HiStencils 2014

HiStencils 2014

Proceedings of the

1st International Workshop on High-Performance Stencil Computations

Vienna, Austria
January 21, 2014

in conjunction with HiPEAC 2014

Workshop organizers and editors:

Armin Größlinger
Harald Köstler

This booklet aggregates the various papers accompanying the presentations at HiStencils 2014. HiStencils does not ask for copyright transfer in any form. Authors retain all rights on their works according to applicable laws, including rights to publish their work elsewhere with or without modification.

Acknowledgements

The organizers are grateful to the members of the program committee for their reviews and discussions during the paper selection process for HiStencils 2014. We are also grateful to the authors who submitted to HiStencils, Rochus Schmid for his keynote speech, Bradley Kuszmaul for his invited talk, and all participants and attendees of HiStencils 2014 in Vienna. We also thank HiPEAC for providing the infrastructure for the HiStencils workshop.

We are grateful to the priority programme 1648 "Software for Exascale Computing" of the German Research Foundation (DFG) for sponsoring HiStencils 2014.

Contents

Marcel Köster, Roland Leißa, Sebastian Hack, Richard Membarth, Philipp Slusallek
Platform-Specific Optimization and Mapping of Stencil Codes through Refinement 1

Alexander Grebhahn, Norbert Siegmund, Sven Apel, Sebastian Kuckuk, Christian Schmitt, Harald Köstler
Optimizing Performance of Stencil Code with SPL Conqueror 7

Stefan Breuer, Michel Steuwer, Sergei Gorlatch
Extending the SkelCL Skeleton Library for Stencil Computations on Multi-GPU Systems 15

Takashi Shimokawabe, Takayuki Aoki, Naoyuki Onodera
A High-productivity Framework for Multi-GPU computation of Mesh-based applications . 23

Yuan Tang, Rezaul Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, Charles E. Leiserson
Pochoir: A Stencil Compiler 31

Pacôme Eberhart, Issam Said, Pierre Fortin, Henri Calandra
Hybrid strategy for stencil computations on the APU 43

Lukasz Szustak, Krzysztof Rojek, Roman Wyrzykowski, Pawel Gepner
Toward efficient distribution of MPDATA stencil computation on Intel MIC architecture . 51

Michael Bader, Alexander Breuer, Sebastian Rettenberger, Kristof Unterweger, Tobias Weinzierl, Roland Wittmann
Hardware-aware block size tailoring on adaptive spacetree grids for shallow water waves . 57

Tobias Grosser, Sven Verdoolaege, Albert Cohen, P. Sadayappan
The relation between diamond tiling and hexagonal tiling 65

Stefan Kronawitter, Christian Lengauer
Optimization of two Jacobi Smoother Kernels by Domain-Specific Program Transformation 75

Siham Tabik, Alin Murarasu, Felipe Romero
Evaluating the Fission/fusion Transformation of an Iterative Multiple 3d-stencil on GPUs 81

Naoya Maruyama, Takayuki Aoki
Optimizing Stencil Computations for NVIDIA Kepler GPUs 89

Organizers and Program Chairs

Armin Größlinger (University of Passau, DE)
Harald Köstler (Friedrich-Alexander-Universität Erlangen-Nürnberg, DE)

Program Committee

Carlo Bertolli (IBM, US)
Matthias Bolten (Bergische Universität Wuppertal, DE)
Mike Clark (NVIDIA, US)
Francisco Gaspar (Universidad de Zaragoza, ES)
Dominik Göddeke (TU Dortmund, DE)
Frank Hannig (Friedrich-Alexander-Universität Erlangen-Nürnberg, DE)
Paul Kelly (Imperial College London, UK)
Bradley C. Kuszmaul (Massachusetts Institute of Technology, US)
Hatem Ltaief (KAUST, SA)
István Reguly (University of Oxford, GB)
Olaf Schenk (Università della Svizzera Italiana, CH)
Jan Treibig (Friedrich-Alexander-Universität Erlangen-Nürnberg, DE)

Additional Reviewers

Ahmad Abdelfattah
Gheorghe-Teodor Bercea
Sebastian Kuckuk
Fabio Luporini
Tareq Malas
Oliver Reiche
Christian Schmitt

Platform-Specific Optimization and Mapping of Stencil Codes through Refinement

Marcel Köster, Roland Leißa, and Sebastian Hack
Compiler Design Lab, Saarland University
Intel Visual Computing Institute
{koester,leissa,hack}@cdl.uni-saarland.de

Richard Membarth and Philipp Slusallek
Computer Graphics Lab, Saarland University
Intel Visual Computing Institute
German Research Center for Artificial Intelligence
{membarth,slusallek}@cg.uni-saarland.de

ABSTRACT

A straightforward implementation of an algorithm in a general-purpose programming language does usually not deliver peak performance: compilers often fail to automatically tune the code for certain hardware peculiarities like memory hierarchy or vector execution units. Manually tuning the code is firstly error-prone as well as time-consuming and secondly taints the code by exposing those peculiarities to the implementation. A popular method to circumvent these problems is to implement the algorithm in a Domain-Specific Language (DSL). A DSL compiler can then automatically tune the code for the target platform.

In this paper we show how to embed a DSL for stencil codes in another language. In contrast to prior approaches we only use a single language for this task. Furthermore, we offer explicit control over code refinement in the language itself which is used to specialize stencils for particular scenarios. Our first results show that our specialized programs achieve competitive performance compared to hand-tuned CUDA programs.

1. INTRODUCTION

Many scientific codes, including stencil codes, require careful tuning to run efficiently on modern computing systems. Specific hardware features like vector execution units require architecture-aware transformations. Moreover, special-case variants of codes often boost the performance. Usually, compilers for general-purpose languages fail to perform the desired transformations automatically for various reasons: First, many transformations are not compatible with the semantics of languages like C++ or Fortran. Second, the hardware models the compiler uses to steer its optimizations, are far too simple. Lastly, the static analysis problems that arise when justifying such transformations are too hard in general.

Therefore, programmers often optimize their code manually, use meta programming techniques to automate the manual tuning or create a compiler for a DSL. A prominent example for meta programming is the C++ template language that is evaluated at compile time and produces a program in the C++ core language. A DSL compiler has the advantage of custom code transformations and code generation. However, implementing a compiler is a cumbersome, time-consuming endeavor. Hence, many programmers embed a DSL in a host language via staging. In DSL staging, a program in a *host language* A is used to construct another program in another language B. A compiler written in A then compiles the B program.

Both approaches have significant limitations concerning the productivity of the programmer: Very often, languages with meta programming capabilities and DSL staging frameworks involve more than one language. One language is usually evaluated at compile time while the other is evaluated during the actual runtime of the program. This requires the programmer to decide which part of the program runs in which stage before he starts implementing. For example, compare the implementation of a simple function, say the factorial, in the C++ template language and the core language. Another significant disadvantage of the two-languages approach is that the type systems of the two languages need to cooperate which is often only rudimentarily supported or not the case at all (C++'s template language is dynamically typed).

Many of the code transformations that are relevant for high-performance codes, and also for stencil codes, can however be expressed by partial evaluation of code written in one single language. Take for example the handling of the boundary condition of a stencil operation, an example we will go through in more detail later. Using a simple conditional the program can test if the stencil overlaps with the border of the field and execute specialized code that handles this situation. However, evaluating this conditional during

HiStencils 2014
First International Workshop on High-Performance Stencil Computations
January 21, 2014, Vienna, Austria
In conjunction with HiPEAC 2014.

http://www.exastencils.org/histencils/2014/

runtime imposes a significant performance overhead. Partially evaluating the program at compile time can specialize the program in way that a particular case of boundary handling is applied to the corresponding region of the field which eliminates unnecessary checks at runtime.

In this paper we investigate the implementation of several stencil codes via DSL embedding in our research-prototype language called *Impala* (Section 2). Impala is an imperative and functional language that borrows heavily from Rust[1]. It extends Rust by a partial evaluator that the programmer can access through Impala's syntax (Section 3). Partial evaluation in Impala is realized in a way that erasing the partial evaluation operators from the program does not change the semantics of the program. This is often not possible in existing languages supporting meta programming. Finally, Impala provides Graphics Processing Unit (GPU) code generation for a subset of programs. In this paper we show that the partially evaluated stencils written in Impala reach a comparable performance to hand-tuned CUDA code (Section 5).

2. STENCIL CODES IN IMPALA

In this section we present our DSL approach for the realization of *stencil codes* in Impala. Consider an implementation which applies a 1D stencil to a data array in Impala written in C-like imperative style:

```
for (let mut i = 0; i<size; ++i) {
  out[i] = 0.25f * arr[i-1] +
           0.50f * arr[i  ] +
           0.25f * arr[i+1];
}
```

The loop iterates over the data array and applies a fixed stencil to each element in the array. The stencil computation is hard-coded in the example for a given kernel. However, this coding style has two problems: First, the stencil is hard-coded and is not generic which means that the code has to be rewritten for a different stencil. Furthermore, an extension to 2D or 3D makes the code even harder to maintain and to understand. Second, the logic iterating over the data array and the computation are tightly coupled which makes it harder to adapt it to different hardware architectures.

To tackle this dilemma, Impala supports code specialization and decoupling of algorithms from schedules. Specialization allows to generate the same optimized code as shown above from a generic stencil function, like `apply_stencil`:

```
fn apply_stencil(arr: [float], stencil: [float],
                 i: int) -> float {
  let mut sum = 0.0f;
  let offset = stencil.size / 2;

  for j in indices(stencil) {
    sum += arr[i + j - offset] * stencil[j];
  }

  return sum;
}
```

The specialization of this function is triggered at a call site which is shown in Section 3.

The desired decoupling of the algorithm and the concrete schedule can be realized by making use of higher-order functions. A custom iteration function `field_indices`, which takes another function (the kernel body) as an argument, can be used for this task in our scenario. The `field_indices` function applies the given body (in form of a lambda function) to all indices of the elements in a passed array. Similar to Rust, Impala offers the possibility to call this function with the syntax of a `for` construct which passes the body of the for loop as function to the `field_indices` functionality:

```
let stencil = [ ... ];

for i in field_indices(arr) {
  out[i] = apply_stencil(arr, stencil, i);
}
```

In our approach the iteration function can be provided in form of a library. That is, the stencil code remains unchanged while the iteration logic can be exchanged by just linking a different target library or just calling a specific library function. The required hardware-specific and cache-aware implementations can then be written separately.

3. CODE REFINEMENT

In this section we describe our refinement approach of algorithms. One of the main reasons for refinement in our setting is to improve performance at run time. An improvement can be achieved by partially evaluating the program at compile time. Especially, a platform-specific mapping of the stencils can be realized with this approach.

3.1 Partial Evaluation

Partial evaluation is a concept for the simplification of program parts which is typically performed at compile time by specialization of compile-time known values. Compilers perform partial evaluation during transformation phases by applying techniques such as constant propagation, loop unrolling, loop peeling, or inlining. However, this is completely transparent to the programmer. That is, programmers cannot control which parts of a program should be partially evaluated with which values. Furthermore, a compiler will usually only apply a transformation, if the compiler can guarantee the termination of the transformation. For this reason, Impala delegates the termination problem to the programmer. He can explicitly trigger partial evaluation by annotating code with `@`. If the annotated code diverges, the compiler will also diverge. On the other hand, partial evaluation goes beyond classic compiler optimizations or unroll-pragmas because the compiler really executes the annotated part of the program. Moreover, the programmer can explicitly forbid partial evaluation via `#`.

In the following example, we trigger partial evaluation of the *apply_stencil* function introduced in the previous section by annotating the call-site of the function:

```
for i in field_indices(arr) {
  out[i] = @apply_stencil(arr, stencil, i);
}
```

During specialization of *apply_stencil*, the compiler tries to evaluate expressions and constructs that are known to be constant and replaces them by the corresponding results of the evaluation.

In our example, the for loop which iterates over the stencil is unrolled and the constants from the stencil are loaded and inserted into the code for each iteration. In order to apply this specialization, the size of the stencil has to be constant. Such an array is called *definite* in Impala and is immutable:

```
let stencil = [ 0.25f, 0.50f, 0.25f ];
```

[1] http://www.rust-lang.org

The arr field, however, is *indefinite* which means that the values of the array are not known at compile time. Hence, accesses to arr remain in the code, but the indices for the accesses are updated: j and offset are replaced by constants in the index computation. The result of partially evaluating *apply_stencil* is the same like hard-coding the stencil computation, as shown before.

3.2 Platform-specific Mapping

In the previous example, we ignored the fact that the arr field is accessed out of bounds at the left and right border when the stencil is applied. One possibility to handle out of bounds memory accesses is to apply boundary handling whenever the field is accessed: For instance, the index can be *clamped* to the last valid entry at the extremes of the field. Therefore, we use two functions: one for the left border (bh_clamp_lower) and one for the right border (bh_clamp_upper):

```
fn bh_clamp_lower(idx: int, lower: int) -> int {
  if (idx < lower) idx = lower;
  return idx;
}

fn bh_clamp_upper(idx: int, upper: int) -> int {
  if (idx >= upper) idx = upper-1;
  return idx;
}

...
for j in indices(stencil) {
  // clamp the index for arr
  let mut idx = i + j - offset;
  idx = bh_clamp_lower(idx, 0);
  idx = bh_clamp_upper(idx, arr.size);
  sum += arr[idx] * stencil[j];
}
```

These checks ensure that the field is not accessed out of range, but at the same time they are applied for each element of the field whether required or not. Applying the check for each memory access comes at the cost of performance when executed on platforms such as GPU accelerators. If we can specialize the code in a way that checks are only executed at the left and right border, there will be no noticeable performance loss. This could be achieved by manually peeling off stencil.size / 2 iterations of the loop iterating over the field and applying boundary handling only for those iterations. However, doing this results in an implementation that cannot be used for different stencils and different scenarios.

Specializing the apply_stencil implementation to consider boundary handling of different field regions, allows us to write reusable code. Hence, we create a function apply_stencil_bh that applies a stencil to a field. It takes two additional functions for boundary handling as arguments. To specialize on the different field regions, we create a loop that iterates over these regions. Applying boundary handling is delegated to the access function that applies boundary handling for the left border only in case of the left field region and for the right border only in case of the right field region:

```
fn access(arr: [float], region: int, i: int, j: int,
          bh_lower: fn(int, int) -> int,
          bh_upper: fn(int, int) -> int
         ) -> float {
    let mut idx = i + j;
    if (region==0) idx = bh_lower(idx, 0);
    if (region==2) idx = bh_upper(idx, arr.size);
    return arr[idx];
}
```

In order to specialize the apply_stencil_bh function, the range for the region and the access function need to be annotated. In addition, we want the stencil computation to be specialized, and thus, also annotate the stencil iteration:

```
fn apply_stencil_bh(arr: [float], stencil: [float],
         bh_lower: fn(int, int) -> int,
         bh_upper: fn(int, int) -> int
         ) -> float {
  let offset = stencil.size / 2;
  // lower bound of regions
  let L = [0, offset, arr.size - offset];
  // upper bound of regions
  let U = [offset, arr.size - offset, arr.size];

  // iterate over field regions
  for region in @range(0,3)
    // iterate over a single field region
    for i in range(L(region), U(region)) {
      let mut sum = 0;
      for j in @indices(stencil)
        // access function applies boundary handling
             depending on the region
        sum += @access(arr, region, i, j+offset,
             bh_lower, bh_upper) * stencil[j];
      arr[i] = sum;
    }
}
```

The apply_stencil_bh function for a single kernel defines an interpreter for stencils which can be specialized through partial evaluation. The synthesized code then performs the actual computation of a specific stencil while the imposed overhead by the interpreter is completely removed according to the first Futamura projection [6, 7].

Consider now the case of partially evaluating the presented interpreter. This yields three distinct loops that iterate over the corresponding regions of the input field. Each loop now only contains region-specific boundary handling checks. Mapping the stencil computation to the GPU results in three distinct compute kernels that operate on the different field regions.

The following code listing shows an application of the previously introduced apply_stencil_bh function and applies it to a specific stencil and boundary handling methods:

```
let stencil = [ 0.25f, 0.50f, 0.25f ];
@apply_stencil_bh(arr, stencil,
                  bh_clamp_lower,
                  bh_clamp_upper)
```

As previously described, this will result in a specialized version of apply_stencil_bh for this scenario:

```
// iterate over the left field region
for i in range(0, 1) {
  let mut sum = 0;
  sum += arr[bh_clamp_lower(i - 1, 0)] * 0.25f;
  sum += arr[bh_clamp_lower(i,     0)] * 0.50f;
  sum += arr[bh_clamp_lower(i + 1, 0)] * 0.25f;
  arr[i] = sum;
}
... // iterate over the center field region
```

Further specialization of the left field region can be used to eliminate the loop iteration and to specialize the boundary-handling calls to bh_clamp_lower. This would evaluate the if-condition of the boundary checks and the following code would emerge:

```
// iterate over the left field region
let mut sum = 0.0f;
sum += arr[0] * 0.25f;
sum += arr[0] * 0.50f;
sum += arr[1] * 0.25f;
arr[0] = sum;
... // iterate over the center field region
```

While we have shown the refinement approach for 1D examples only, the concept can be applied to the multi-dimensional case by introducing a generic `index` type. This type encapsulates the index handling for an arbitrary number of dimensions. Consequently, further changes in the code can be minimized which may typically be required during an adaption to another number of dimensions.

4. APPLICATIONS

In this section we present two example applications, one from the field of image processing and one from the field of scientific computing. We discuss how specialization triggers important optimizations opportunities for the compiler.

Consider a bilateral filter [14] from the field of image processing. This filter smoothes images while preserving the sharp edges of an image. Algorithm 1 shows the pseudo code for the parallel computation of the bilateral filter.

The computation of the filter mainly consists of two components: closeness and similarity. Closeness depends on the distance between pixels and can be precomputed. Similarity depends on the difference of the pixel values and is evaluated on the fly.

Listing 1 shows an implementation in Impala. The precomputed closeness function is stored in a `mask` array. The two inner loops which iterate over the range of the kernel are annotated, to enforce partial evaluation for a given `sigma_d`. This will propagate the constant mask into the computation and will specialize the index calculation. Another possibility in this context would be a mapping of the `mask` to constant memory, in the case of a GPU.

The use of a Jacobi iteration to solve the heat equation can be specialized similarly (Listing 2). We can use the presented `apply_stencil` function to apply the stencil for Jacobi in each step of the iteration. Partial evaluation of this call site propagates the Jacobi stencil into the function. This causes the calculations that would normally be multiplied with zero at run time, to be evaluated to zero at compile time. Hence, these computations will not be performed during execution of the stencil later on.

5. EVALUATION

In this section we outline our compiler framework and show first results on the CPU and GPU.

5.1 Compiler Framework

Our compiler provides back ends for CPUs and CUDA-capable graphics cards from NVIDIA. Programs written in Impala are parsed into an higher-order intermediate representation (IR) that captures functional properties. All transformations and code refinements described in Section 3 are applied on this level of the intermediate representation.

Stencils are not limited to a particular output field but can perform write accesses to different output targets which simplifies the development of stencil codes. Moreover, we allow random read and write accesses to fields as well as different resolutions of the input data.

When generating code for the GPU, for instance, memory allocations on the target device are fully managed by the generated code of our compiler. Required data transfer from the host device to the GPU (and vice versa) is also performed automatically.

