C,
  float alpha, float beta
) {
  int i, j, k;

  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      C[i][j] *= beta;
      for (k = 0; k < p; k++) {
        C[i][j] += alpha * A[i][k] * B[k][j];
      }
    }
  }
}
```

**(b) sgemm: single precision general matrix multiply**

```
void syr2k(
  int n, int m,
  float ** A, float ** B, float ** C,
  float alpha, float beta
) {
  int i, j, k;
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      C[i][j] *= beta;
      for (k = 0; k < m; k++) {
        C[i][j] += alpha * A[i][k] * B[j][k];
        C[i][j] += alpha * B[i][k] * A[j][k];
      }
    }
  }
}
```

**(c) syr2k: symmetric rank-2k operations**

```
void doitgen(
  int R, int Q, int P,
  float *** A, float *** sum, float ** C4
) {
  int r, q, p, s;

  for (r = 0; r < R; r++) {
    for (q = 0; q < Q; q++) {
      for (p = 0; p < P; p++) {
        sum[r][q][p] = 0;
        for (s = 0; s < P; s++)
          sum[r][q][p] += A[r][q][s] * C4[s][p];
      }
      for (p = 0; p < P; p++)
        A[r][q][p] = sum[r][q][p];
    }
  }
}
```

**(d) doitgen: multiresolution analysis kernel (MADNESS)**

**Listing 7: These listing show the four codes used to evaluate TileK: 2d-conv, sgemm, syr2k, and doitgen. For each of these codes, the two outer loops are parallel. For each code, the loop bodies are different showing the flexibility of our TileK compiler. We have a stencil expression stencil expression for 2d-conv, reduction loops for sgemm and syr2k, and multiple reductions and a copy loop for doitgen.**

| # | parallelized loop | before/after dynamic tile | dynamic tiles order |
|---|---|---|---|
| 1 | i | before | i - j |
| 2 | i | before | j - i |
| 3 | i | after | i - j |
| 4 | i | after | j - i |
| 5 | j | before | i - j |
| 6 | j | before | j - i |
| 7 | j | after | i - j |
| 8 | j | after | j - i |

**Table 2: The eight tile configurations used to evaluate the TileK *Threads* extension. The first column gives the version's ID. The second column specifies which of the two loops is annotated with the `tile(thread)` clause. The third column tells whether this thread tile is placed before or after the dynamic tile. Finally, the fourth column shows the order in which the two dynamic tiles are unparsed (interchange).**

- Even poorly performing versions can have at least one input for which they perform as well as the best version

For all four codes, versions 7 and 8 perform poorly. This behavior is easily explained as these versions have their parallel tile as the inner tile of the innermost loop. Distributing the loop nest in this fashion almost always reduce the locality of the kernel. Mostly because when translating an algorithm in C, we order the loops in the same order as the dimensions of the principal array. It means that the inner loop usually accesses rows with the most locality. Hence, placing the parallel tile on the inner tile of the innermost loop, breaks this locality.

## 5.2 Accelerator Experiments

While the TileK *Threads* extension only provides one dimension to distribute the loop nest, the *Accelerator* extension enables up to six dimensions, thus giving us a large search space of code versions, i.e., forty in our case. In these experiments, we consider the two machine models for the Accelerator extension shown in Table 3. These machine models include 1D and 2D configurations of gangs and workers. For both of these machine models, we evaluate the performance of different tiling configurations. To construct these tiling configurations, we start with one dynamic tile for each loop. For the 1D machine model, we add one gang tile and one worker tile. We place one of each on the *i* and *j* loops, either before or after the dynamic tile. Having either the gang or worker tile on the *i* loop (and vice-versa for the *j* loop) creates two possibilities. The position of the gang and worker tile in relation to the dynamic tiles creates

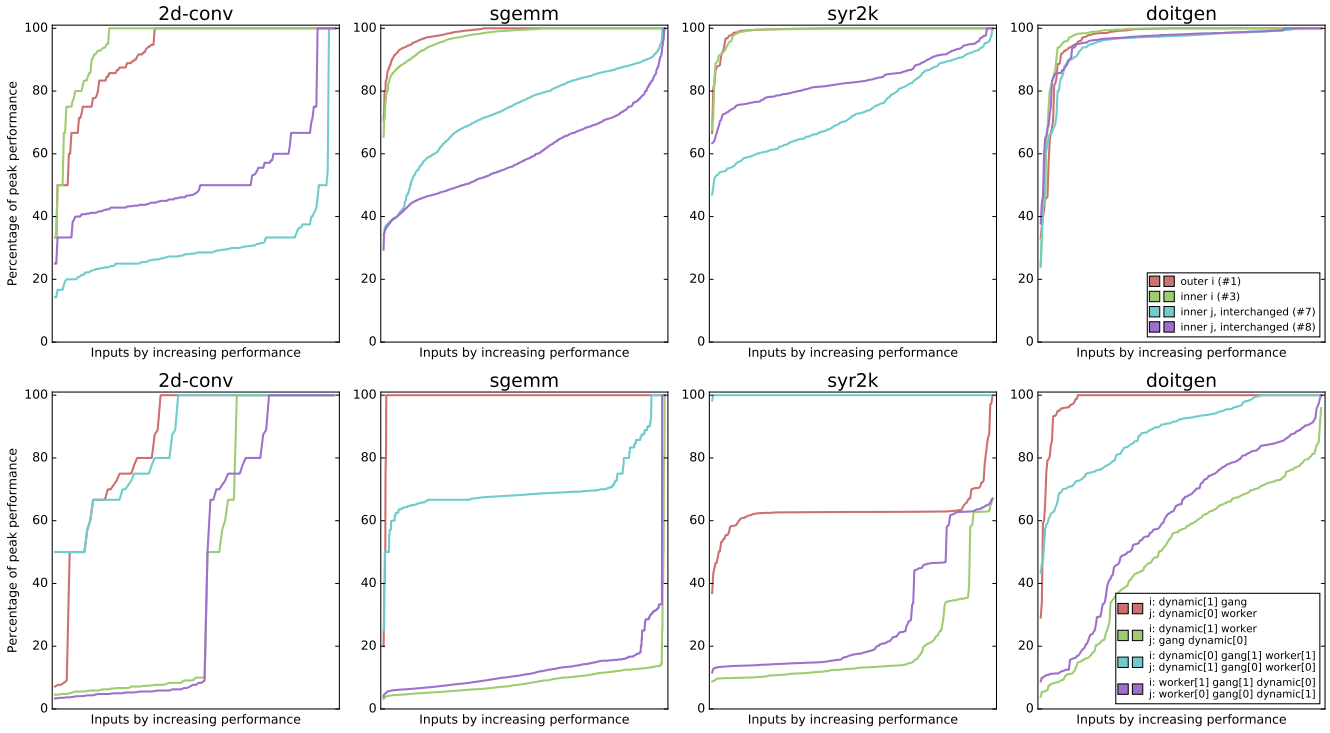



**Figure 6: Inputs space exploration for TileK threads and accelerator. X-axis represent the various inputs sorted by increasing performance (independently for each version). Y-axis show the percentage of peak performance reached by each version for the different inputs. The percentage of peak performance compare a version performance for one input to the performance of the best version for the same input. The top charts presents the results for TileK thread. Version numbers from Table 2 are provided. The bottom charts presents the results for TileK accelerator. Each version is described using the tile applied to the two loops.**

four possibilities. Finally, the dynamic tiles can be interchanged creating another two possibilities for a total of 16 different tiling configurations.

For the 2D machine model, we add two gang tiles and two worker tiles. Each of the two loops receives either gang and worker #0 or gang and worker #1, creating two possibilities. Then, there are six ways to order the dynamic, gang, and worker tiles. Finally the dynamic tiles can be interchanged creating another two possibilities for a total of 24 tiling configurations.

This search space is not exhaustive, search spaces for both 1D and 2D machine models can be extended and other machine models can be tested using various numbers of gangs and workers. We

| gangs | workers | # tiling |
|---|---|---|
| (16,) | (256,) | 16 |
| (4,4) | (16,16) | 24 |

**Table 3: We evaluated two different machine models: 1D and 2D. The total number of gangs is always 16. The total number of workers is always 256.**

evaluated each of the forty versions of the four codes on GPUs. The bottom row of graphs in Figure 6 highlights some results from four representative versions. The X-axis is ordered based on increases in performance. The Y-axis shows the percentage of peak performance. Two of the four versions shown use a 1D topology and the other use a 2D topology. The first version (based on the legend's order) uses a 1D topology. It has the gang and worker as the innermost tile of the $i$ and $j$ loops, respectively. This configuration is more efficient than the configuration of the second version as it increases the memory coalescence (neighboring execution units accesses neighboring memory cells). Similarly, the third configuration, which uses a 2D topology, has the two worker tiles as innermost tiles. Again this configuration has more memory coalescence than the fourth configuration.