For target code generation, we use LLVM [9]. When our

Algorithm 1: Parallel bilateral filter algorithm.

```
1  foreach pixel pix in image arr do in parallel
2      x ← get_index_x(arr, pix)
3      y ← get_index_y(arr, pix)
4      p, k ← 0
5      for yf = −2 · sigma_d to 2 · sigma_d do
6          for xf = −2 · sigma_d to 2 · sigma_d do
7              c ← closeness((x, y), (x + xf, y + yf))
8              s ← similarity(arr[x, y], arr[x + xf, y + yf])
9              k ← k + c · s
10             p ← p + c · s · arr[x + xf, y + yf]
11         end
12     end
13     out[x, y] ← p/k
14 end
```

```
fn main() {
  let width   = 1024;
  let height  = 1024;
  let sigma_d = 3;
  let sigma_r = 5.0f;
  let mut arr = ~array::new(width, height);
  let mut out = ~array::new(width, height);

  let mask = @precompute(...);

  for i: index in field_indices(out) {
    let c_r = 1.0f/(2.0f*sigma_r*sigma_r);
    let mut k = 0.0f;
    let mut p = 0.0f;

    for yf in @range(-2*sigma_d, 2*sigma_d+1) {
      for xf in @range(-2*sigma_d, 2*sigma_d+1) {
        let diff = arr[i + index(xf, yf)] - arr[i];
        let s = exp(-c_r * diff*diff) *
                mask[xf + sigma_d][yf + sigma_d];
        k += s;
        p += s * arr[i + index(xf, yf)];
      }
    }

    out[i] = p/k;
  }
}
```

Listing 1: Bilateral filter description in Impala.

```
fn apply_stencil(arr: [float], stencil: [float],
                 i: index) -> float {
  let mut sum = 0.0f;
  let offset  = stencil.size / 2;
  for j in indices(stencil) {
    sum += arr[i + j - offset] * stencil[j];
  }
  return sum;
}

fn main () {
  let mut arr = ~array::new(width, height);
  let mut out = ~array::new(width, height);
  let a = 0.2f;
  let b = 1.0f - 4.0f * a;
  // stencil for Jacobi
  let stencil = [[0.0f, b, 0.0f],
                 [   b, a,    b],
                 [0.0f, b, 0.0f]];
  while (/* not converged */) {
    for i in field_indices(out) {
      out[i] = @apply_stencil(arr, stencil, i);
    }

    swap(arr, out);
  }
}
```

Listing 2: Jacobi iteration in Impala.

Table 1: Execution times in ms for the Jacobi kernel in Impala on an Intel Core i7 3770 for a field of size 2048×2048.

	Scalar	Vectorized (SSE)
Impala (generic)	13.23	4.39
Impala (specialized)	5.41	2.51

IR is converted to LLVM IR, target-specific mapping is also performed: In particular for GPU execution, index variables are mapped to special hardware registers and the compute kernels are annotated as *kernel*. The resulting IR conforms to the NVVM IR specification[2] and can be compiled to PTX assembly using the CUDA compiler SDK[3]. Code for the CPU can be automatically vectorized with the help of our data-parallel vectorizer [8].

Using the same mapping with similar annotations, LLVM IR can be generated that conforms to Standard Portable Intermediate Representation (SPIR)[4]. SPIR is supported in OpenCL 1.2 via extensions and will allow us to support GPU accelerators from other vendors in the future.

5.2 Performance Estimation

We evaluate the performance of the generated code on two GPU architectures from NVIDIA, using the GTX 580 and GTX 680 GPUs and on an Intel Core i7 3770 CPU. In order to estimate the quality of the generated code we consider one iteration of the Jacobi iteration from Listing 2.

CPU Evaluation

Table 1 shows the evaluation of the Jacobi kernel on the CPU on a single core. The generic version contains an inner loop which iterates over the elements of a given stencil array. Each value of the stencil is loaded from memory for each access (even for elements which are zero). The specialized version is the generated one from Impala after partially evaluating the program. It does not contain an inner loop and no further memory accesses for loading of stencil elements are required. With the help of our data-parallel vectorizer we are able to vectorize the main loops of both programs.

The scalar generic version takes about 2.5× longer than the scalar specialized version. However, making use of vectorization reduces the execution times of both versions significantly. For the specialized version, vectorization improves performance by a factor of 2.1.

GPU Evaluation

Since it is known that stencil codes are usually bandwidth limited, we list the theoretical peak and the achievable memory bandwidth of the GPUs in Table 2. For the Jacobi kernel, we have to load one value from main memory (from `arr`) and to store one value (`out`), if we assume that all neighboring memory accesses for the stencil are in cache. This means for single precision accuracy we have to transfer $4 \cdot 2 = 8$ bytes per element. On the GTX 680 with achievable memory bandwidth of $b = 147.6$ GB/s and for a problem size $N = 2048 \times 2048$ we thus estimate $\frac{N \cdot 8}{b} \cdot 1000 \approx 0.23$ ms for the kernel. This matches quite well to the measured runtime of our hand-tuned CUDA implementation (0.23 ms) for the Jacobi kernel as seen in Table 3.

[2] docs.nvidia.com/cuda/nvvm-ir-spec/index.html
[3] developer.nvidia.com/cuda-llvm-compiler
[4] www.khronos.org/registry/cl/specs/spir_spec-1.2-provisional.pdf

Table 2: Theoretical **peak** and the achievable (**memcpy**) memory bandwidth in GB/s for the GTX 580 and GTX 680 GPUs.

	GTX 580	GTX 680
Peak	192.4 GB/s	192.2 GB/s
Memcpy	161.5 GB/s	147.6 GB/s
Percentage	83.9 %	76.8 %

Table 3: Execution times in ms for the Jacobi kernel on the GTX 580 and GTX 680 for a field of size 2048×2048.

	GTX 580	GTX 680
CUDA (hand-specialized)	0.33	0.35
CUDA (hand-tuned)	0.26	0.23
Impala (specialized)	0.32	0.35

The table shows results for three different variants of the Jacobi kernel:

- a hand-specialized implementation in CUDA where the stencil is hard-coded,
- a hand-tuned refinement of the first implementation with device-specific optimization (custom kernel tiling, unrolling of the iteration space, and usage of texturing hardware), and
- our generated implementation from Impala after specialization.

It can be seen that our generated version from a generic description is as fast as the corresponding hand-specialized implementation in CUDA. The additional performance gains of the hand-tuned implementations stem from unrolling of the global iteration space and usage of texturing memory. In particular, the use of texturing memory is device-specific and only the GTX 680 benefits from this. While we do not support texturing hardware and unrolling of the global iteration space yet, we believe that we will see the same performance improvement once supporting these features.

6. RELATED WORK

Stencil codes are an important algorithm class and consequently a considerably high effort has been spent in the past on tuning stencil codes to target architectures. To simplify this process, specialized libraries [1, 2], auto tuners [3], and DSLs [5, 13, 11] were developed.

We follow the direction of DSLs in our work, but we give the programmer additional opportunities to control the optimization process. Previous DSL approaches such as Liszt [5] and Pochoir [13] focus on providing a simple and concise syntax to express algorithms. However, they offer no control over the applied optimization strategies. An advancement to this is the explicit specification of schedules in Halide [11]: target-specific scheduling strategies can be defined by the programmer. Still it is not possible to trigger code refinement explicitly. Explicit code refinement can be achieved through staging like in Terra [4] and in *Spiral in Scala* [10]. Terra is an extension to Lua. Program parts in Lua can be

evaluated and used to build Terra code during the run of the Lua program. However, this technique makes use of two different languages and type safety of the constructed fragments can only be checked before executing the constructed Terra code. Spiral in Scala uses the concept of lightweight modular staging [12] to annotate types in Scala. Computations which make use of these types, are automatically subject to code refinement. Transformations on these computations, however, are performed implicitly, and thus, the programmer has no further control over the applied transformations.

7. CONCLUSION

In this paper, we have presented a domain-specific language for stencil codes through language embedding in Impala. Unique to our approach is our code refinement concept. By partially evaluating fragments of the input program it is possible to achieve platform-specific optimizations. Compared to traditional compilers, code refinement is triggered explicitly by the programmer through annotations. This allows to achieve traditional optimizations such as constant propagation and loop unrolling. Moreover, we outlined how our concept can be used for domain-specific mapping of stencil codes. As an application of domain-specific mapping, we have shown that we are able to generate specialized code variants for different field regions.

Currently, our research compiler is able to generate target code for execution on GPU accelerators from NVIDIA as well as CPUs. First results show that we can generate GPU code that is as fast as manual implementations for simple stencil codes. Our next steps include evaluation of more complex stencils such as bilateral filtering and 3D stencils. Furthermore, we are currently working on the integration of device-specific optimizations (e. g., support for texturing hardware).

8. ACKNOWLEDGMENTS

This work is partly supported by the Federal Ministry of Education and Research (BMBF), as part of the ECOUSS project as well as by the Intel Visual Computing Institute Saarbrücken.

9. REFERENCES

[1] A. Baker, R. Falgout, T. Kolev, and U. M. Yang. Scaling Hypre's Multigrid Solvers to 100,000 Cores. *High-Performance Scientific Computing*, pages 261–279, 2012.

[2] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part I: Abstract Framework. *Computing*, 82(2):103–119, July 2008.

[3] M. Christen, O. Schenk, and H. Burkhart. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 676–687. IEEE, May 2011.

[4] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: A Multi-Stage Language for High-Performance Computing. In *Proceedings of the 34th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 105–116. ACM, June 2013.

[5] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 9:1–9:12. ACM, Nov. 2011.

[6] Y. Futamura. Partial evaluation of computation process, revisited. *Higher Order Symbol. Comput.*, 1999.

[7] N. D. Jones. Mix ten years later. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*. ACM, 1995.

[8] R. Karrenberg and S. Hack. Whole-Function Vectorization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 141–150. IEEE, Apr. 2011.

[9] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*, pages 75–86. IEEE, Mar. 2004.

[10] G. Ofenbeck, T. Rompf, A. Stojanov, M. Odersky, and M. Püschel. Spiral in Scala: Towards the Systematic Construction of Generators for Performance Libraries. In *Proceedings of the 12th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 125–134. ACM, Oct. 2013.

[11] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *ACM Transactions on Graphics (TOG)*, 31(4):32, July 2012.

[12] T. Rompf and M. Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 127–136. ACM, Oct. 2010.

[13] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The Pochoir Stencil Compiler. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 117–128. ACM, June 2011.

[14] C. Tomasi and R. Manduchi. Bilateral Filtering for Gray and Color Images. pages 839–846. IEEE, Jan. 1998.

Optimizing Performance of Stencil Code with SPL Conqueror

Alexander Grebhahn,
Norbert Siegmund, Sven Apel
University of Passau
Passau, Germany

Sebastian Kuckuk, Christian Schmitt,
Harald Köstler
University of Erlangen-Nuremberg
Erlangen, Germany

ABSTRACT

A standard technique to numerically solve elliptic partial differential equations on structured grids is to discretize them via finite differences and then to apply an efficient geometric multi-grid solver. Unfortunately, finding the optimal choice of multi-grid components and parameters is challenging and platform dependent, especially, in cases where domain knowledge is incomplete. Auto-tuning is a viable alternative, but faces the problem of large configuration spaces and feature interactions. To improve the state of the art, we explore whether recent work on configuration optimization in product lines can be applied to the stencil-code domain. In particular, we extend and use the domain-independent tool SPL Conqueror in a series of experiments to predict the performance-optimal configurations of two geometric multi-grid codes: a program using the HIPA^{cc} framework and an evaluation prototype called HSMGP. For HIPA^{cc}, we can predict the performance of all configurations with an accuracy of 98 %, on average, when measuring 57.5 % of the configurations, and we are able to predict a configuration that is close to the optimal one after measuring only less than 4 % of all configurations. For HSMGP, we can predict the performance with an accuracy of 88 % when measuring 11 % of all configurations.

Keywords

Stencil computations, parameter optimization, auto-tuning, product lines, SPL Conqueror

1. INTRODUCTION

In many areas of computation, large linear or nonlinear systems have to be solved. Geometric multi-grid is one method to solve such systems that have a certain structure, e. g., that arise from the discretization of partial differential equations (PDEs) on structured grids and lead to sparse and symmetric positive definite system matrices. For good introductions and a comprehensive overview on multi-grid methods, we refer to [6, 20]. Algorithmically, most of the multi-grid components are functions that sweep over a computational grid and perform nearest neighbor computations. Mathematically, these computations are linear-algebra operations, such as matrix-vector products. Since the matrices are sparse and contain often similar entries in each row, they can be described by a *stencil*, where one stencils represents one row in the matrix.

A multi-grid algorithm consists of several components and traverses a hierarchy of grids several times until the linear system is solved up to a certain accuracy. As already mentioned, the components and also the parameters, such as how many times a certain component is applied on each level, are highly problem and platform dependent. That is, depending on the hardware and the application scenario, some settings perform faster than others. This gives rise to a considerable number of *configuration options* to customize multi-grid algorithms.

Selecting configuration options (i. e., specifying a *configuration*), to maximize performance is an essential activity in exascale computing [7]. If we use a non-optimal configuration, we may not exploit the full computational power, leading to increased cost and time. However, identifying the performance-optimal configuration is a complex task. Measuring performance of all configurations to determine the fastest one does not scale, because the number of configurations may grow exponentially with the number of configurations options. Alternatively, we can use domain knowledge to identify the fastest configuration. However, domain knowledge may not always be available, and domain experts are rare and expensive.

In product-line engineering, different approaches have been developed to tackle the problem of finding optimal configurations [9, 16, 17]. The idea is to measure some configurations of a program and *predict* the performance of all other configurations (e. g., by using machine-learning techniques).

Our goal is to find out whether existing product-line techniques can be applied for automatic stencil-code optimization. In particular, we use the tool SPL Conqueror by Siegmund et al. [17] to determine the performance influence of configuration options of stencil code implementations. The general idea is to determine the performance influence of individual configuration options first, and to consider interactions between them subsequently. To this end, we use a heuristic with which we learn the performance contribution of numerical parameters with the help of functions learning.

Overall, we make the following contribution: We demonstrate that SPL Conqueror can be applied to the stencil domain to identify an optimal configuration after performing

HiStencils 2014
First International Workshop on High-Performance Stencil Computations
January 21, 2014, Vienna, Austria
In conjunction with HiPEAC 2014.

http://www.exastencils.org/histencils/2014/

a small number of measurements. After performing more measurements, we can predict the performance of all configurations with an accuracy of about 88 % in average.

We demonstrate applicability of our approach with two case studies from the stencil domain. One of the systems investigated is a geometric multi-grid implementation using HIPA^{cc} [14], a framework for generating GPU code. The other system is a prototype of a highly scalable geometric multi-grid solver, developed for testing various algorithms and data structures on high-performance computing systems.

Our experiments show that there are a huge number of interactions between configuration options having considerable influence on performance. However, we can predict a configuration with a good performance even with a small number of measurements.

2. PRELIMINARIES

In this section, we present preliminaries of our approach of finding performance-optimal stencil-code configurations. Since our approach stems from the product-line domain, we introduce respective terminology and provide background information on how to model variability.

2.1 Feature Models

Customization options of variable software systems are often called *features* [11]. Since features may depend on or exclude each other, we use *feature models* to describe a set of valid combinations [2]. In detail, a feature model describes relationships among the features of a configurable system. As examples, we present the feature models of our two subject systems in Figure 1 and Figure 2.

A feature can either be *mandatory* (i. e., required in all configurations where its parent feature is selected), *optional*, or be part of an *Or-group* or an *Xor-group*. If the parent feature of a group is selected, exactly one option of an Xor-group and at least one feature of an Or-group has to be selected. Furthermore, we can define arbitrary propositional formulas to further constrain the variability. For example, one feature may *imply* or *exclude* another one.

Extended Feature Models. Standard feature models can express only the presence or absence of features in a configuration; we refer to these feature as *Boolean features*. Unfortunately, this is insufficient when modeling variability of arbitrary options exposed by stencil code. To overcome this limitation, *extended* or *attributed feature models* have been proposed [3, 4]. These models are extended with possible non-Boolean attributes (*Parameters*), describing properties or special characteristics. To model stencil-code parameters, we use a notation in which attributes have a domain type (i. e., the definition range of the parameter) and a default value. An example of a feature attribute is *Padding* (see Figure 1a). The type of this parameter is "Integer, between 0 and 512" with a step size of 32, and a default value of 0.

2.2 Predicting Performance of Customizable Programs

To predict the performance of configurations of a customizable program, Siegmund et al. propose the SPL Conqueror approach that quantifies the influence of individual features on performance [17, 18]. To this end, they propose several heuristics that assess the individual performance contributions to predict the total performance of a given configuration.

Heuristics. The first heuristic, *feature-wise (FW)*, measures the performance influence of each individual feature. For each feature, two configurations – one with and one without the regarded feature – are measured. The performance difference is interpreted as the performance contribution of the feature in question. With this heuristic, the number of required measurements grows linearly with the number of configuration options. As a consequence, this heuristic can also be applied to huge configuration spaces. A drawback, however, is that the FW-based prediction does not always correspond to the actual performance, because features may influence each other. This impact on performance is called *feature interaction* [17]. For example, effects from changing the used *Texture Memory* can vary based on the current *API* (see Figure 1). There can be even interactions between more than two features. To account for these different orders of interactions (i. e., the number of interacting features), Siegmund et al. propose three further heuristics [17].

The first heuristic considers interactions between all pairs of features (i. e., order of one), called *pair-wise heuristic (PW)*. To measure interactions of a higher order, the *higher-order heuristic (HO)* is used. Lastly, in some customizable programs there are features interacting with many other. To consider the contribution of such *hot-spot features*, we can apply the *hot-spot heuristic (HS)*. The three heuristics considering interactions build on each other and on the FW heuristic. As a consequence, the HS heuristic uses all measurements performed by the FW, PW and HO heuristic.

The heuristics of Siegmund et al. can predict performance contributions of Boolean features only. As a consequence, it is not possible to incorporate parameters. To overcome this limitation, we use a *function-learning heuristic (FL)*. The basic idea is to learn performance-contribution functions for each parameter (non-Boolean attribute). Since each parameter can have a performance influence on different features, we learn multiple functions per parameter. The polynomial of the performance function has to be given by a domain expert or learned by a machine-learning approach. Consequently, a feature model with n features and m parameters requires $n \cdot m$ functions. To learn these *performance-contribution functions*, we sample the type of the parameter (i. e., we select values from the domain) and measure performance of the corresponding configurations. Using a least-squares approach, we determine the contribution of the parameter. When sampling a single parameter, we keep the remaining parameters constant and use their default values.

3. VARIABILITY IN THE MULTI-GRID DOMAIN

In general, a (standard) multi-grid cycle can be defined as follows: The algorithm starts at the finest level. Firstly, high-frequency errors are smoothed with a fixed number of pre-smoothing steps. Afterwards, to get rid of the low-frequency errors, the residual is calculated, restricted to the next coarser level, and then solved for recursively. Next, the solution from this process is propagated onto the current level and used to correct the solution. Lastly, the error is smoothed again by applying a fixed number of post-

smoothing steps.

Obviously, the recursion has to be stopped at some point, at the latest when there is only one unknown left. In this case, direct and exact solving is possible and feasible. In the context of large-scale applications, however, this is not practicable and thus the cycle is stopped before, usually when only a few unknowns are left on each compute unit, and a specialized coarse grid solver is employed. This solver is usually chosen according to the requirements arising from the domain structure, the problem description, and performance considerations.

In multi-grid computations, different types of variabilities arise, which can be grouped according to six criteria.

1. A suitable *hardware platform* and concomitant *external software components* should be chosen. Here, hardware choices include the number of compute units and their type (i. e., CPUs, GPUs, other accelerators or even a combination of them). Concerning software, different compilers and abstraction layers for hardware access, parallelization, and inter-process communication are possible. They include, CUDA, OpenCL, OpenMP, as well as a number of MPI implementations.
2. The algorithmic and numeric components can be adapted. To this end, the *cycle type*, basically, a description of how and when the recursion is performed, can be chosen. The most prominent choices are *V-cycle* and *W-cycle*. Furthermore, different components of the multi-grid cycle can be exchanged, usually only targeting the *smoother* and the *coarse grid solver*. Yet, in general, altering the *restriction* and *propagation operators* is possible as well.
3. Different parameters can be tuned, where these are usually described through numerical values. Examples are the *number of pre-* or *post-smoothing steps* and the *smoothing parameter* ω. Usually, these parameters are limited in the range of values they can take, and further constraints, such as a restriction to integer values, may apply.
4. Optional optimizations can be added on demand. These include basic optimization strategies, such as *padding*, *vectorization*, *(software) prefetching*, *tiling*, and many more. As a detailed overview of possible techniques is beyond the scope of this work, please be referred to [10, 12] for further reading.
5. There is also variability in the problem to be solved. This, however, is usually not controllable from the optimization process, as it is fixed by the applications. Nevertheless, impacts on the (performance) characteristics of the components can be quite prominent and highly diverse. The choices of the PDE to be solved and the applied *boundary conditions* fall into this category, both of which mostly influence the computational complexity. Additionally, different *geometric properties* (i. e., a uniform domain in contrast to a general block-structured domain) are common and influence mostly the communication behavior.
6. Lastly, there are various other changes that could be of interest including different *discretization* schemes (e. g., finite elements instead of the presented finite differences).

Although the range of adjustments is quite broad, their impacts can basically be divided into two groups, namely *convergence* and *performance* impacts. Roughly said, convergence describes how many iterations are required to achieve a satisfyingly accurate solution to the given problem, and performance describes how much time one iteration takes. In our context, most high-level decisions influence both, however often in opposing matter (i. e., the number of iterations decreases while the time per iteration increases or vice versa). In contrast, low-level optimizations typically only influence performance behavior, since the algorithmic layout, and thus convergence, remain unchanged.

4. EXPERIMENTS

The goal of our experiments is to evaluate whether the SPL Conqueror approach is feasible for predicting performance of multi-grid solver configurations. To this end, we define the following research question:

- Q1: What is the prediction accuracy and measurement effort of the heuristics (FW, PW, HO, HS, FL)?
- Q2: What is the performance difference between the optimal configuration and the configuration predicted to perform best.

4.1 Experimental Setup and Procedure

To answer the research questions, we selected two multi-grid solver implementations for different application domains, which will be described in the following sections. Although different evaluation criteria are of interest, we decided for the time to compute the solution. Note that this does not include the time required for compilation, which can easily be in the ranges of minutes for a single configuration and thus may need more time than the computation itself.

Each experiment consists of two phases. In the first phase, we measure a subset of configurations of the subject systems and determine the contribution of features, feature interactions, and parameters; the measured configurations are selected by the heuristic applied (FW, PW, HO, HS, FL). In the second phase, the performance of all possible configurations is predicted, based on the contributions measured before. To determine the prediction accuracy, we measured all configurations of the current system, an approach we call *Brute Force* (BF), as a reference. Then, the average error rate μ of each prediction can be calculated for each heuristic, as follows:

$$\mu = \frac{1}{n} \sum_{0 \leq i < n} \frac{|t_{i,\text{measured}} - t_{i,\text{predicted}}|}{t_{i,\text{measured}}},$$

where n is the number of configurations of the subject system.

Additionally, we determine the performance difference between the performance-optimal configuration and the configuration predicted to perform best.

4.2 HIPA^{cc}

HIPA^{cc} – the Heterogeneous Image Processing Acceleration Framework[1] – generates efficient low-level code from a high-level abstract domain-specific language (DSL) [14]. Coming from the domain of medical image processing, HIPA^{cc} supports several boundary conditions, such as mirroring and clamping. Image filters are described as kernels applied to the target image in a stencil notation and may have read access to multiple source images. Automatic parallelization and generation of CUDA, OpenCL, or Android

[1] `http://hipacc-lang.org`

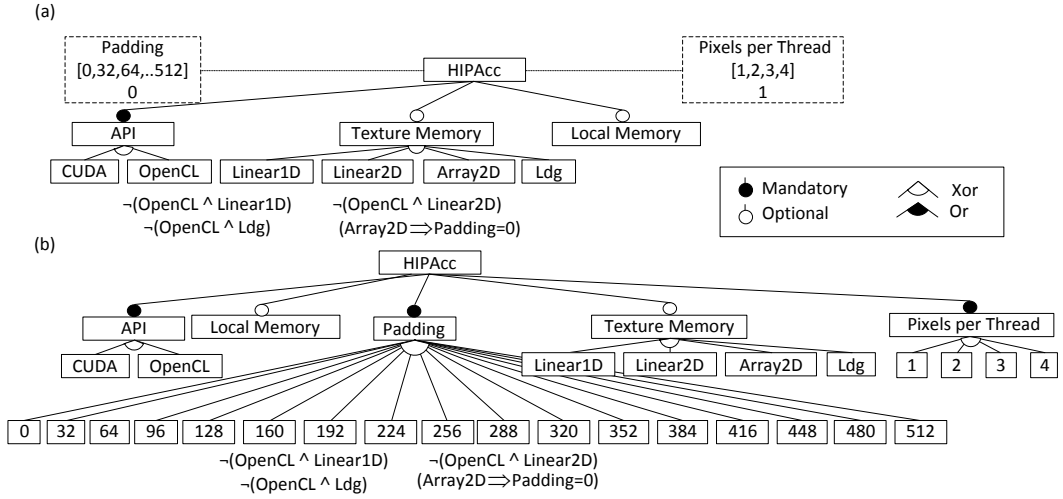


Figure 1: Feature model for HIPA^{cc} experiments, as initially (a) modeled and as modeled (b) with Xor-groups for *Padding* and *Pixels per Thread*.

FilterScript/RenderScript code is performed by a source-to-source-compiler based on Clang. During compilation, HIPA^{cc} optimizes the computational kernels by exploiting domain and hardware knowledge.

The geometric multi-grid code that we use in our experiments solves a finite difference discretization of the Poisson equation. As HIPA^{cc} is a framework for image processing, only a regular, rectangular grid can be employed. The test case uses the Jacobi method for pre- and post-smoothing steps and the standard restriction and prolongation operators. It uses a fixed number of V-cycles and smoothing steps, independent of problem size or convergence rates. For coarse grid solving, the smoother is applied again with a sufficient number of iterations. While there is no variability in the program itself, we consider various optimization switches of HIPA^{cc}. HIPA^{cc} has a built-in list of supported hardware platforms, allowing the target architecture to be selected via a switch at compile time. This switch, however, is only used for the built-in auto-tuning process and to prevent the generation of code unsuitable for the targeted device. Therefore, it has not been varied in our experiments. Since the *API* used for interaction with the device has to be specified, it was modeled as a *Xor-group*. Valid options for the targeted hardware platform are *CUDA* and *OpenCL*. Another *Xor-group* is the memory layout to be used, determined via the parameter *Texture Memory*. Additionally, *Local Memory* is an optional feature toggling the use of another memory type. The integer value *Padding*, modeled as a parameter, may be specified to optimize memory layout. For technical reasons, this parameter can only be increased in steps of 32. Furthermore, the integer value *Pixels per Thread*, denoting how many pixels are calculated per thread, may be varied. We illustrate this variability with the model in Figure 1a. Please note that this feature model is not complete and only resembles the parameters varied for this paper. A more complete description of the HIPA^{cc} parameter space can be found in [15].

Although this model can be used as base for the FL heuristic presented in Section 2.2, the model has to be adjusted for the other heuristics (FW, PW, HO, HS), as they can only hardly handle non-Boolean parameters. To circumvent this problem, we create an *Xor-group* for each of these parameters and introduce a feature for every possible value, resulting in an adapted model, as depicted in Figure 1b.

All measurements of the HIPA^{cc} system are performed on an nVidia K20 card.

4.3 Highly Scalable MG Prototype (HSMGP)

HSMGP is a prototype code for benchmarking HHG (Hierarchical Hybrid Grids) [5, 8] data structures, algorithms, and concepts [13]. It was designed to run on large scale systems such as JuQueen, a Blue Gene/Q system, located at the Jülich Supercomputing Centre, Germany. In its current form, it solves a finite differences discretization of Poisson's equation on a general block-structured domain. Concerning variability, the solver provides different smoothers, where the most relevant ones to us are *Jacobi (Jac)*, *Gauss-Seidel (GS)*, *Red-Black Gauss-Seidel (RBGS)*, and a *Block-Smoother (BS)*. For two of the smoothers, GS and RBGS, additional communication (AC) steps can be performed within the smoother iterations, resulting in a performance overhead but, in turn, also in a possibly improved convergence. Since we want to avoid modeling this as optional feature, we decided for introducing additional modified versions of the original components. Consequently, two new smoothers, *GS with additional communication (GSAC)* and *RBGS with additional communication (RBGSAC)*, are added.

The second customizable algorithmic component is the coarse grid solver. Here, a parallel Conjugate Gradient (CG) and an implementation of an Algebraic Multi-Grid (AMG) provided by the software package HYPRE [2] are available. If AMG is used, HSMGP can perform a redistribution of the coarse grid data onto a smaller number of compute nodes, before starting the solver. Again, we follow the idea presented before and end up with three choices, an *in-place CG (IP_CG)*, an *in-place AMG (IP_AMG)*, and an *AMG with data reduction (RED_AMG)*. As using multiple *smoother* types or *coarse grid solvers* is something usually not occuring in our context, we model these choices as two Xor-

[2] `http://www.llnl.gov/CASC/hypre/`

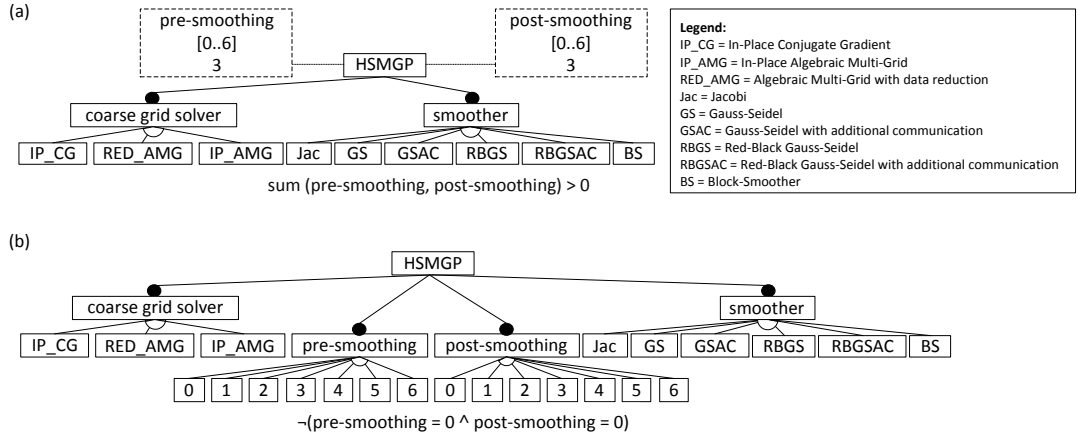


Figure 2: Feature model for HSMGP, as (a) initially modeled and as (b) modeled with Xor-groups for the number of *pre-* and *post-smoothing* steps.

groups.

Furthermore, various parameters can be tuned. In our experiments, we chose the number of *pre-* and *post-smoothing* steps, both of which we limit to integer values between 0 and 6, the default value is set to 3. Additionally, we introduce a constraint that the sum of the two parameters can not be 0, since this would disable smoothing and thus prevent solving. Overall, we end up with the model given in Figure 2a.

For the same reasons given previously, we create an Xor-group for each of the parameters and introduce a feature for every possible value. This results in the feature model depicted in Figure 2b.

Since, the configuration space is quite small, compared to a full model of HSMGP, we are able to measure all configurations (BF), which is necessary to determine accuracy of the SPL Conqueror approach.

For performing our measurements, we chose the JuQueen system at the Jülich Supercomputing Centre, Germany. Although the application scales up to the whole machine (458752 cores) [13], we decided to use only a smaller test case to ensure reproducibility and cost effectiveness. Thus, we setup HSMGP to run with 16384 threads on 4096 cores, solving for roughly $4 \cdot 10^9$ unknowns.

5. RESULTS

The Measurement results for the two subject systems are given in Table 1, we describe them in detail in the remaining section.

5.1 HIPA^{cc}

With respect to the results from our BF measurements, the performance-optimal configuration of HIPA^{cc} use *OpenCL*, a *Padding* of *256*, *4 Pixels per Thread* with *Local Memory* and *Texture Memory* options *disabled*. The time-to-solution of the configuration is 21.25 ms.

With the FW heuristic, we can predict performance for all configurations of HIPA^{cc} with a average error rate of 8.6 % (see Table 1). To achieve this prediction accuracy we perform 26 measurements (3.7 % of all configurations). The absolute time-to-solution difference between the performance-optimal configuration and the configuration predicted to perform best is 0.45 ms, resulting in an error of 2.1 %.

Using the PW heuristic the average error rate decreases to 5.7 % and requires 126 additional measurements, resulting in 152 measurements in total. The absolute performance difference between the optimal configuration and the configuration predicted to perform best decreases to 0.03 ms (0.1 % of the time needed to perform the optimal configuration).

The average error rate decreases to 1.5 % if we apply the HO heuristic. For this heuristic, we need to perform 277 measurements. The performance difference between the optimal configuration and the configuration predicted to perform best is the same as for PW.

If we consider hot-spot features by applying the HS heuristic, we have to perform 407 measurements and reach an average error rate of 1.2 %. We found that the different *Texture Memory* features, the *API* features, *Local Memory*, and different *Pixels per Thread* are hot-spot features. The absolute performance difference between the configuration predicted to perform best and the performance-optimal configuration is only 0.03 ms.

For the FL heuristic, we perform only 52 measurements. The average error rate of this heuristic is about 9.9 %. The absolute difference between the optimal configuration and the configuration predicted to be optimal is 0.02 ms. This is less than 0.1 % of the time to perform the optimal configuration.

5.2 HSMGP

The performance-optimal configuration of HSMGP uses the *IP_AMG* coarse grid solver, the *GS* smoother, one *pre-* and five *post-smoothing* steps with a time-to-solution of 1137.41 ms.

The average error rate of predictions of the FW heuristic is 51.9 % with measuring 22 configurations. The difference between the configuration predicted to perform best and the optimal configuration is 93.8 ms (8.3 % of the time to perform the optimal configuration).

With the PW heuristic, the average error rate decreases to 12.2 %, but requires to measure 192 configurations, 22 % of all valid configurations. The run-time difference between the optimal configuration and the configuration predicted to be optimal is 184 ms.

Program	Heuristic	# M	Time [ms]	Faultrate distribution	$\mu \pm \sigma$	Δ [ms]	δ [%]
HIPA^{cc}	FW	26	698.30		8.6 ± 10.2	0.45	2.1
	PW	152	4 155.11		5.7 ± 9.9	0.03	0.1
	HO	277	9 384.90		1.5 ± 3.7	0.03	0.1
	HS	407	15 491.94		1.2 ± 3.1	0.03	0.1
	FL	52	1 513.92		9.9 ± 15.7	0.02	<0.1
	BF	696	24 580.92	0 20 40 60 80	—	—	—
HSMGP	FW	22	51 773.21		51.9 ± 59.3	93.81	8.3
	PW	192	518 721.48		12.2 ± 14.7	184.80	16.3
	HO	636	2 037 253.25		8.5 ± 17	2474.11	217.5
	HS	864	2 230 326.94		0.2 ± 0.6	5.06	0.4
	FL	88	209 415.50		11.2 ± 10.9	695.31	61.1
	BF	864	2 230 326.94	0 20 40 60 80	—	—	—

Table 1: Experimental results for the two subject systems; FW: feature-wise, PW: pair-wise, HO: higher-order, HS: hot-spot, FL: function learning, BF: brute force, # M: number of measurements required for the heuristic, Time: runtime for all measurements neglecting compilation times, μ: average error rate, σ: standard deviation, Δ: absolute difference between the measured performance of the optimal configuration and the measured performance of the configuration predicted to perform best, δ: percentage share of Δ on measured performance of the optimal configuration.

For the HO heuristic, the number of required measurements increases to 636, which is 73 % of all valid configurations. With this heuristic, the average error rate drops to 8.5 %. Although the average error rate decreases, a non-performance-optimal configuration is predicted to perform best. The absolute difference between this configuration and the optimal configuration is 2474.1 ms (about 217.5 % of the time to perform the optimal configuration).

When applying the HS heuristic, we detect that all available *coarse grid solvers* are hot-spot features. When measuring the interactions of all hot-spot features, we reach an error rate of 0.2 %, but we have to measure *all* configurations for that. The performance difference between the optimal configuration and the configuration predicted to perform best is about 5.1 ms (0.4 % of performance of the optimal configuration).

For the FL heuristic, we observe an average error rate of 11.2 %. We have to measure 88 configurations, representing only 10.2 % of all configurations. With the performance predictions of this heuristic, we predict a configuration to be optimal that has a performance difference of 695.3 ms to the optimal configuration.

6. DISCUSSION

Next, we discuss results for the two subject systems with respect to the research questions Q1 and Q2.

6.1 HIPA^{cc}

Using the FW heuristic, we have to measure only 26 of 696 configurations. With this small set, we are able to predict the performance of a configuration with an accuracy of about 91 % on average. We are also able to predict a configuration to be performance optimal that is only slightly worse than the optimal configuration. But the average error rate suggests the existence of feature interactions. When applying the PW heuristic and considering first-order interactions, we identified several feature interactions, while performing 126 additional measurements. With these measurements, the average error rate decreases to 5.7 %. Additionally, the absolute performance difference between the configuration predicted to be optimal and the performance-optimal configuration is less than 1 % of performance of the optimal configuration. With the HO heuristic, the average error rate decreases to 1.5 % in average. However, the difference between the optimal configuration and the configuration predicted to perform best remains constant.

When additionally considering hot-spot features by using the HS heuristic, we are able to cover all existing interactions. However, the performance-optimal configuration was not predicted to perform best. Yet, the absolute performance difference between the optimal configuration and the configuration predicted to perform best is only 0.02 ms and thus negligible because it is less than 1 % of the performance. Although, we identify more hot-spot interactions than second-order iteration the accuracy improvement is minimal when applying the HS heuristic in contrast to HO heuristic. As a consequence, we state that there are many hot-spot interaction having only a small impact on performance.