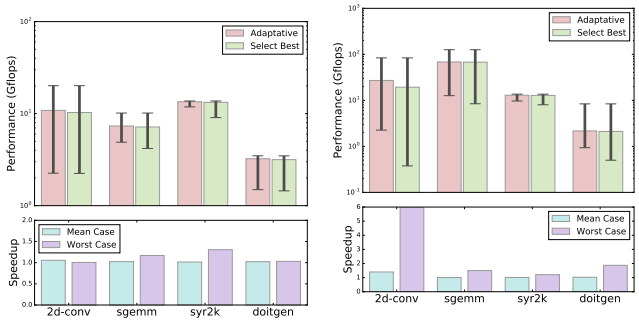
Across the four codes, there a few noticeable effects especially when looking into the first and third versions. These two versions have very different performance profile for **sgemm** and **syr2k**. In the case of **sgemm**, the first version is almost always the best. This could be due to the flattening of the 2D topology when OpenCL threads are mapped to the execution units. For **syr2k**, the third version is almost always the best. In this case, the symmetrical distribution of gangs and workers across both loops is advantageous to the symmetrical nature of the **syr2k** kernel.

## 5.3 Adaptive Kernels

In this section, we use our experiments to motivate the need for adaptive kernels. We can create an adaptive kernel that is made of multiple versions of a kernel and rules which select one version given an input. Figure 7 presents graphs and Table 4 gives raw numbers when we compare an "ideal" adaptive kernel against a "average best" kernel. By an "ideal" adaptive kernel, we mean a

| | | (A) Adaptive Kernel | | | (B) Best Performing Version on Average over all Inputs | | | Speedup (A/B) | |
|---|---|---|---|---|---|---|---|---|---|
| | | Mean | Min | Max | Mean | Min | Max | Mean | Worst |
| Threads | **2d-conv** | 10.85 | 2.25 | 20.13 | 10.28 | 2.24 | 20.13 | 1.06 | 1.01 |
| | **sgemm** | 7.34 | 4.89 | 10.15 | 7.16 | 4.19 | 10.15 | 1.02 | 1.17 |
| | **syr2k** | 13.46 | 11.84 | 13.73 | 13.27 | 9.06 | 13.73 | 1.01 | 1.31 |
| | **doitgen** | 3.24 | 1.49 | 3.48 | 3.16 | 1.45 | 3.48 | 1.02 | 1.03 |
| Accelerator | **2d-conv** | 26.70 | 2.25 | 83.73 | 18.04 | 0.28 | 83.73 | 1.48 | 8.00 |
| | **sgemm** | 68.30 | 12.65 | 126.16 | 67.71 | 2.37 | 126.16 | 1.01 | 5.33 |
| | **syr2k** | 12.93 | 9.68 | 13.64 | 12.80 | 8.02 | 13.64 | 1.01 | 1.21 |
| | **doitgen** | 2.42 | 1.23 | 8.39 | 2.11 | 0.50 | 8.39 | 1.15 | 2.46 |

**Table 4: Raw data for Figure 7. In addition to the mean, minimum, and maximum performances across the input space, this table shows which the selected version and its corresponding speedup. For Threads, version 3 of 2d-conf has the best mean performance of 10.28x over the entire set of optimized variants. However, by using the best version for each input in the space, there would be a gain of performance of 6%. Another observation is none of the eight versions is the best for all of the codes.**




**(a) Based on 8 Thread versions**

**(b) Based on 40 Accelerator versions**

**Figure 7: Comparing the performance of the "ideal" adaptive kernel with the performance of the "average best" kernel. The upper charts depicts the performance, in Gflops on a logscale. For each codes there is two bars: the adaptive kernel and the kernel which performs the best in average over the input space. The error-bars show the worst and best performance over the input space. The values are given in column (A) and (B) of Table 4, respectively. The lower charts presents the potential speedups when using the adaptive kernels. They show both mean and worst case speedups corresponding to the last column of Table 4.**

kernel that for any input uses the best optimized code version for the specific input being used. The "average best" kernel is the version of the kernel which has the highest mean performance over the whole input space. However, having the highest mean performance does not mean that the version performs the best for all inputs.