Although the FL heuristic performs more measurements than FW heuristic, and also considers interactions between features and parameters, it has the worst prediction accuracy for HIPA^{cc}. This is because interactions between two

or more features and between two or more parameters are not considered.

Overall, the FW heuristic is able to give an impression over the general performance distribution of the configurations. Moreover, an almost optimal configuration was predict to perform best. On the top, even when considering all interactions, we can not identify the optimal configuration but only a configuration that is almost optimal. Additionally, it is possible to predict performance of a random selected configuration with a high accuracy when applying the HO heuristic.

6.2 HSMGP

When considering contributions of individual features only, we have to measure 22 of 864 configurations. The average prediction error is about 51.9 %. As a consequence, there are many feature interactions having an impact on performance. However, the performance loss when using the configuration predicted to perform best is with 8.3 % relatively small.

When using the PW heuristic on HSMGP, 170 additional measurements are needed, compared to the simpler FW heuristic, and the error rate drops substantially, so we conclude that the vast majority of interactions are pair-wise interactions. Measuring 22 % of all configurations of the system to reach an accuracy of 88 %, on average, may be sufficient for some application scenarios. Yet, the accuracy for predicting the optimal configuration decreases.

With the HO heuristic, we considered also interactions between three features, which however required to measure 73.6 % of all configurations. The small improvement of 4 % compared to the PW heuristic does not justify this large number of additional measurements. Worse, a configuration with a bad performance is predicted to perform best. In examining the predictions of all configurations, we find that the configuration predicted to perform second optimal has only an performance penalty of 2.5 % compared to the real optimal configuration.

Similar to the results of HIPA^{cc}, the HS heuristic reaches the best overall prediction accuracy, but for this system, we need to measure all configurations. This strongly indicates that all features interact with each other. In general, this is the worst case for this heuristic. However, even with measuring all configuration the performance-optimal configuration was not predicted to perform best. This is because we use the measured feature contributions to predict performance of a configuration and do not simply return the measured value if the configuration was already measured by the approach.

The FL heuristic requires only 88 measurements (10.2 % of all configurations) to reach a prediction accuracy of 88.8 %, on average. Thus, it produces more accurate predictions with less measurements than the PW heuristic (which was not the case for HIPA^{cc}). However, the performance difference between the optimal configuration and the configuration predicted to perform best is much larger.

As a result, because of the high number of interactions in this system, using the HO and HS heuristics is not beneficial. Again, the FW heuristic performs best to predict the performance optimal configuration when considering the number of performed measurements. However, the PW and FL heuristic can be used to minimize the configuration space.

6.3 Threats to Validity

Internal Validity. Performance measurements are often biased by environmental factors or concurrent system load. Since we use measurements for training and evaluating the predictions' accuracies, these factors threaten internal validity. To reduce this threat, we repeated each measurement multiple times and computed the average (arithmetic mean), which we then use for our evaluation. Although this approach increases the time needed to perform a prediction, it minimizes the measurement noise.

External Validity. As we performed experiments with only two configurable stencil codes, we cannot safely transfer the results to all other stencil programs. However, we examine two systems from different domains that increases external validity slightly. Moreover, we repeated our experiments on JuQueen using a larger test case (16384 instead of 4096 cores), and observed almost identical performance-prediction accuracies.

However, our experiments are just a proof of concept, testing whether applying existing approaches to performance-optimal configuration is feasible in the stencil domain. We are aware, that further experiments and a more refined approach are needed to draw more robust conclusions.

7. RELATED WORK

Optimization and auto-tuning of configurable programs is a widely researched area. There is a number of approaches for performing configuration optimization without relying on domain knowledge. These approaches can be divided in white-box and black-box approaches.

Siegmund et al. propose an white-box approach [19], extending the approach we use for our experiments. They create a simulator of the configurable program [1] to predict the performance contributions of the configuration parameters. Since the simulator contains all variability and all executable code, they are able to predict contributions of multiple parameters within one run.

Gou et al. propose an black-box approach [9] similar to the one of Siegmund et al. They use statistical learning to predicting performance of program configurations. However, in contrast to Siegmund's approach, in which configurations are selected based on heuristics, they perform a random selection of configurations. After measuring an initial set of configurations, they determine which features are responsible for the largest performance deviation. Then, they divide the set of configurations according to the selection of these performance-critical features. When predicting a configuration, they follow the path according to the selected features. Eventually, they use the measured performance of a configuration that has a similar feature selection than the configuration to be predicted. However, this approach is not applicable to non-Boolean parameter values and, since it does not determine feature interactions, it is unclear how this approach handles the huge number of interactions that exist in stencil codes, as in our subject systems.

Datta performed auto-tuning of stencil-code computations for several multicore systems such as IBM Blue Gene/P [7]. With the help of the roofline model [21], Datta predicts performance bounds of stencil codes and the related quality of the optimizations. The optimization consists of two steps: First, he performs an optimization of the parameter ranges

to minimize the configuration space. Second, he performs an iterative greedy search. As a consequence, he optimizes one parameter while the values of other parameters are fixed. The order of parameters are given as domain knowledge.

However, since it is necessary rewriting the code of the target program for some parameters, they need to re-adjust some already optimized parameters. Moreover, he does not consider the number of measurements required for an optimization.

8. CONCLUSION

Stencil codes expose a number of customization options to tune their performance. Without any domain knowledge, it is hard to determine which configuration (i. e., selection of customization options) leads to the best performance. To tackle this problem, we transfer an approach from product-line engineering by Siegmund et al. [17], which predicts performance of software configurations, to the stencil code domain. In a series of experiments, we demonstrated that the approach of Siegmund et al. can be used to predict performance of configurations of stencil codes. For the HIPA^{cc} system, we can predict the performance of all valid configurations with an accuracy of 98 %, on average, when measuring 277 out of 696 configurations. For the HSMGP system, we can predict the performance of all configurations with an accuracy of 88.8 %, when measuring 88 out of 864 configurations. However, our predictions for the optimal configuration have an inaccuracy of, at least, 8.3 %. Yet, it is possible to use our predictions to minimize the configuration space without losing configurations with a good performance.

As future work, we will examine the prediction accuracy of SPL Conqueror for larger configuration spaces. Moreover, we will extend the approach with heuristics considering feature interactions and parameter interactions.

9. ACKNOWLEDGMENTS

We thank Richard Membarth for providing the tested multi-grid implementation in HIPA^{cc} as well as advice on suitable evaluation parameters. This work is supported by the German Research Foundation (DFG), as part of the Priority Programme 1648 “Software for Exascale Computing”, under the contracts AP 206/7-1, TE 163/17-1 and RU 422/15-1. Sven Apel’s work is also supported by the DFG under the contracts AP 206/4-1 and AP 206/6-1.

10. REFERENCES

[1] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of feature interactions using feature-aware verification. In *Proc. ASE*, pages 372–375. IEEE, 2011.

[2] D. Batory. Feature models, grammars, and propositional formulas. In *Proc. SPLC*, pages 7–20. Springer, 2005.

[3] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[4] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In *Proc. CAISE*, pages 491–503. Springer, 2005.

[5] B. Bergen, T. Gradl, F. Hülsemann, and U. Rüde. A massively parallel multigrid method for finite elements. *Computing in Science and Engineering*, 8(6):56–62, 2006.

[6] W. Briggs, V. Henson, and S. McCormick. *A Multigrid Tutorial*. 2nd edition, 2000.

[7] K. Datta. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms*. PhD thesis, EECS Department, University of California, Berkeley, 2009.

[8] B. Gmeiner, H. Köstler, M. Stürmer, and U. Rüde. Parallel multigrid on hierarchical hybrid grids: A performance study on current high performance computing clusters. *Concurrency and Computation: Practice and Experience*, 26(1):217–240, 2014.

[9] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski. Variability-Aware Performance Prediction: A Statistical Learning Approach. In *Proc. ASE*, pages 301–311. IEEE, 2013.

[10] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, Inc., 1st edition, 2010.

[11] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-2, CMU, SEI, 1990.

[12] H. Köstler, M. Stürmer, and T. Pohl. Performance engineering to achieve real-time high dynamic range imaging. *Journal of Real-Time Image Processing*, pages 1–13, 2013.

[13] S. Kuckuk, B. Gmeiner, H. Köstler, and U. Rüde. A generic prototype to benchmark algorithms and data structures for hierarchical hybrid grids. 2013. accepted at ParCo2013.

[14] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert. Generating Device-specific GPU Code for Local Operators in Medical Imaging. In *Proc. IPDPS*, pages 569–581. IEEE, 2012.

[15] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert. Mastering Software Variant Explosion for GPU Accelerators. In *Proc. HeteroPar*, pages 123–132. Springer, 2012.

[16] A. S. Sayyad, T. Menzies, and H. Ammar. On the value of user preferences in search-based software engineering: A case study in software product lines. In *Proc. ICSE*, pages 492–501. IEEE, 2013.

[17] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting performance via automated feature-interaction detection. In *Proc. ICSE*, pages 167–177. IEEE, 2012.

[18] N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Information & Software Technology*, 55(3):491–507, 2013.

[19] N. Siegmund, A. von Rhein, and S. Apel. Family-based performance measurement. In *Proc. GPCE*, pages 95–104. ACM, 2013.

[20] U. Trottenberg, C. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, 2001.

[21] S. Williams, D. Patterson, L. Oliker, J. Shalf, and K. Yelick. The roofline model: A pedagogical tool for auto-tuning kernels on multicore architectures. In *HOT Chips, A Symposium on High Performance Chips*. IEEE, 2008.

Extending the SkelCL Skeleton Library for Stencil Computations on Multi-GPU Systems

Stefan Breuer
stefan.breuer@uni-muenster.de

Michel Steuwer
michel.steuwer@uni-muenster.de

Sergei Gorlatch
gorlatch@uni-muenster.de

Departement of Mathematics and Computer Science
University of Münster, Germany

ABSTRACT

The implementation of stencil computations on modern, massively parallel systems with GPUs and other accelerators currently relies on manually-tuned coding using low-level approaches like OpenCL and CUDA, which makes it a complex, time-consuming, and error-prone task. We describe how stencil computations can be programmed in our SkelCL approach that combines high level of programming abstraction with competitive performance on multi-GPU systems. SkelCL extends the OpenCL standard by three high-level features: 1) pre-implemented parallel patterns (a.k.a. skeletons); 2) container data types for vectors and matrices; 3) automatic data (re)distribution mechanism. We introduce two new SkelCL skeletons which specifically target stencil computations – MapOverlap and Stencil – and we describe their use for particular application examples, discuss their efficient parallel implementation, and report experimental results on manycore systems with multiple GPUs.

Keywords

Stencils, Manycores, GPU, OpenCL, Skeletons, SkelCL

1. INTRODUCTION

Stencil computations play an important role in a number of different application domains including time-intensive scientific simulations, image processing and others. Modern manycore architectures with Graphics Processing Units (GPUs) and other accelerators provide potentially tremendous computing power for challenging applications including stencil computations.

However, the current programming approaches for manycore architectures are low level, the most popular examples being OpenCL [1] and CUDA [2]. These approaches require the programmer to explicitly manage GPU's memory (including memory (de)allocations and data transfers to/from the system's main memory) and explicitly specify parallelism in the computation. This leads to lengthy, low-level, complicated and, thus, error-prone code. For multi-GPU systems, programming with CUDA and OpenCL is even more complex, as both approaches require an explicit implementation of data exchange between the GPUs, as well as disjoint management of each GPU, including low-level pointer arithmetics and offset calculations. When implementing stencil computations, additional challenges arise, like handling out-of-bound memory accesses and achieving high performance by making efficient use of the fast but small local GPU memory.

In this paper, we present our SkelCL [8] approach to high-level, manycore programming, and we describe how it simplifies stencil programming and achieves competitive performance on multi-GPU systems. SkelCL extends the OpenCL standard by three high-level mechanisms:

1) computations are easily expressed using pre-implemented parallel patterns (a.k.a. *skeletons*);

2) memory management is simplified using *container data types* for vectors and matrices;

3) data movement in multi-GPU systems are handled automatically by SkelCL's *(re)distribution mechanism*.

For stencil computations, we extend SkelCL with two specialized skeletons: MapOverlap for simple stencil computations, and Stencil for more complex, in particular iterative, stencil computations.

The paper is organized as follows. In Section 2 we introduce stencil computations and their programming on systems with GPUs. Section 3 presents our SkelCL library for high-level GPU programming. In the next two sections we discuss how SkelCL can be used for stencil computations in single- (Section 4) and multi-GPU systems (Section 5). We evaluate our approach using two real-world stencil computations in Section 6, before we compare our approach with related work and conclude in Section 7.

2. STENCILS USING OPENCL

A *stencil computation* is a computation pattern on a multi-dimensional grid, where each point of the grid is updated (often iteratively) as a function of its neighboring points. Each point of the grid stores a set of application-dependent values. The computation performed to update the values of each point is called the *stencil operation*. A stencil operation updates the value of a point depending on the values of the neighboring points. The points taken into account for a stencil operation are defined by the *stencil shape*.

Let us consider how stencil computations are implemented on manycore systems with GPUs using the state-of-the-art

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
HiStencils 2014 '14 Vienna, Austria
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

```
1  kernel
2    void sobel(global const char* in_img,
3               global char* out_img,
4               int w, int h) {
5    int i = get_global_id(0);
6    int j = get_global_id(1);
7
8    if (i < w && j < h) {
9      char ul = (j-1 > 0 && i-1 > 0)
10         ? in_img[((j-1)*w)+(i-1)] : 0;
11     ...
12     char lr = (j+1 < h && i+1 < w)
13         ? in_img[((j+1)*w)+(i+1)] : 0;
14
15     out_img[j*w+i] =
16         computeSobel(ul, ..., lr);
17   }
18 }
```

Listing 1: Structure of the OpenCL implementation of Sobel edge detection

OpenCL standard. Listing 1 presents the structure of an OpenCL implementation of the Sobel operator on one GPU, a typical stencil computation used in image processing for detecting edges in images. Lines 9–13 show how the direct neighboring elements, e.g., the *upper left* (`ul`) neighbor, are accessed and passed to a function performing the Sobel operation in line 16. Many low-level details have to be considered for a correct implementation, like raw pointer handling, including index computations (e.g., line 10), and explicit out-of-bound accesses handling (e.g., line 9).

The OpenCL version is obviously correct, but not efficient: the fast local GPU memory is not used and the control flow diverges heavily between different work items, which is disadvantageous on current GPU architectures. However, the corresponding optimizations require a deep knowledge of the GPU's architecture and must be programmed and tuned manually and are, therefore, a complicated task for application developers. If the program is to be used on a multi-GPU system then the application developer has to additionally implement and optimize the explicit data distribution across GPUs and the communication between them.

3. THE SKELCL SKELETON LIBRARY

We develop SkelCL [8] – a skeleton library for programming systems with Graphics Processing Units (GPUs). By providing skeletons on container data types, SkelCL alleviates programming of systems with GPUs: parallelism is expressed implicitly, using skeletons, and memory management is performed automatically by the SkelCL implementation built on top of OpenCL. The especially tricky programming of multi-GPU systems is greatly simplified by SkelCL's data distribution mechanism which automatically moves data between multiple GPUs.

Algorithmic Skeletons.

In original OpenCL, computations are expressed as *kernels*, e.g., as in Listing 1, which are executed in a parallel manner on a GPU; the application developer must explicitly specify how many instances of a kernel are launched. In addition, kernels usually take pointers to GPU memory as input and contain program code for reading/writing single data items from/to it. These pointers have to be used carefully, because no boundary checks are performed by OpenCL.

To shield the application developer from these low-level programming issues, SkelCL extends OpenCL by introducing high-level programming patterns, called *algorithmic skeletons* [5]. Formally, a skeleton is a higher-order function that executes one or more user-defined (so-called *customizing*) functions in a pre-defined parallel manner, while hiding the details of parallelism and communication from the user [5].

The current version of SkelCL provides four basic skeletons (*Map*, *Reduce*, *Zip*, and *Scan*) and three more advanced skeletons (*Allpairs*, *MapOverlap*, and *Stencil*). Due to lack of space, we only describe the first two basic skeletons here; the other basic skeletons are described in detail in [8].

The Map skeleton applies a unary function f to each element of an input vector $[v_1, v_2, \ldots, v_n]$, i. e.:

$$map\ f\ [v_1, v_2, \ldots, v_n] = [f(v_1), f(v_2), \ldots, f(v_n)]$$

The Reduce skeleton computes a scalar value from a vector using an associative binary operator $\oplus$, i. e.

$$red\ \oplus\ [v_1, v_2, \ldots, v_n] = v_1 \oplus v_2 \oplus \cdots \oplus v_n$$

In SkelCL, rather than writing low-level kernels, the application developer customizes suitable skeletons by application-specific functions which work on basic data types and, therefore, they are often much simpler than kernels that work with pointers. Skeletons can be executed on both single- and multi-GPU systems; on a multi-GPU system, the calculation specified by a skeleton is performed automatically on all GPUs of the system.

Container Data Types.

SkelCL offers two container classes – vector and matrix – which are transparently accessible by both the CPU and the GPUs. The *vector* abstracts a one-dimensional contiguous memory area while the *matrix* provides an interface to a two-dimensional memory area.

The advantage of the container data types in SkelCL as compared with OpenCL is that data transfers between the memories of the CPU and GPUs are performed implicitly. All computations in SkelCL accept containers as their input and output. Before execution, the SkelCL implementation ensures that all input containers' data is available on all participating GPUs. This may result in implicit (automatic) data transfers from the CPU memory to GPU memory, which in OpenCL would require explicit programming. Similarly, before any data is accessed on the CPU, the implementation of SkelCL ensures that this data on the CPU is up-to-date. This may result in implicit data transfers from the GPU which are performed automatically too. Thus, the container classes free the programmer from low-level operations like memory allocation (on GPU) and data transfers between CPU and GPU.

While all data transfers are performed implicitly by SkelCL we understand that advanced application developers want fine grained control over the data transfers between CPU and GPU. For that purpose SkelCL offers a set of APIs developers can use to explicitly initiate and control the data transfer to and from the GPU.

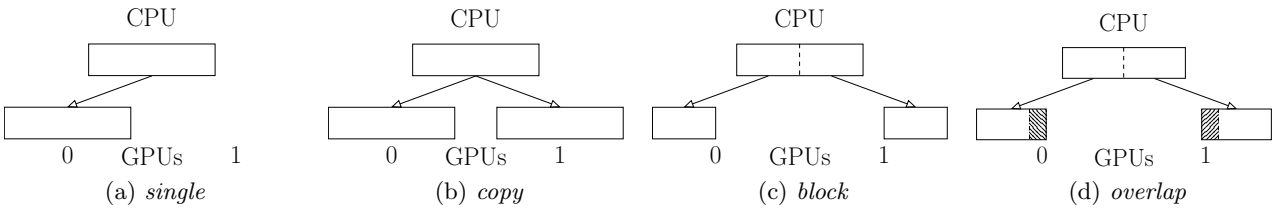


Figure 1: Distributions of a vector in SkelCL.

(Re)Distribution Mechanism.

For multi-GPU systems, SkelCL's parallel container data types (vector and matrix) abstract from the separate memory areas on multiple GPUs, i. e., container's data is accessible by each GPU. To simplify the partitioning of a container on multiple GPUs, SkelCL supports the concept of *distribution* that specifies how a container is distributed among the GPUs. It allows the application developer to abstract from explicitly managing memory ranges which are shared or partitioned across multiple GPUs.

Four kinds of distributions are currently available to the application developer in SkelCL: *single*, *copy*, *block*, and *overlap* (see Fig. 1 for illustration on a system with two GPUs). If set to the *single* distribution (Fig. 1a), container's whole data is stored on a single GPU (the first GPU if not specified otherwise). The *copy* distribution (Fig. 1b) copies container's entire data to each available GPU. With the *block* distribution (Fig. 1c), each GPU stores a contiguous, disjoint block of the container. The *overlap* distribution (Fig. 1d) is used for the MapOverlap and Stencil skeletons: it stores on both GPUs a common block of data from the border between the GPUs.

The application developer can set the distribution of containers explicitly or every skeleton selects a default distribution for its input and output containers otherwise. The distribution of a container can be changed at runtime: this implies data exchanges between multiple GPUs and the CPU, which are performed by the SkelCL implementation implicitly. Implementing such data transfers in standard OpenCL is a cumbersome task: data has to be downloaded to the CPU before it can be uploaded to other GPUs, including the corresponding length and offset calculations; this results in a lot of low-level code which is completely hidden when using SkelCL.

4. NEW SKELETONS FOR STENCILS

The idea of our approach is that while the stencil operation varies for different applications, the overall structure of stencil computations stays the same. Therefore, stencil computations can be implemented as a skeleton which is customized by the application developer with a particular stencil operation and particular stencil shape.

To simplify the development of stencil applications, we introduce two specialized skeletons in SkelCL: *MapOverlap* and *Stencil*. While MapOverlap supports simple stencil computations, the Stencil skeleton provides support for more complex stencil computations with more complex stencil shapes and (possibly) iterative execution.

Listing 2 shows the implementation of the Sobel edge detection using the *MapOverlap* skeleton. The MapOverlap skeleton applies a given function $func$ (defined in lines 2–6) to each element of an input matrix in_{img} while taking the neighboring elements within the range $[-d, +d]$ in each dimension into account. Here, d is the second parameter (line 7) and two additional parameters define how the skeleton handles out-of-bound memory accesses (line 8). A helper function (`get`) is used to easily access the neighboring elements. The indexes are specified relative to the current element, e. g. to access the element on the left the function call `get(in, -1, 0)` is used.

```
1  MapOverlap<char(char)> sobel(
2   "char func(const char* in_img) {
3      char ul = get(in_img, -1, -1);
4      ...
5      char lr = get(in_img, +1, +1);
6      return computeSobel(ul,..., lr);}",
7   1, Padding::NEUTRAL, 0);
8
9  output = sobel(input);
```

Listing 2: Implementation of Sobel edge detection using the MapOverlap skeleton

```
1  Stencil<char(char)> heatSim(
2   "char func(const char* in) {
3      char lt = get(in, -1, -1);
4      char lm = get(in, -1,  0);
5      char lb = get(in, -1, +1);
6      return computeHeat(lt, lm, lb); }",
7   StencilShape(1, 0, 1, 1),
8   Padding::NEUTRAL, 255);
9
10 output = heatSim(100, input);
```

Listing 3: Implementation of heat simulation using the Stencil skeleton

Special handling is necessary when accessing elements out of the boundaries of the matrix, e.g., when the item in the top-left corner of the matrix accesses elements above and left of it. The MapOverlap skeleton can be configured to handle such out-of-bound memory accesses in two possible ways: 1) a specified neutral value is returned; 2) the nearest valid value inside the matrix is returned. In Listing 2, the first option is chosen and 0 is provided as neutral value.

Simple stencil computations with a regular stencil shape can easily be expressed using the MapOverlap skeleton. For more complex stencil computations, e. g. iterative stencils, we introduce the more advanced *Stencil* skeleton.

The MapOverlap Skeleton.

Listing 3 shows the implementation of an iterative stencil application simulating heat transfer. This application simulates heat spreading from one location and flowing throughout a two-dimensional simulation space.

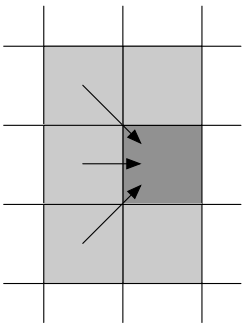


Figure 2: Stencil shape for heat transfer simulation

The application developer specifies the function (line 2–6) describing the computation and, therefore, the stencil shape, as well as the stencil shape's extents (line 7) and the out-of-bound handling (line 8). The stencil shape's extents are specified using four values for each of the directions: up, right, down, left. In the example in Listing 3, the heat flows from left to right, therefore, no accesses to elements to the right are necessary and the stencil space's extents are specified accordingly (note the 0 in line 7 representing the extent to the right). Figure 2 illustrates this situation: the dark gray element is updated by using the values from the left. The specified stencil shape's extent is highlighted in light gray. In our current implementation, the user has to explicitly specify the stencil shape's extents, which is necessary for performing the out-of-bound handling on the GPU. In future work, we plan to automatically infer the stencil shape's extents from the customizing function using source code analysis in order to free the user from specifying this information explicitly.

Many stencil applications apply a stencil multiple times for a fixed number of iterations, or until a certain condition is met. For example, to iterate the heat transfer simulation for one hundred steps, we specify the number of iterations to perform when executing the skeleton (line 10). In the future, we plan to allow the user to specify a custom function which checks a condition to stop the iterations.

The MapOverlap skeleton can be configured to handle out-of-bounds accesses by returning the nearest valid value of the input matrix. Another distinction can be made regarding iterations and sequences of stencil operations: using elements of the **initial**, user-provided input matrix or using elements of the **current** step's input matrix, which already was updated during earlier stencil operations. The Stencil skeleton can be configured to handle out-of-bounds accesses in both ways, thus offering three possible ways, including the neutral value. For each of them, there is an own kernel function, loading appropriate elements into local memory.

The Stencil Skeleton.

Sequence of Stencil Operations.

Many real-world applications perform different stencil operations in a sequence. Let us consider the popular *Canny algorithm* which is used for detecting edges in images. For the sake of simplicity we consider a simplified version, which applies the following stencil operations in a sequence: first, a noise reduction operation is applied, e.g., a Gaussian filter; second, an edge detection operator like the Sobel filter

```
1  Stencil<Pixel(Pixel)> gauss(...);
2  Stencil<Pixel(Pixel)> sobel(...);
3  Stencil<Pixel(Pixel)> nms(...);
4  Stencil<Pixel(Pixel)> threshold(...);
5
6  StencilSequence<Pixel(Pixel)> canny(
7    gauss, sobel, nms, threshold);
8
9  output = canny(1, input);
```

Listing 4: Structure of the Canny algorithm as implemented with a sequence of skeletons.

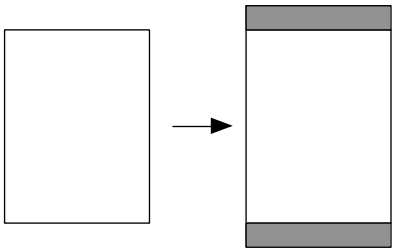


Figure 3: The MapOverlap skeleton prepares a matrix by copying data on the top and bottom.

is applied; third, the so-called non-maximum suppression is performed, where all pixels in the image are colored black except pixels being a local maximum; finally, a threshold operation is applied to produce the final result. A more complex version of the algorithm performs the edge tracking by hysteresis, as an additional step. This results in detecting some weaker edges, but even without this additional step the algorithm usually achieves good results.

In SkelCL, each single step of the Canny algorithm can be expressed using the Stencil skeleton. The last step, threshold operation, does not need access to neighboring elements, as the user threshold function only checks the value of the current pixel. Therefore, this step can be expressed using SkelCL's simpler Map skeleton. The Stencil skeleton's implementation automatically uses the simpler Map skeleton's implementation when the user specifies a stencil shape which extents are 0 in all directions.

To implement the Canny algorithm in SkelCL, the single steps can be combined as shown in Listing 4. The individual steps are defined in lines 1–4 and then combined to a sequence of stencils in line 6 and 7. During execution (line 9), the stencil operation are performed in the order which is specified when creating the *StencilSequence* object.

Implementation.

In order to achieve high performance, our implementations of both the MapOverlap and the Stencil skeleton use the GPU's fast local memory. Both implementations perform the same basic steps on the GPU: first, the data is loaded from the global memory into the local memory; then, the user-defined function is called for every data element by passing a pointer to the element's location in the local memory; finally, the result of the user-defined function is copied back into the global memory.

Although both implementations perform the same basic steps, different strategies are implemented for loading the data from the global into the local memory.

The MapOverlap skeleton prepares the input matrix on the CPU before uploading it to the GPU: padding elements are appended; they are used to avoid out-of-bounds memory accesses to the top and bottom of the input matrix, as shown in Figure 3. This slightly enlarges the input matrix, but it reduces branching on the GPU due to avoiding some out-of-bound checks. In SkelCL a matrix is stored row-wise in memory on the CPU and GPU, therefore, it would be complex and costly to add padding elements on the left and right of the matrix. To avoid out-of-bound accesses for these regions, the boundary checks are performed on the GPU.

The Stencil skeleton has to use a different strategy in order to enable the usage of different padding modes and stencil shapes when using several Stencil skeletons in a sequence. As an example, consider two stencil shapes in a sequence where the first shape defines a neutral element 0 and the second defines a neutral element 1. This cannot be implemented using MapOverlap's implementation strategy. Therefore, Stencil does not append padding elements on the CPU, but rather manages all out-of-bounds accesses on the GPU, which slightly increases branching.

5. TARGETING MULTI-GPU SYSTEMS

The implicit and automatic support of systems with multiple OpenCL devices is one of the key features of SkelCL. By using distributions, SkelCL completely liberates the user from error-prone and low-level explicit programming of data (re)distributions on multiple GPUs.

The MapOverlap skeleton uses the overlap distribution with *border regions* in which the elements calculated by a neighboring device are located. When it comes to iteratively executing a skeleton, data has to be transferred among devices between iteration steps, in order to ensure that data for the next iteration step is up-to-date. As the MapOverlap skeleton does not explicitly support iterations, its implementation is not able to exchange data between devices besides a full down- and upload of the matrix. In addition, data exchange has to be performed after each iteration. We can enlarge the number of elements in the border regions and perform multiple iteration steps on each device before exchanging data. However, this introduces redundant computations, such that a trade-off between data exchange and redundant computations has to be found.

For the Stencil skeleton, the user can specify the number of iterations between *device synchronisations*, where all border regions are updated with elements from the corresponding inner border regions of the neighboring device. The border regions are sized by default in such a way that the specified number of iterations can be performed without leading to incorrect results. However, there may be cases in which a different number of iterations between device synchronizations may result in better performance. Therefore, Stencil offers the user the possibility to specify that number. Please note that the implementation of the Stencil skeleton only exchanges elements from the border region and does not perform a full down- and upload of the matrix, as the MapOverlap skeleton does.

Figure 4 shows the device synchronization. Only the appropriate elements in the inner border region are downloaded and stored as `std::vector`s in a `std::vector`. Within the outer vector, the inner vectors are swapped pair-wise on the host, so that the inner border regions can be uploaded in order to replace the out-of-date border regions.

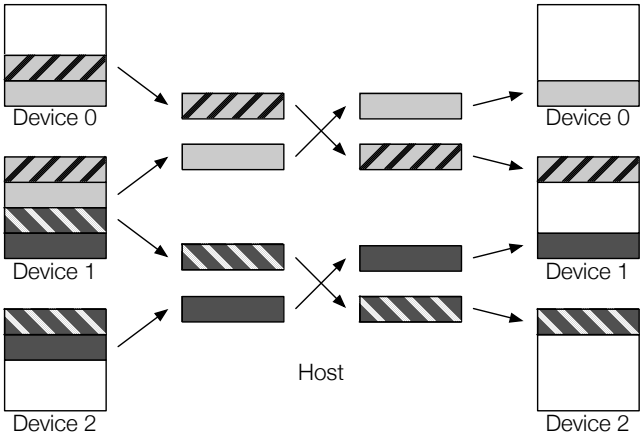


Figure 4: Device synchronization for three devices. Equally patterned and colored chunks represent the border regions and their matching inner border region. After the download of the appropriate inner border regions, they are swapped pair-wise on the host. Then the inner border regions are uploaded in order to replace the out-of-date border regions.

For the first iteration after a device synchronization, there are as many work-items on the GPU active as there are total elements on the device. As the first and last rows of the border regions become invalid within an iteration, the corresponding work-items become inactive in the following iteration step. This is done by using an offset and by reducing the number of total work-items when launching the OpenCL kernel. The Stencil's four kernel functions (one for each out-of-bounds handling mode and one for the adapted Map skeleton) can be used for both single- and multi-GPU systems.

6. EVALUATION

For evaluating our two skeleton implementations, we study two stencil applications: 1) the Gaussian blur, a popular noise reduction technique in image processing, and 2) the Canny algorithm for detecting edges in images. These two applications have different characteristics. The Gaussian blur applies a single stencil computation, possibly iterated multiple times, for reducing the noise in images. The Canny edge detection algorithm consists of a sequence of stencil operations which are applied once to obtain the final result. For each application, we compare the performance of our MapOverlap and Stencil skeletons using an input image of size 4096×3072.

The measurements run on a Tesla S1070 computing system with 4 GPUs, each providing 4 GB of memory, accessing this memory with 102 GB/s, and 240 compute units per GPU running at 1.44 GHz. The GPUs are connected to the host system with a quad-core CPU (Intel E5520, 2.26 GHz) and 12 GB of main memory. 200 runs were performed for each configuration and the average was calculated; to reduce measuring inaccuracy, the best and worst 5% measurements were not considered.

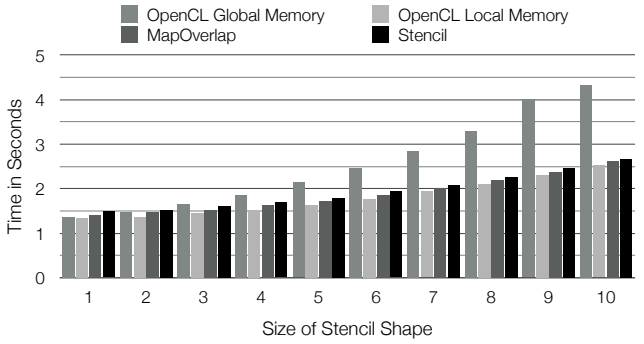


Figure 5: Runtime of the Gaussian blur using a naïve OpenCL implementation with global memory, an OpenCL version using local memory and SkelCL's MapOverlap and Stencil skeletons.

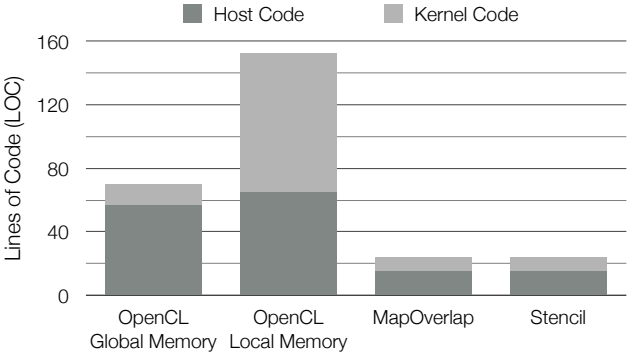


Figure 6: Lines of code (LOCs) of the Gaussian blur using a naïve OpenCL implementation with global memory, an optimized OpenCL version using local memory and SkelCL's MapOverlap and Stencil skeletons.

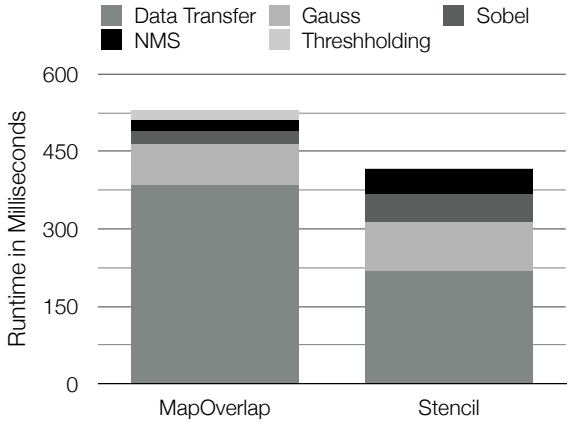


Figure 7: Runtime of the Canny algorithm implemented with the MapOverlap and Stencil skeletons.

Gaussian Blur with a single iteration.

Figure 5 shows the total runtime of the Gaussian blur using: 1) a naïve OpenCL implementation using global memory, 2) an optimized OpenCL version using local memory, and 3) the MapOverlap, and 4) the Stencil skeletons for different sizes of stencil shape, correspondingly. We observe that on larger stencil shape sizes, MapOverlap and Stencil outperform the naïve OpenCL implementation by 65% and 62%, respectively. The optimized OpenCL version, which copies all necessary elements into local memory prior to calculation, is 5% faster than MapOverlap and 10% faster than Stencil for small stencil shapes. When increasing the stencil shape size, this disadvantage is reduced to 3% for MapOverlap and 5% for Stencil with stencil shape's extent of 10 in each direction.

As expected, the Stencil skeleton's implementation is slower for small stencil shapes than the MapOverlap skeleton's, up to 32% slower for an stencil shape size of 1. However, this disadvantage is reduced to 4.2% for an stencil shape size of 5 and becoming negligible for bigger stencil shape sizes. Due to the increased branching in Stencil's kernel function, one might expect a worse runtime for the Stencil skeleton. As the ratio of copying into local memory decreases in comparison to the number of calculations when enlarging the stencil shape's extents, the Stencil skeleton kernel function's runtime converges to the MapOverlap skeleton's. The Stencil skeleton's disadvantage is also due to its ability to manage multiple stencil shapes and explicitly support the use of iterations. While both features are not used in this use case, they incur some overhead for the Stencil skeleton as compared to the MapOverlap skeleton for simple stencil computations.

Figure 6 shows the program sizes (in lines of code) for the four implementations. The application developer needs 57 lines of OpenCL host code and 13 LOCs for performing a Gaussian blur with global memory. When using local memory, some more arguments are passed to the kernel, increasing the host-LOCs to 65, while the LOCs for the kernel function, which copies all necessary elements for a work-group's calculation into local memory, requires 88 LOCs with explicit out-of-bounds handling and complex index calculations. MapOverlap and Stencil are similar to use and both require only 15 LOCs host code and 9 LOCs kernel code to perform a Gaussian blur. The support for multi-GPU systems is implicitly given when using SkelCL's skeletons, such that the kernel remains the same as for one-GPU systems. This is an important advantage of SkelCL over the OpenCL implementations of the Gaussian blur which are single-GPU only, and they require additional LOCs when fitting to multi-GPU environments.

The implementations using MapOverlap and Stencil are only $5 - 10\%$ slower than an optimized OpenCL implementation of the Gaussian blur while being much shorter than the OpenCL version.

Canny edge detection.

Figure 7 shows the absolute runtime of the Canny algorithm (Listing 4). As the MapOverlap skeleton appends padding elements to the matrix, the matrix has to be downloaded, resized and uploaded again to the GPU between each step of the sequence. This additional work to an increased time for data transfers. The Gaussian blur with a stencil shape extent of 2, as well as the Sobel filter and the non-maximum suppression with a stencil shape of 1, are 2.1 to 2.2 times faster when using MapOverlap. However, the threshold operation, which is expressed as the Map skeleton in the Stencil sequence, is 6.8 times faster than MapOverlap's

threshold operation. Overall, when performing sequences of stencil operations, the Stencil skeleton reduces the number of copy operations and therefore leads to a better overall performance. When performing the Canny algorithm, Stencil outperforms MapOverlap by 21%.