The results in Figure 7 and Table 4 show that adaptive kernels only improves the mean speedup (over the whole input space) by a few percent and as little as 1% for **syr2k**. **2d-conv** gets the most improvements: 6% for the Threads extension and 48% for Accelerator extension.

However, adaptive kernels have a clear advantage when it comes to worst case performance. The worst case performance is the lowest performance achieved for an input over the entire input space.. In this case, adaptive kernels are useful improving the worst case from 1.01 to 1.31 for threads 1.2x to 8x for accelerators. This is due to the high variability of accelerator versions when presented with various inputs.

## 6 FUTURE WORK

Faced with a large optimization space, finding the most suitable version is a tedious task, especially with the possibility of varying inputs. Building adaptive kernels requires knowledge of how the different versions perform across many inputs. In our future work, we will look into the construction of adaptive kernels without evaluating the whole search space. First, we will evaluate classic search algorithms to look for optimal solutions for several inputs. Then, we will investigate the utilization of performance prediction. Specifically, we want to leverage the expressive characterization the compiler IR, to automatically induce predictive models that use the IR as input.

## 7 RELATED WORK

This work is a generalization of our experimentations with OpenACC [1]. In [18] we presented how a static tiling of loops could be used generate OpenCL kernels for OpenACC. This paper presents a much more flexible technique.

In [20, 21] loop tiling is shown as a technique to generate parallel workloads. The classic approach to tiling transforms one loop into outer and inner tiles. When multiple loops are tiled, all outer tiles and inner tiles are grouped. Computations are distributed by assigning each iteration of the outer tiles to one executor. In our technique, each loops can be divided into many tiles. Tiles are not ordered from the outer to the inner tile, instead the order is set by the user. This approach enables more control over the resulting data access pattern by mixing tiling and interchanges.

HOMP [9] is an early implementation of the OpenMP Accelerator Model using the ROSE Compiler [13]. In OpenMP [5], loops are divided into chunks (or blocks) and each chunk is executed "atomically" by one thread. In HOMP, the workload is distributed in the same fashion preventing usage of the multidimensional grids and blocks of CUDA.

An implementation of OpenACC targeting CUDA is presented in [17]. In this work, each loop has at least one counterpart in the generated kernel. The bounds and stride of the generated loop depends on the index of the kernel. Finally the iteration variable is reconstructed for each iteration of this loop. Unfortunately, this publication does not provide details on this process, preventing a better comparaison.

Other implementations of OpenACC [14, 15] directly map loops to one dimension of the CUDA grid/block.

OpenACC 2.0 introduces a tile clause that can be applied to loops and which is supported by the PGI Compiler [19]. However, this tile clause is different from our approach. OpenACC tile clause has a similar effect to the tile directive of the HMPP Codelet Generator Directives [3]. In both compilers, the loop to be tiled is separated between an outer-tile and an inner-tile. When multiple loops are tiled they are reorganized accordingly, all outer-tiles then all inner-tiles. Finally, in the PGI Compiler, other clauses applied to the loop are moved to either of the generated loops. *gang* clauses move to the outer-tile, while *worker* and *vector* clauses are applied to the inner-tile.

## 8 CONCLUSION

We presented TileK, a directive-based compiler that uses tiling to distribute computation on parallel architectures. This paper provides the necessary information to implement TileK's tiling approach of kernel generation. We also introduced a syntax that enables the use of directives to describe complex tiling configurations, including interchange and loop distribution.

Using TileK, we were able to compile annotated C codes to target either multicore CPUs or GPUs. We evaluated many versions of four classic computational codes on both Intel CPUs and NVIDIA GPUs for a variety of different inputs.

Our results show that the best version of a code for one input can perform poorly on other inputs.

The work was funded by award B617435 for the Lawrence Livermore National Laboratory[1].

## REFERENCES

[1] OpenACC: Directives for Accelerators. http://www.openacc-standard.org/. (????).