Gaussian Blur using multiple GPUs.

Figure 8 shows the speedup achieved on the Gaussian blur using `Stencil` on up to four devices. The higher the computational complexity for increasing size of stencil shape, the better the overhead is hidden, leading to a maximum speedup of 1.90 for two devices, 2.66 for three devices, and 3.34 for four devices.

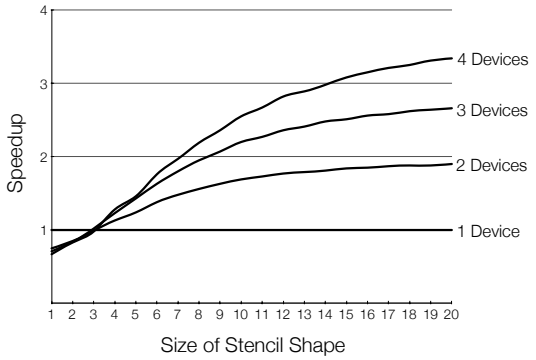


Figure 8: Speedup on up to four GPUs.

7. CONCLUSION AND RELATED WORK

In the paper, we describe how stencil computations are programmed in our SkelCL approach that combines high level of programming abstraction with competitive performance on multi-GPU systems. We introduce two SkelCL skeletons for stencil computations – MapOverlap and Stencil – and we discuss their efficient parallel implementation, and report experimental results. We demonstrate that when executing a single stencil shape once, the MapOverlap skeleton should be used; in all other cases, the Stencil skeleton is the better choice regarding both user comfort and performance. Both skeletons meet SkelCL's requirements of offering high levels of programming abstraction together with a competitive performance on multiple devices, and yield much shorter and simpler codes than when using OpenCL.

For future work, we want to study the applicability of our approach for stencil applications from different fields, like physical simulations, or solving partial differential equations. To enable more applications to use SkelCL, we want to introduce a three-dimensional data structure and adopt the existing skeletons to it. Furthermore, we are interested in building a performance model for our skeletons to better understanding and predicting the runtime of skeleton based applications on different target architectures.

Several approaches aiming at simplifying GPU programming exist. *SkePU* [4] provides a vector class similar to our `Vector` class, but unlike SkelCL it does not support different kinds of data distribution on multi-GPU systems. SkelCL provides a more flexible memory management than SkePU, as data transfers can be expressed by changing data distribution settings. *Thrust* [6] provides two vector types similar to the vector type of the C++ Standard Template Library. While these types refer to vectors stored in CPU or GPU memory, respectively, SkelCL's vector data type provides a unified abstraction for CPU and GPU memory. Thrust also contains data-parallel implementations of higher-order functions, similiar to SkelCL's skeletons. SkelCL adopts several of Thrust's ideas, but it is not limited to CUDA-capable GPUs and supports multiple GPUs. Both SkePU and Thrust provide no explicit support for stencil computations.

Several projects focus on stencil computations on GPUs. PATUS [3] is a code generation and tuning framework for stencil computations. It can generate optimized code for multicore processors and a single GPU. PARTANS [7] is a code generation and autotuning framework for stencil computations on multiple GPUs. It automatically distributes and optimizes stencil computations on multiple GPUs, by searching for optimal parameters for a given hardware architecture. These specialized domain-specific approaches can only be applied to stencil computations, whereas SkelCL is a general-purpose approach.

8. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments, as well as NVIDIA for their generous hardware donation used in our experiments.

9. REFERENCES

[1] *The OpenCL Specification*, November 2012. Version 1.2.

[2] *NVIDIA CUDA C Programming Guide*, July 2013. Version 5.5.

[3] M. Christen, O. Schenk, and H. Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687, 2011.

[4] J. Enmyren and C. Kessler. SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems. In *Proceedings 4th Int. Workshop on High-Level Parallel Programming and Applications (HLPP-2010)*, 2010.

[5] S. Gorlatch and M. Cole. Parallel skeletons. In *Encyclopedia of Parallel Computing*, pages 1417–1422. 2011.

[6] J. Hoberock and N. Bell. Thrust: A Parallel Template Library, 2009. Version 1.1.

[7] T. Lutz, C. Fensch, and M. Cole. PARTANS: An autotuning framework for stencil computation on multi-GPU systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):59, 2013.

[8] M. Steuwer and S. Gorlatch. SkelCL: Enhancing OpenCL for high-level programming of multi-GPU systems. In M. Victor, editor, *Parallel Computing Technologies - 12th International Conference (PaCT 2013)*, volume 7979 of *Lecture Notes in Computer Science*, pages 258–272. Springer Berlin Heidelberg, 2013.

A High-productivity Framework for Multi-GPU computation of Mesh-based applications

Takashi Shimokawabe
Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku,
Tokyo, Japan

Takayuki Aoki
Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku,
Tokyo, Japan

Naoyuki Onodera
Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku,
Tokyo, Japan

shimokawabe@sim.gsic.titech.ac.jp

ABSTRACT

The paper proposes a high-productivity framework for multi-GPU computation of mesh-based applications. In order to achieve high performance on these applications, we have to introduce complicated optimized techniques for GPU computing, which requires relatively-high cost of implementation. Our framework automatically translates user-written functions that update a grid point and generates both GPU and CPU code. In order to execute user code on multiple GPUs, the framework parallelizes this code by using MPI and OpenMP. The framework also provides C++ classes to write GPU-GPU communication effectively. The programmers write user code just in the C++ language and can develop program code optimized for GPU supercomputers without introducing complicated optimizations for GPU computation and GPU-GPU communication. As an experiment evaluation, we have implemented multi-GPU computation of a diffusion equation by using this framework and achieved good weak scaling results. By using peer-to-peer access between GPUs in this framework, the framework-based diffusion computation using two NVIDIA Tesla K20X GPUs is 1.4 times faster than manual implementation code. We also show computational results of the Rayleigh-Taylor instability obtained by 3D compressible flow computation written by this framework.

Categories and Subject Descriptors

D.3.3 [**Software**]: Language Constructs and Features—*Frameworks*

General Terms

Languages

Keywords

Application Framework, Multi-GPU Computing, High Performance Computing

HiStencils 2014
First International Workshop on High-Performance Stencil Computations
January 21, 2014, Vienna, Austria
In conjunction with HiPEAC 2014.

http://www.exastencils.org/histencils/2014/

1. INTRODUCTION

Mesh-based physical simulations are important applications in the field of high-performance computing. Because of extremely memory-bottlenecked computation, these simulations are difficult problems on conventional supercomputers. Exploiting GPU for general-purpose computing has emerged as a high-performance computing technique to accelerate mesh-based physical applications [9, 8, 10, 6, 4, 3].

Although various applications are accelerated by GPUs, programming on different types of devices by using low level platform-specific programming languages such as CUDA [7] that is specific to NVIDIA GPUs forces the programmer to learn multiple distinctive programming models especially to achieve high performance as expected. To solve this problem and improve programmer productivity, various types of high-level programming models were proposed. Mint was proposed as a high-level framework specialized for stencil computations on CUDA-enabled GPUs [11]. As another example, Physis was proposed as a high-level programming framework based on a domain-specific language (DSL) for large-scale GPU computation specialized to stencil computations with regular multidimensional Cartesian grids [5]. PATUS was proposed as a code generation and auto-tuning DSL for stencil computations targeted at multi- and many core processors [1, 2].

Recently, supercomputers such as Titan at the Oak Ridge National Laboratory and TSUBAME 2.5 at the Tokyo Institute of Technology are equipped with large number of GPUs. Multi-GPU computation of mesh-based applications has the potential to achieve high performance. Introducing optimizations allow us to achieve optimal performance. However, programming for large-scale parallel computing on GPU-rich supercomputers is a more difficult task than programming for a single device since introducing optimizations for communication along with single-GPU optimizations is required.

In this paper, high-productivity framework for multi-GPU computing of mesh-based applications is proposed. Since part of existing codes and external existing libraries are often used for development of real applications, the framework should be designed to have the capability of the cooperation with these existing codes. In addition, in order to enhance extensibility and portability of user codes with the framework, they should be written in the standard language without using the non-standard programming model and language extension. Thus, unlike previous research, the proposed framework can be used in the user code developed in the C++ languages. The framework itself is written in the

C++ language with CUDA. The framework provides C++ classes that support the programmer to write stencil functions that update a grid point, execute these functions and describe efficient GPU-GPU communication. By using these classes, the programmer can write user code just in the C++ language and develop program code optimized for multiple GPU systems including GPU-rich supercomputers without introducing complicated optimizations. Since the programmer can write the stencil functions without depending on platform-specific programming languages, the framework is possible to translate these user-written functions to several platforms; the proposed framework currently generates CPU code and GPU code.

2. OVERVIEW OF FRAMEWORK

The proposed framework is designed to provide highly-productive programming environment for stencil applications with explicit time integration running on regular structured grids. The framework updates the physical variables defined on grid points and stored in arrays in user programs. The framework is intended to execute user programs on NVIDIA's GPUs; the C/C++ language and CUDA are used for the implementation of CPU code and GPU code, respectively. The framework also supports multi-GPU computation.

Our major design goals of the framework are described as follows.

- The proposed framework is written in the C++ language and CUDA and can be used in the user code developed in the C++ language. It is important that the user code with the framework can be written in the standard language without using the non-standard programming model and language extension, especially considering the cooperation with external existing libraries. As array data types, the framework exploits arrays of C/C++ language without introducing any unique data types for arrays. These full-compatible data types allow us to call the external library freely in the user code.
- In the proposed framework, multiple GPUs on a same node are handled by a process with inter-node MPI communication. Since multiple GPUs on a same node are handled by a single process instead of several MPI processes, we may use peer-to-peer access between these GPUs, resulting in performance improvement.
- The framework allows us to write multi-GPU code without considering handling multiple GPUs on a single process, which often requires careful programming techniques.
- To perform stencil computations on grids, the programmer only defines C++ functions that update a grid point, which is applied to entire grids by the framework. Our framework automatically translates these functions and generates both GPU and CPU code. The framework allows us to write the user code just in the C++ language and we can develop program code optimized for GPU computing without introducing complicated optimizations.
- The framework provides a single function that supports efficient both inter-node and intra-node GPU-GPU communications. By using this, we do not have to call MPI functions for communications explicitly.

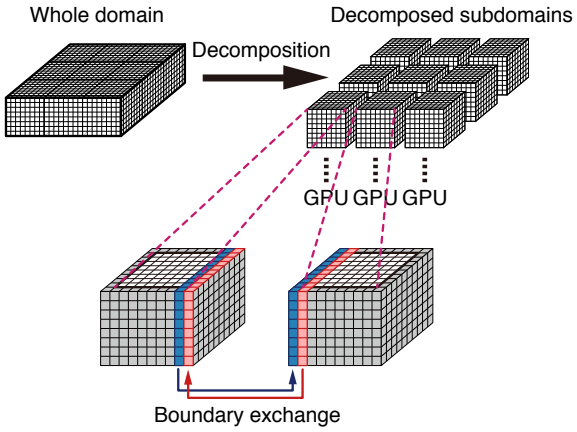


Figure 1: Multi-GPU computing of mesh-based computation.

3. FRAMEWORK IMPLEMENTATION

This section describes the implementation of the proposed framework. First we describe the structure of the entire framework and the execution of the user-written functions that update the physical variables on grids. Then, we describe the implementation of GPU-GPU communication that is required in multi-GPU computing.

3.1 Structure of Framework

The proposed framework supports multiple GPU computing. In the multi-GPU computation of mesh-based applications, the domain decomposition is often used for these parallelization. Figure 1 shows the domain decomposition of computational grid. Since stencil computation that updates to a point of grid needs to access its neighbor points, the data exchanges of boundary regions between subdomains are performed frequently.

Figure 2 shows the multi-GPU computing in this framework using both MPI and OpenMP. In order to utilize peer-to-peer communication between GPUs on the same node, OpenMP and MPI library are used for intra-node and inter-node parallelization, respectively. Each GPU is assigned to a single OpenMP thread. This framework parallelizes not parts of GPU computation in the user code but the entire user code from beginning to end including memory allocation and time integration loop. Thus, the programmer can focus on a single GPU as programming with only MPI (a red frame shown in Figure 2) and can write the user code without considering handling multiple GPUs with both MPI and OpenMP. Note that since computing on CPU using the framework is parallelized by OpenMP in the same way as multi-GPU computation, the computing on CPU needs boundary data exchanges as well as the GPU case even when computation is performed on a single process without using MPI. Note that although overlapping communication with computation is an effective optimization for multi-GPU computation, this framework currently does not introduce it.

3.2 Stencil Computation on Grids

In order to execute stencil computation on grids, the pro-

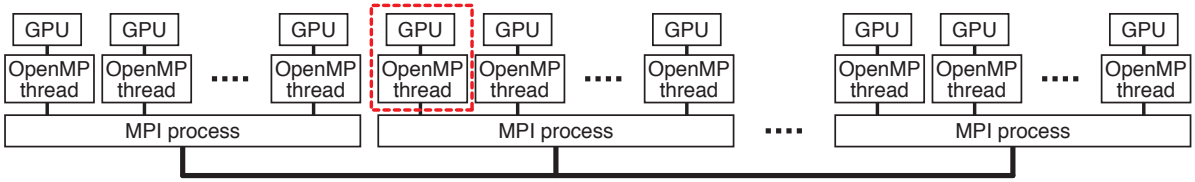


Figure 2: Multi-GPU computing by using both MPI and OpenMP.

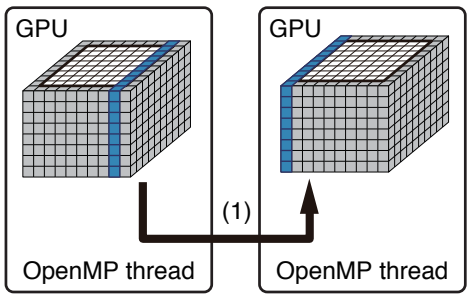


Figure 3: Intra-node GPU-GPU communication by the OpenMP threads.

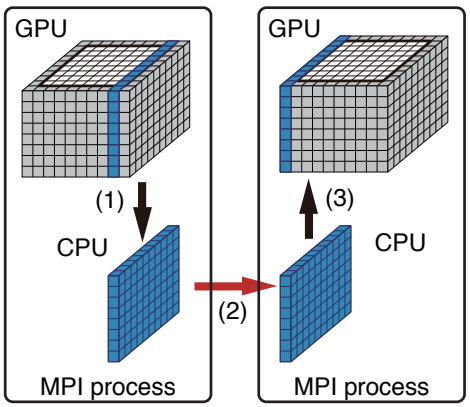


Figure 4: Inter-node GPU-GPU communication by MPI.

grammer must describe functions that update a grid point. The framework provides C++ classes that apply user-written functions to grids. While the user-written functions are executed on grids sequentially for CPU within these classes, these are executed on grids in parallel for GPU using CUDA's global kernel functions. On GPU, read-only data in global memory is loaded through read-only data cache by the framework to improve performance of global loads. To calculate the user-written functions for a given grid size (n_x, n_y, n_z), the kernel functions are configured for execution with $(64, 2, n_z/16)$ threads in each CUDA block. Each thread performs calculations consecutive 16 elements marching in the z direction, resulting in performance improvement.

3.3 GPU-GPU Communication

In this framework, multi-GPU calculations within a same node are performed by an MPI process with several OpenMP threads, each of which is assigned to a single GPU. Since pointers that point to arrays holding the data transferred between GPUs are registered in the memory space that are shared among all threads by using the framework functions prior to data transfer, each thread in the process is able to access memory allocated in others directly. Based on this, the intra-node GPU-GPU communication is performed by just a copy between the memories of two different GPUs using `cudaMemcpy`. When two GPUs support GPUDirect peer-to-peer access, communication between these two GPUs no longer needs to be staged through the host and is therefore faster. Figure 3 illustrates intra-node GPU-GPU communication based on peer-to-peer access.

On the other hand, inter-node GPU-GPU communication is performed by using the MPI library. Figure 4 illustrates this communication. Since GPUs cannot directly access data stored on device memory of other GPUs on other nodes, the host CPUs are used as bridges to exchange boundary data between neighbor GPUs. This process is composed of the following three steps: (1) the data transfer from GPU to CPU using the CUDA runtime library, (2) the data exchange between nodes with the MPI library, and (3) the data transfer back from CPU to GPU with the CUDA runtime library. In order to improve performance of MPI communication and maintain the portability of the framework, master thread executes all MPI communications required by all threads in the same process. Buffers on host memory used for MPI are allocated automatically by the framework.

4. PROGRAMMING MODEL

The framework is written in the C++ language and CUDA. The programmer is required to use the C++ template classes and functions provided by this framework to express stencil-based computations, which are executed with optimizations on GPU. In this section, we describe the programming model of this framework by taking a diffusion computation as an example.

4.1 Parallelizing User Code

The user program must first create a computational domain in each MPI process by using `DomainGroup`, `DomainManager` and `DomainSize`, which are C++ classes provided by the framework.

```
DomainManager manager(px, py, pz);
DomainSize domsize(nx, ny, nz, mgnx, mgny, mgnz);
manager->
  init_domain_size_by_local_domain_size(domsize);
manager->set_thread_assignment(nthreads);
DomainGroup domain_group(rank, &manager);
domain_group.run(main_run);
```

`DomainManager` manages the 3D domain decomposition in the user program. The three parameters of this specify the division numbers of each of the three dimensions. Currently all decomposed subdomains must have the same size, which is specified by `init_domain_size_by_local_domain_size` with

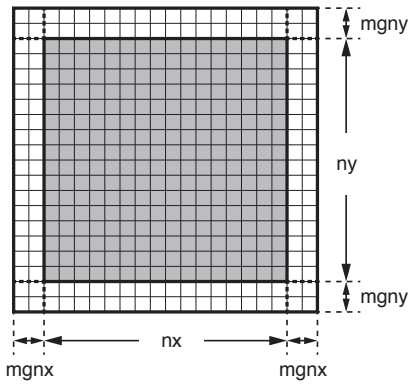


Figure 5: X-Y plane of a computational subdomain that assigned to a GPU.

`DomainSize`. The first three parameters of `DomainSize` are size of each of the three dimensions of the subdomain. The second three parameters are the boundary thickness of each of the three dimensions. Figure 5 shows X-Y plane of a decomposed subdomain. The boundary regions (i.e., the white cells in the figure) are used to store the data sent by neighbor subdomains. Since each subdomain is assigned to an OpenMP thread, it is computed by a single GPU. `DomainGroup` is initialized with the MPI rank `rank` and `DomainManager`, which has `nthreads` parameter that represents the number of OpenMP threads created in each MPI process. `DomainGroup` actually creates `nthreads` of OpenMP threads by using an OpenMP parallel directive. Each of threads executes a user-written function specified by `DomainManager::run`, i.e., `main_run` in the above code.

The user-written function executed in the multiple threads has to run simulation code from beginning to end including memory allocation and time integration loop as follows:

```
int main_run(const Domain &domain) {
  const DomainSize &domsize
      = domain.local_domain_size();
  float *f, *fn;
  cudaMalloc(&f , domsize.ln()*sizeof(float));
  cudaMalloc(&fn, domsize.ln()*sizeof(float));
  initialize_diffusion(domsize, f);
  ...
}
```

The function specified by `DomainManager::run` must receive a `Domain` object as the first parameter. `Domain` holds the information of a computational subdomain assigned to each OpenMP thread, including its size and the connection relation with neighbor subdomains. The size of computational subdomain can be retrieved by `Domain.local_domain_size()`, which may be used for allocation memory and initialization in the user code.

Utilizing multiple GPUs in a single process often requires the programmer to allocate the same number of arrays for a just single physical variable. However, thanks to using OpenMP for intra-node parallelization in this framework, the programmer allocates just one array for each physical variable in the user code in the same way as MPI code without OpenMP (i.e., flat-MPI code) even when multiple GPUs are handled by a single process, which contributes to simplifying the user code.

4.2 Stencil Computation

4.2.1 Writing Stencil Functions

In this framework, stencils must be defined as C++ functors called *stencil functions*. The stencil function for three-dimensional diffusion equation is defined as follows:

```
struct Diffusion3d {
__host__ __device__
void operator()(const ArrayIndex3D &idx,
  float ce, float cw, float cn, float cs,
  float ct, float cb, float cc,
  const float *f, float *fn) {
  fn[idx.ix()] = cc*f[idx.ix()]
  +ce*f[idx.ix<1,0,0>()]+cw*f[idx.ix<-1,0,0>()]
  +cn*f[idx.ix<0,1,0>()]+cs*f[idx.ix<0,-1,0>()]
  +ct*f[idx.ix<0,0,1>()]+cb*f[idx.ix<0,0,-1>()];
}
};
```

Stencil access patterns on three-dimensional grids are described by using `ArrayIndex3D`, which is provided by the framework. Similarly, classes for writing 1D and 2D access patterns are provided. These classes contribute to simplifying writing stencil accesses and enforcing regular neighbor data accesses patterns in stencil functions.

`ArrayIndex3D` holds the size of each dimension of a grid (n_x, n_y, n_z) and index parameters (i, j, k). `ArrayIndex3D` can be used for an array `f` that has $n_x n_y n_z$ elements. When `idx` is an object of `ArrayIndex3D`, `f[idx.ix()]` will return an element on the (i, j, k) point of the grid. `ArrayIndex3D` has C++ template member functions that provide indices of points around the (i, j, k) point of the grid; `idx.ix<1,0,0>()` and `idx.ix<-1,-2,0>()` will, for example, return indices of $(i + 1, j, k)$ and $(i - 1, j - 2, k)$ points, respectively. Using template functions for writing stencil accesses allows us to assume that data dependencies between stencil points can be statically identified and compiler optimizations for index calculations can be expected.

The function parameter of stencil functions must begin with `ArrayIndex3D`, which represent the coordinate of the point where this function is applied. This is followed by any number of additional parameters, including scalar values and pointers of arrays, which typically have $n_x n_y n_z$ elements. The return type of stencil functions must be void. In stencil functions, any dependency among different stencil points must not be assumed, since stencil functions may be executed in parallel with an arbitrary order.

`ArrayIndex3D` returns indices that can be used for arrays which of variables stored sequentially in the order of the x, y, z (ijk-ordering). In stencil applications, other orderings are used to expect optimizations, for example, to increase the cache hit rate and to enable coalesced memory access. In order to change ordering without modifying user-written stencil functions, similar classes to access arrays with other ordering are also provided.

4.2.2 Run Stencil Functions on Grids

In order to apply user-written stencil functions to grids, the framework provides the `Loop3D` class, which is used to

invoke the diffusion equation on the three-dimensional grid as follows:

```
Loop3D loop3d(nx+2*mgnx, mgnx, mgnx,
              ny+2*mgny, mgny, mgny,
              nz+2*mgnz, mgnz, mgnz);
loop3d.run(Diffusion3d(), ce, cw, cn, cs,
           ct, cb, cc, f, fn);
```

`Loop3D` is initialized with parameters that specify a 3D rectangular range where stencil functions are applied. Similarly, `Loop1D` and `Loop2D` for 1D and 2D grids are provided. The first three parameters of the constructor of `Loop3D` specify the range in the x direction. When the first three parameters are n_x, i_0 and i_1, stencil functions must be applied from index i_0 to index $n_x - i_1 - 1$ with assuming the size of grid in the x direction is n_x.

The parameters of `Loop3D::run` must begin with a stencil function defined as a functor, followed by any number of additional parameters that are provided to this functor. We use C++ type inference and call an appropriate functor at `Loop3D::run`. The programmer can define stencil functions as both host and device (i.e., GPU) functions using the qualifiers `__host__` and `__device__` provided by CUDA.

`Loop3D` executes stencil functions on grids sequentially on CPU or in parallel on GPU using CUDA's global functions. `Loop3D` determine whether a pointer given by `Loop3D::run` as a parameter points to host memory or device memory, and call appropriate internal functions within `Loop3D`.

4.3 GPU-GPU Communication

The framework supports computation using multiple GPUs and provides the `BoundaryExchange` class that enables to describe GPU-GPU communication easily. While this class performs inter-node GPU-GPU communication using the MPI library through host memory, it performs peer-to-peer communication between GPUs on a same node when possible. `BoundaryExchange` hides complicated communication procedure required by using MPI along with OpenMP. `BoundaryExchange` is typically used as follows:

```
BoundaryExchange *exchange = domain.exchange();
exchange->append(f);
exchange->transfer();
```

`BoundaryExchange` is initialized by `domain`, which is a `Domain` object, and holds the connection relation with neighbor subdomains and the size of data exchanged with them. Boundary regions of arrays appended by `BoundaryExchange::append` are exchanged when `BoundaryExchange::transfer` is executed. The all pointers that point to arrays registered by `BoundaryExchange::append` are shared among all OpenMP threads. `BoundaryExchange` may allocate buffers on host memory when it uses MPI communication.

4.4 Boundary Condition

In order to simplify writing boundary conditions, the framework provides the `BoundaryCondition` class. This class is typically written as follows:

```
BoundaryCondition *condition
                        = domain.bcondition();
condition->reflected_copy(f);
```

Similar to `BoundaryExchange`, `BoundaryCondition` is initialized by a `Domain` object. After the initialization, a function that applies a boundary condition can be called. This class currently supports the Neumann and Dirichlet boundary conditions. Periodic boundary conditions can be applied to computation by using `BoundaryExchange` since these conditions need data transfer between subdomains. To apply more complicated boundary conditions to computation, the programmer needs to write the stencil functions that apply boundary conditions.

5. EXPERIMENTAL EVALUATION

To evaluate the proposed framework, we use a diffusion equation as a benchmark test. We also apply the proposed framework to a real stencil application; we show computational results of the Rayleigh-Taylor instability obtained by 3D compressible flow computation written by this framework.

5.1 Diffusion computation

A diffusion equation is well used in physical simulations such as computational fluid dynamics and written as follows:

$$\frac{\partial f}{\partial t} = \kappa \nabla^2 f, \tag{1}$$

where f is a physical variable and κ is a diffusion coefficient. This computation needs three-stencil access in each direction; seven neighbor elements of the physical variable are used to update it on a center point of the grid. The boundary regions that have one-element-thick are needed for this computation.

We use the TSUBAME 2.5 supercomputer at the Tokyo Institute of Technology for our evaluation. The TSUBAME 2.5 supercomputer is equipped with 4224 NVIDIA Tesla K20X GPUs. The peak performance of each GPU in single precision is 3.95 TFlops. The on-board device memory (also called global memory in CUDA) provides 250 GB/s peak bandwidth in a Tesla K20X. Each node of TSUBAME 2.5 has three Tesla K20X attached to the PCI Express bus 2.0 $\times$16 (8 GB/s), two QDR InfiniBand and two sockets of the Intel CPU Xeon X5670 (Westmere-EP) 2.93 GHz 6-core.

Figure 6 shows performance of diffusion computation written with the proposed framework and one of manual implementation as reference. The diffusion computation is performed using either one or two GPUs on a same node varying the mesh size from 64^3 to 512^3 in single precision. In the manual implementation using two GPUs, the MPI library is used for communication between them. In the framework case using two GPUs, performance results of diffusion computation with and without peer-to-peer access are shown in this figure. Handling multiple GPUs in a single process is required by peer-to-peer access, which tends to complicate the user code. From this reason, hand-written multi-GPU codes often adopt flat-MPI as their parallelization. Thus we do not use peer-to-peer access in the manual implementation in these evaluations. Since applications using this framework can run on CPU as well as GPU, we also show the performance obtained by using 1 CPU core and 4 CPU cores in this figure as reference.

As shown in this figure, in the one-GPU computation, the performance obtained by using the framework is higher than that by manual implementation; the performance of 194.2 GFlops is achieved by using the framework for a 512^3

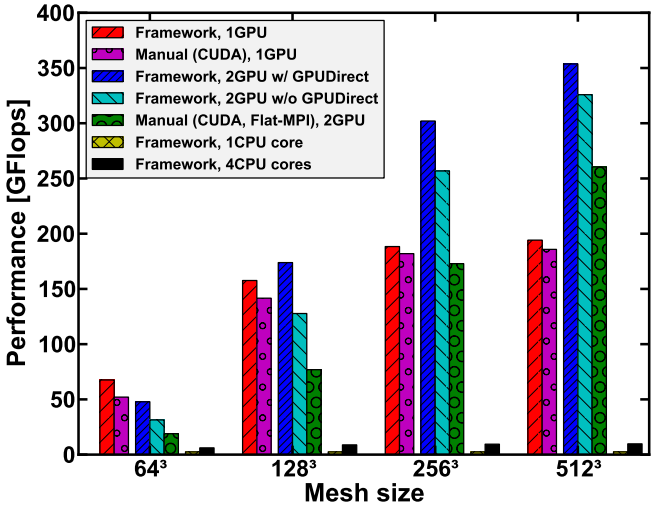


Figure 6: Performance of diffusion computation obtained by the proposed framework and manual implementation.

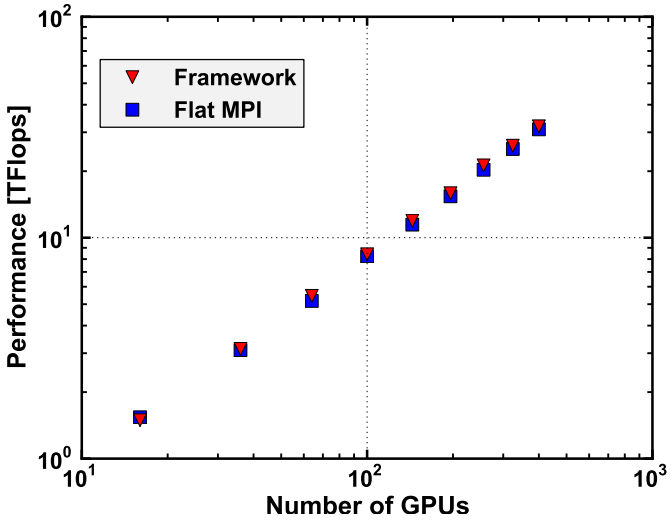


Figure 7: Weak scaling results of diffusion computation on TSUBAME2.5.

mesh, which result is 1.04 times faster than the manual implementation code. Since this computation executes 13 floating-point operations (flop) and needs to read seven elements from the memory and write back one updated value to the memory, which is 32-byte access, to update each point of the grid in one time step, flop per byte ratio of this stencil computation is 0.41. Assuming that this computation is a memory-bound one and utilizes the peak bandwidth of GPU memory (i.e., 250 GB/s), the attainable performance is estimated to be 102.5 GFlops. Since the performance of 194.2 GFlops is achieved by using the framework for a 512^3 mesh, which is more than the estimated value, the stencil function implemented using the framework is likely to be well optimized. In the case of two GPUs, the framework improves the performance drastically compared with the manual implementation code, since GPUs can communicate to each other using peer-to-peer access in the framework instead of MPI communication. The performance of 353.7 GFlops is achieved using the framework, which is 1.4 times faster than the manual implementation code using two GPUs.

Figure 7 shows the weak scaling results of diffusion equation using TSUBAME 2.5. The performance of the manual implementation code and that of the code using the framework are shown in this figure. The calculations are performed in single precision. For the manual implementation, multi-GPU computation is parallelized by MPI without OpenMP (i.e., flat-MPI computation). Both codes use three GPUs per node. Note that while the framework handles three GPUs each node using OpenMP, the manual implementation code handles these three GPUs using three MPI processes. In this weak scaling measurements, each GPU computes a $1024 \times 256 \times 256$ mesh. As shown in this figure, the performance of the code using the framework reaches that by the manual implementation. The weak scaling efficiency of the implementation using the framework is above 85% on 400 GPUs with respect to the 16-GPU performance.

5.2 3D Compressible Flow

We perform 3D compressible flow computation written by this framework and show computational results of the Rayleigh-Taylor instability. To simulate this, we solve 3D Euler equations described as follows:

$$\frac{\partial U}{\partial t} + \frac{\partial E}{\partial x} + \frac{\partial F}{\partial y} + \frac{\partial G}{\partial z} = S, \tag{2}$$

$$U = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho e \end{bmatrix}, \; E = \begin{bmatrix} \rho u \\ \rho u u + p \\ \rho v u \\ \rho w u \\ (\rho e + p) u \end{bmatrix}, \; F = \begin{bmatrix} \rho v \\ \rho u v \\ \rho v v + p \\ \rho w v \\ (\rho e + p) v \end{bmatrix},$$

$$G = \begin{bmatrix} \rho w \\ \rho u w \\ \rho v w \\ \rho w w + p \\ (\rho e + p) w \end{bmatrix}, \; S = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \rho g \\ \rho w g \end{bmatrix},$$

where ρ is density, (u, v, w) are velocity, p is pressure, and e is energy. Here, g is gravitational acceleration. An advection term is solved using three-order upwind scheme with three-order TVD Runge-Kutta method. In this simulation, time integration of five variables ρ, ρu, ρv, ρw, and ρe is solved, which requires 13 neighbor elements of the each variable are used to update them on a center point of the grid in 3D computation. Since the above five variables are used for this simulation, GPU-GPU communications for them can be described as follows:

```
BoundaryExchange *exchange = domain.exchange();
exchange->append(r);
exchange->append(ru);
exchange->append(rv);
exchange->append(rw);
exchange->append(re);
exchange->transfer();
```

Figure 8 shows computational results of the Rayleigh-Taylor instability obtained by 3D compressible flow computation written by this framework. We use 12 GPUs of TSUBAME2.5 for this calculation. As shown in this figure, the real mesh-based applications can be written using the framework.

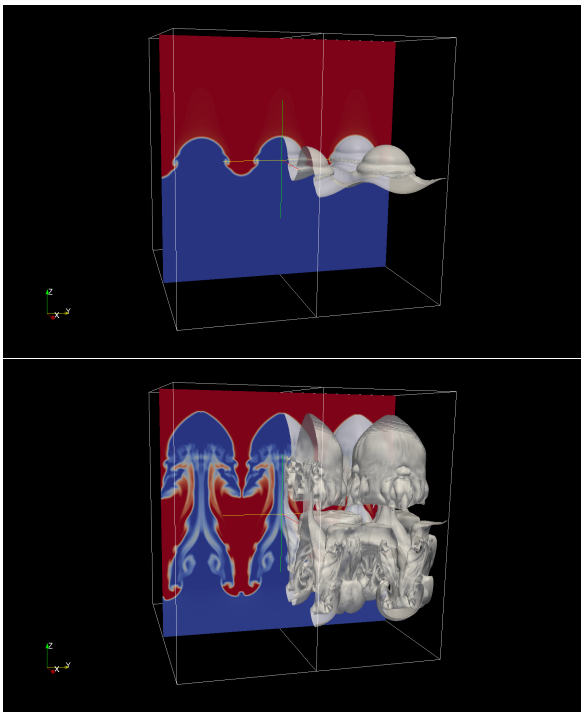


Figure 8: Simulation results of the Rayleigh-Taylor instability.

6. CONCLUSIONS

In order to improve the productivity of writing mesh-based applications optimized for multi-GPU systems and GPU-rich supercomputers, we have proposed a multi-GPU framework for these applications. From the viewpoint of portability of both framework and user code and cooperation with the existing codes, unlike previous research, the proposed framework itself is written in the C++ language with CUDA and can be used in the user code developed in the C++ languages. The programmer can write user code just in the C++ language and develop program code optimized for multiple GPU systems without introducing complicated optimizations explicitly.

In order to perform intra-node GPU-GPU communications effectively, the proposed framework handles multiple GPUs on a same node by OpenMP threads with inter-node parallelization by the MPI library. The programmer can apply MPI with OpenMP parallelization to their user code without considering handling multiple GPUs on a single process, which often requires careful programming techniques. For stencil computation, the programmer writes only the stencil functions that update a grid point using its neighbor points, which are executed over grids by the framework. For multi-GPU computation, the framework provides C++ classes to write GPU-GPU communication effectively. The framework exploits peer-to-peer access between GPUs on a same node to improve performance of intra-node communication. Introducing overlapping communication with computation as an optimization and CUDA-aware MPI to improve the performance will be a subject of our future work.

For our evaluation, we performed a diffusion equation written by the framework on the TSUBAME2.5 supercomputer at the Tokyo Institute of Technology. The performance of the code using the framework reaches that by the manual implementation. In computation using two GPUs on a same node, the user code using the framework is 1.4 times faster than the manual implementation code thanks to utilizing peer-to-peer access. We also showed computational results of the Rayleigh-Taylor instability simulated by 3D compressible flow computation written by this framework.

7. ACKNOWLEDGMENTS

This research was supported in part by KAKENHI, Grant-in-Aid for Young Scientists (B) 25870223, Grant-in-Aid for Scientific Research (B) 23360046 and Grant-in-Aid for Young Scientists (B) 25870226 from the Ministry of Education, Culture, Sports, Science and Technology (MEXT) of Japan, and in part by the Japan Science and Technology Agency (JST) Core Research of Evolutional Science and Technology (CREST) research program “Highly Productive, High Performance Application Frameworks for Post Petascale Computing”. The authors thank the Global Scientific Information and Computing Center, the Tokyo Institute of Technology for use of the resources of the TSUBAME2.5 supercomputer.

8. REFERENCES

[1] M. Christen, O. Schenk, and H. Burkhart. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687, 2011.

[2] M. Christen, O. Schenk, and Y. Cui. Patus for convenient high-performance stencils: Evaluation in earthquake simulations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC ’12, pages 11:1–11:10, Salt Lake City, Utah, 2012. IEEE Computer Society Press.

[3] C. Feichtinger, J. Habich, H. Köstler, G. Hager, U. Rüde, and G. Wellein. A Flexible Patch-Based Lattice Boltzmann Parallelization Approach for Heterogeneous GPU–CPU Clusters. *Parallel Computing*, 37(9):536–549, 2011.

[4] J. C. Linford, J. Michalakes, M. Vachharajani, and A. Sandu. Multi-core acceleration of chemical kinetics for simulation and prediction. In *SC ’09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[5] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’11, pages 11:1–11:12, New York, NY, USA, 2011. ACM.