[2] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. 2006. Using Machine Learning to Focus Iterative Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '06)*. IEEE Computer Society, Washington, DC, USA, 295–305. DOI: http://dx.doi.org/10.1109/CGO.2006.37

[3] CAPS-e. (????). https://www.olcf.ornl.gov/wp-content/uploads/2012/02/HMPPWorkbench-3.0_HMPPCG_Directives_ReferenceManual.pdf.

[4] Yang Chen, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Liang Peng, Olivier Temam, and Chengyong Wu. 2010. Evaluating Iterative Optimization Across 1000 Datasets. *SIGPLAN Not.* 45, 6 (June 2010), 448–459. DOI: http://dx.doi.org/10.1145/1809028.1806647

[5] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE* 5, 1 (1998), 46–55.

[6] Frederica Darema. 2001. The SPMD Model: Past, Present and Future. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Yiannis Cotronis and Jack Dongarra (Eds.). Lecture Notes in Computer Science, Vol. 2131. Springer Berlin Heidelberg, 1–1. DOI: http://dx.doi.org/10.1007/3-540-45417-9_1

[7] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, and K. Yelick. 2005. Self-Adapting Linear Algebra Algorithms and Software. *Proc. IEEE* 93, 2 (Feb 2005), 293–312. DOI: http://dx.doi.org/10.1109/JPROC.2004.840848

[8] R. Dolbeau, F. Bodin, and G. C. de Verdire. 2013. One OpenCL to rule them all?. In *Multi-/Many-core Computing Systems (MuCoCoS), 2013 IEEE 6th International Workshop on*. 1–6. DOI: http://dx.doi.org/10.1109/MuCoCoS.2013.6633603

[9] Chunhua Liao, Yonghong Yan, Bronis R de Supinski, Daniel J Quinlan, and Barbara Chapman. 2013. Early Experiences With The OpenMP Accelerator Model. In *OpenMP in the Era of Low Power Devices and Accelerators*. Springer, 84–98.

[10] Yixun Liu, E. Z. Zhang, and X. Shen. 2009. A cross-input adaptive framework for GPU program optimizations. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. 1–10. DOI: http://dx.doi.org/10.1109/IPDPS.2009.5160988

[11] Patrick McCormick and others. Programming Models. https://asc.llnl.gov/content/assets/docs/exascale-pmWG.pdf. (????).

[12] Aaftab Munshi. 2008. OpenCL. *Parallel Computing on the GPU and CPU, SIGGRAPH* (2008).

[13] Daniel Quinlan and others. ROSE Compiler Infrastructure. http://www.rosecompiler.org. (????).

[14] Ruyman Reyes, Ivan Lopez-Rodriguez, Juan J. Fumero, and Francisco de Sande. 2012. accULL: An OpenACC Implementation with CUDA and OpenCL Support. (2012).

[15] Open source UH Compiler. (????). http://web.cs.uh.edu/~openuh/.

[16] Erich Strohmaier. 2013. Highlights of the 42nd TOP500 List. SC13 BoF. (2013).

[17] Akihiro Tabuchi, Masahiro Nakao, and Mitsuhisa Sato. 2014. A Source-to-Source OpenACC Compiler for CUDA. In *Euro-Par 2013: Parallel Processing Workshops*, Dieter an Mey, Michael Alexander, Paolo Bientinesi, Mario Cannataro, Carsten Clauss, Alexandru Costan, Gabor Kecskemeti, Christine Morin, Laura Ricci, Julio Sahuquillo, Martin Schulz, Vittorio Scarano, StephenL. Scott, and Josef Weidendorfer (Eds.). Lecture Notes in Computer Science, Vol. 8374. Springer Berlin Heidelberg, 178–187. DOI: http://dx.doi.org/10.1007/978-3-642-54420-0_18

[18] Tristan Vanderbruggen and John Cavazos. 2014. Generating OpenCL C kernels from OpenACC. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*. ACM. http://dx.doi.org/10.1145/2664666.2664675

[19] M. Wolfe. (????). http://www.openacc.org/sites/default/files/PGI-Wolfe-GTC-Notes_0.pdf.

[20] M. Wolfe. 1989. More Iteration Space Tiling. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing (Supercomputing '89)*. ACM, New York, NY, USA, 655–664. DOI: http://dx.doi.org/10.1145/76263.76337

[21] Jingling Xue. 2000. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, Norwell, MA, USA.

[1] LLNL IM release number is: LLNL-CONF-718002