[6] J. Michalakes and M. Vachharajani. GPU acceleration of numerical weather prediction. In *IPDPS*, pages 1–7. IEEE, 2008.

[7] NVIDIA. CUDA C Programming Guide 5.5. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`, 2013.

[8] T. Shimokawabe, T. Aoki, J. Ishida, K. Kawano, and C. Muroi. 145 TFlops performance on 3990 GPUs of TSUBAME 2.0 supercomputer for an operational weather prediction. *Procedia Computer Science*, 4:1535 – 1544, 2011. Proceedings of the International Conference on Computational Science, ICCS 2011.

[9] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, and S. Matsuoka. An 80-fold speedup, 15.0 TFlops full GPU acceleration of non-hydrostatic weather model ASUCA production code. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, New Orleans, LA, USA, 2010. IEEE Computer Society.

[10] T. Shimokawabe, T. Aoki, T. Takaki, A. Yamanaka, A. Nukada, T. Endo, N. Maruyama, and S. Matsuoka. Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer. In *Proceedings of the 2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 1–11, Seattle, WA, USA, 2011. ACM.

[11] D. Unat, X. Cai, and S. B. Baden. Mint: realizing CUDA performance in 3D stencil methods with annotated C. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 214–224, New York, NY, USA, 2011. ACM.

Pochoir: A Stencil Compiler

Yuan Tang Rezaul Chowdhury Bradley C. Kuszmaul
Chi-Keung Luk Charles E. Leiserson
MIT Computer Science and Artificial Intelligence Laboratory
Cambridge, MA 02139, USA

ABSTRACT

A stencil computation repeatedly updates each point of a d-dimensional grid as a function of itself and its near neighbors. Parallel cache-efficient stencil algorithms based on "trapezoidal decompositions" are known, but most programmers find them difficult to write. The Pochoir stencil compiler allows a programmer to write a simple specification of a stencil in a domain-specific stencil language embedded in C++ which the Pochoir compiler then translates into high-performing Cilk code that employs an efficient parallel cache-oblivious algorithm. Pochoir supports general d-dimensional stencils and handles both periodic and aperiodic boundary conditions in one unified algorithm. The Pochoir system provides a C++ template library that allows the user's stencil specification to be executed directly in C++ without the Pochoir compiler (albeit more slowly), which simplifies user debugging and greatly simplified the implementation of the Pochoir compiler itself. A host of stencil benchmarks run on a modern multicore machine demonstrates that Pochoir outperforms standard parallel-loop implementations, typically running 2–10 times faster. The algorithm behind Pochoir improves on prior cache-efficient algorithms on multidimensional grids by making "hyperspace" cuts, which yield asymptotically more parallelism for the same cache efficiency.

1. INTRODUCTION

Pochoir (pronounced "PO-shwar") is a compiler and runtime system for implementing stencil computations on multicore processors. A ***stencil*** defines the value of a grid point in a d-dimensional spatial grid at time t as a function of neighboring grid points at recent times before t. A ***stencil computation*** [2, 9, 11, 12, 16, 17, 26–28, 33, 34, 36, 40, 41] computes the stencil for each grid point over many time steps.

This work was supported in part by a grant from Intel Corporation and in part by the National Science Foundation under Grants CCF-0937860 and CNS-1017058. An version of this paper appeared in SPAA '11.

Yuan Tang is Assistant Professor of Computer Science at Fudan University in China and a Visiting Scientist at MIT CSAIL. Bradley C. Kuszmaul is Research Scientist at MIT CSAIL and Chief Architect at Tokutek, Inc. Chi-Keung Luk is Senior Staff Engineer at Intel Corporation and a Research Affiliate at MIT CSAIL. Rezaul Chowdhury is Research Scientist at Boston University and Research Affiliate at MIT CSAIL. Charles E. Leiserson is Professor of Computer Science and Engineering at MIT CSAIL.

HiStencils 2014
First International Workshop on High-Performance Stencil Computations
January 21, 2014, Vienna, Austria
In conjunction with HiPEAC 2014.

http://www.exastencils.org/histencils/2014/

Stencil computations are conceptually simple to implement using nested loops, but looping implementations suffer from poor cache performance. Cache-oblivious [15, 38] divide-and-conquer stencil codes [16, 17] are much more efficient, but they are difficult to write, and when parallelism is factored into the mix, most application programmers do not have the programming skills or patience to produce efficient multithreaded codes.

As an example, consider how the 2D ***heat equation*** [13]

$$\frac{\partial u_t(x,y)}{\partial t} = \alpha\left(\frac{\partial^2 u_t(x,y)}{\partial x^2} + \frac{\partial^2 u_t(x,y)}{\partial y^2}\right)$$

on an $X \times Y$ grid, where $u_t(x,y)$ is the heat at a point (x,y) at time t and α is the thermal diffusivity, might be solved using a stencil computation. By discretizing space and time, this partial differential equation can be solved approximately by using the following Jacobi-style update equation:

$$\begin{aligned} u_{t+1}(x,y) &= u_t(x,y) \\ &\quad + \frac{\alpha\Delta t}{\Delta x^2}\left(u_t(x-1,y) + u_t(x+1,y) - 2u_t(x,y)\right) \\ &\quad + \frac{\alpha\Delta t}{\Delta y^2}\left(u_t(x,y-1) + u_t(x,y+1) - 2u_t(x,y)\right) . \end{aligned}$$

One simple parallel program to implement a stencil computation based on this update equation is with a triply nested loop, as shown in Figure 1. The code is invoked as LOOPS$(u;0,T;0,X;0,Y)$ to perform the stencil computation over T time steps. Although the loop indexing the time dimension is serial, the loops indexing the spatial dimensions can be parallelized, although as a practical matter, only the outer loop needs to be parallelized. There is generally no need to store the entire space-time grid, and so the code uses two copies of the spatial grid, swapping their roles on alternate time steps. This code assumes that the boundary conditions are ***periodic***, meaning that the spatial grid wraps around to form a torus, and hence the index calculations for x and y are performed modulo X and Y, respectively.

This loop nest is simple and fairly easy to understand, but its performance may suffer from poor cache locality. Let $\mathcal{M}$ be the number of grid points that fit in cache, and let $\mathcal{B}$ be the number of grid points that fit on a cache line. If the space grid does not fit in cache — that is, $XY \gg \mathcal{M}$ — then this simple computation incurs $\Theta(TXY/\mathcal{B})$ cache misses in the ideal-cache model [15].

Figure 2 shows the pseudocode for a more efficient cache-oblivious algorithm called TRAP, which is the basis of the algorithm used by the Pochoir compiler. We shall explain this algorithm in Section 3. It achieves $\Theta(TXY/\mathcal{B}\sqrt{\mathcal{M}})$ cache misses, assuming that $X \approx Y$ and $T = \Omega(X)$. TRAP easily outperforms LOOPS on large data sets. For example, we ran both algorithms on a 5000×5000 spatial grid iterated for 5000 time steps using

```
LOOPS(u;ta,tb;xa,xb;ya,yb)
1  for t = ta to tb − 1
2      parallel for x = xa to xb − 1
3          for y = ya to ya − 1
4              u((t+1) mod 2,x,y) = u(t mod 2,x,y)
                   + CX · (u(t mod 2,(x−1) mod X,y)
                   + u(t mod 2,(x+1) mod X,y) − 2u(t mod 2,x,y))
                   + CY · (u(t mod 2,x,(y−1) mod Y)
                   + u(t mod 2,x,(y+1) mod Y) − 2u(t mod 2,x,y))
```

Figure 1: A parallel looping implementation of a stencil computation for the 2D heat equation with periodic boundary conditions. The array u keeps two copies of an $X \times Y$ array of grid points, one for time t and one for time $t+1$. The parameters *ta* and *tb* are the beginning and ending time steps, and *xa*, *xb*, *ya*, and *yb* are the coordinates defining the region of the array u on which to perform the stencil computation. The constants $CX = \alpha\Delta t/\Delta x^2$ and $CY = \alpha\Delta t/\Delta y^2$ are precomputed. The call LOOPS$(u;0,T;0,X;0,Y)$ performs the stencil computation over the whole 2D array for T time steps.

the Intel C++ version 12.0.0 compiler with Intel Cilk Plus [23] on a 12-core Intel Core i7 (Nehalem) machine with a private 32-KB L1-data-cache, a private 256-KB L2-cache, and a shared 12-MB L3-cache. The code based on LOOPS ran in 248 seconds, whereas the Pochoir-generated code based on TRAP required about 24 seconds, more than a factor of 10 performance advantage.

Figure 3 shows Pochoir's performance on a wider range of benchmarks, including heat equation (Heat) [13] on a 2D grid, a 2D torus, and a 4D grid; Conway's game of Life (Life) [18]; 3D finite-difference wave equation (Wave) [32]; lattice Boltzmann method (LBM) [30]; RNA secondary structure prediction (RNA) [1, 6]; pairwise sequence alignment (PSA) [19]; longest common subsequence (LCS) [7]; and American put stock option pricing (APOP) [24]. Pochoir achieves a substantial performance improvement over a straightforward loop parallelization for typical stencil applications, such as Heat and Life. Even LBM, which is a complex stencil having many states, achieves good speedup. When Pochoir does not achieve as much speedup over the loop code, it is often due to the spatial grid being too small to yield good parallelism, the innermost loop containing many branch conditionals, or the benchmark containing a high ratio of memory accesses to floating-point operations. For example, RNA's small grid size of 300^2 yields a parallelism of just over 5 for both Pochoir and parallel loops, and its innermost loop contains many branch conditionals. PSA operates over a diamond-shaped domain, and so the application employs many conditional branches in the kernel in order to distinguish interior points from exterior points. These overheads can sometimes significantly mitigate a cache-efficient algorithm's advantage in incurring fewer cache misses.

The Berkeley autotuner [8, 26, 41] focuses on optimizing the performance of stencil kernels by automatically selecting tuning parameters. Their work serves as a good benchmark for the maximum possible speedup one can get on a stencil. K. Datta and S. Williams graciously gave us their code for computing a 7-point stencil and a 27-point stencil on a 258^3 grid with "ghost cells" (see Section 4) using their system. Unfortunately, we were unable to reproduce the reported results from [8] — presumably because there were too many differences in hardware, compilers, and operating system — and thus we are unable to offer a direct side-by-side comparison. Instead, we present in Figure 5 a comparison of our results to their reported results.

We tried to make the operating conditions of the Pochoir tests as similar as possible to the Berkeley environment reported in [8]. We compared Pochoir running 8 worker threads on a 12-core system to the reported numbers for the Berkeley autotuner running 8

```
TRAP(u;ta,tb;xa,xb,dxa,dxb;ya,yb,dya,dyb)
 1  Δt = tb − ta
 2  Δx = max{xb − xa, (xb + dxbΔt) − (xa + dxaΔt)} // Longer x-base
 3  Δy = max{yb − ya, (yb + dybΔt) − (ya + dyaΔt)} // Longer y-base
 4  k = 0 // Try hyperspace cut
 5  if Δx ≥ 2σx Δt
 6      Trisect the zoid with x-cuts
 7      k += 1
 8  if Δy ≥ 2σy Δt
 9      Trisect the zoid with y-cuts
10      k += 1
11  if k > 0
12      Assign dependency levels 0, 1, ..., k to subzoids
13      for i = 0 to k // for each dependency level i
14          parallel for all subzoids
                (ta,tb;xa′,xb′,dxa′,dxb′;ya′,yb′,dya′,dyb′)
                with dependency level i
15              TRAP(ta,tb;xa′,xb′,dxa′,dxb′;ya′,yb′,dya′,dyb′)
16  elseif Δt > 1 // time cut
17      // Recursively walk the lower zoid and then the upper
18      TRAP(ta,ta+Δt/2;xa,xb,dxa,dxb;ya,yb,dya,dyb)
19      TRAP(ta+Δt/2,tb; xa+dxaΔt/2,xb+dxbΔt/2,dxa,dxb;
        ya+dyaΔt/2,yb+dybΔt/2,dya,dyb)
20  else // base case
21      for t = ta to tb − 1
22          for x = xa to xb − 1
23              for y = ya to yb − 1
24                  u((t+1) mod 2,x,y) = u(t mod 2,x,y)
                        + CX · (u(t mod 2,(x−1) mod X,y)
                        + u(t mod 2,(x+1) mod X,y) − 2u(t mod 2,x,y))
                        + CY · (u(t mod 2,x,(y−1) mod Y)
                        + u(t mod 2,x,(y+1) mod Y) − 2u(t mod 2,x,y))
25          xa += dxa
26          xb += dxb
27          ya += dya
28          yb += dyb
```

Figure 2: The Pochoir cache-oblivious algorithm that implements a 2D stencil computation to solve the 2D heat equation using a trapezoidal decomposition with hyperspace cuts. The parameter u is an $X \times Y$ array of grid points. The remaining variables describe the hypertrapezoid, or "zoid," embedded in space-time that is being processed: *ta* and *tb* are the beginning and ending time steps; *xa*, *xb*, *ya*, and *yb* are the coordinates defining the base of the zoid; *dxa*, *dxb*, *dya*, and *dyb* are the slopes (actually inverse slopes) of the sides of the zoid. The values σ_x and σ_y are the slopes of the stencil in the x- and y-dimensions, respectively, which are both 1 for the heat equation.

threads on 8 cores. The comparison may result in a disadvantage to the Berkeley autotuner, because their reported numbers involve only a single time step, whereas the Pochoir code runs for 200 time steps. (It does not make sense to run Pochoir for only 1 time step, since its efficiency is in large measure due to the temporal locality of cache use.) Likewise, the Pochoir figures may exhibit a disadvantage compared with the Berkeley ones, because Pochoir had to cope with load imbalances due to the scheduling of 8 threads on 12 cores. Notwithstanding these issues, as can be seen from the figure, Pochoir's performance is generally comparable to that of the Berkeley autotuner on these two benchmarks.

The Pochoir-generated TRAP code is a cache-oblivious [15, 38] divide-and-conquer algorithm based on the notion of ***trapezoidal decompositions*** introduced by Frigo and Strumpen [16, 17]. We improve on their code by using ***hyperspace*** cuts, which produce an asymptotic improvement in parallelism while attaining essentially the same cache efficiency. As can be seen from Figure 2, however, this divide-and-conquer parallel code is far more complex than LOOPS, involving recursion over irregular geometric regions. Moreover, TRAP presents many opportunities for optimiza-

Benchmark	*Dims*	*Grid size*	*Time steps*	*Pochoir* 1 *core*	*Pochoir* 12 *cores*	*Pochoir speedup*	*Serial loops time*	*Serial loops ratio*	12-*core loops time*	12-*core loops ratio*
Heat	2	$16,000^2$	500	277s	24s	11.5	612s	25.5	149s	6.2
Heat	2p	$16,000^2$	500	281s	24s	11.7	1,647s	68.6	248s	10.3
Heat	4	150^4	100	154s	54s	2.9	433s	8.0	104s	1.9
Life	2p	$16,000^2$	500	345s	28s	12.3	2,419s	86.4	332s	11.9
Wave	3	$1,000^3$	500	3,082s	447s	6.9	3,170s	7.1	1,071s	2.4
LBM	3	$100^2 \times 130$	3,000	345s	68s	5.1	304s	4.5	220s	3.2
RNA	2	300^2	900	90s	20s	4.5	121s	6.1	26s	1.3
PSA	1	100,000	200,000	105s	18s	5.8	432s	24.0	77s	4.3
LCS	1	100,000	200,000	57s	9s	6.3	105s	11.7	27s	3.0
APOP	1	2,000,000	10,000	43s	4s	10.7	515s	128.8	48s	12.0

Figure 3: Pochoir performance on an Intel Core i7 (Nehalem) machine. The stencils are nonperiodic unless the *Dims* column contains a "p." The header *Serial loops* means a serial `for` loop implementation running on one core, whereas 12-*core loops* means a parallel `cilk_for` loop implementation running on 12 cores. The header *ratio* indicates how much slower the looping implementation is than the 12-core Pochoir implementation. For nonperiodic stencils, the looping implementations employ ghost cells [8] to avoid boundary processing.

	Berkeley	*Pochoir*
CPU	Xeon X5550	Xeon X5650
Clock	2.66GHz	2.66 GHz
cores/socket	4	6
Total # cores	8	12
Hyperthreading	Enabled	Disabled
L1 data cache/core	32KB	32KB
L2 cache/core	256KB	256KB
L3 cache/socket	8MB	12 MB
Peak computation	85 GFLOPS	120 GFLOPS
Compiler	icc 10.0.0	icc 12.0.0
Linux kernel		2.6.32
Threading model	Pthreads	Cilk Plus
3D 7-point	2.0 GStencil/s	2.49 GStencil/s
8 cores	15.8 GFLOPS	19.92 GFLOPS
3D 27-point	0.95 GStencil/s	0.88 GStencil/s
8 cores	28.5 GFLOPS	26.4 GFLOPS

Figure 5: A comparison of Pochoir to the reported results from [8]. The 7-point stencil requires 8 floating-point operations per grid point, whereas the 27-point stencil requires 30 floating-point operations per grid point.

tion, including coarsening the base case of the recursion and handling boundary conditions. We contend that one cannot expect average application programmers to be able to write such complex high-performing code for each stencil computation they wish to perform.

The Pochoir stencil compiler allows programmers to write simple functional specification for arbitrary d-dimensional stencils, and then it automatically produces a highly optimized, cache-efficient, parallel implementation. The Pochoir language can be viewed as a domain-specific language [10, 21, 31] embedded in the base language C++ with the Cilk multithreading extensions [23].

As shown in Figure 4, the Pochoir system operates in two phases, only the second of which involves the Pochoir compiler itself. For the first phase, the programmer compiles the source program with the ordinary Intel C++ compiler using the Pochoir template library, which implements Pochoir's linguistic constructs using unoptimized but functionally correct algorithms. This phase ensures that the source program is ***Pochoir-compliant***. For the second phase, the programmer runs the source through the Pochoir compiler, which acts as a preprocessor to the Intel C++ compiler, performing a source-to-source translation into a postsource C++ program that employs the Cilk extensions. The postsource is then compiled with the Intel compiler to produce the optimized binary executable. The Pochoir compiler makes the following promise:

> **The Pochoir Guarantee:** If the stencil program compiles and runs with the Pochoir template library during Phase 1, no errors will occur during Phase 2 when it is compiled with the Pochoir compiler or during the subsequent running of the optimized binary.

```
1  #define mod(r,m) ((r)%(m) + ((r)<0)? (m):0)

2  Pochoir_Boundary_2D(heat_bv, a, t, x, y)
3    return a.get(t,mod(x,a.size(1)),mod(y,a.size(0)));
4  Pochoir_Boundary_End

5  int main(void) {
6    const int X = 1000, Y = 1000, T = 1000;

7    Pochoir_Shape_2D 2D_five_pt[] = {{1,0,0}, {0,0,0},
       {0,1,0}, {0,-1,0}, {0,0,-1}, {0,0,1}};
8    Pochoir_2D heat(2D_five_pt);

9    Pochoir_Array_2D(double) u(X, Y);
10   u.Register_Boundary(heat_bv);
11   heat.Register_Array(u);

12   Pochoir_Kernel_2D(heat_fn, t, x, y)
13     u(t+1, x, y) = CX * (u(t, x+1, y) - 2 * u(t, x,
         y) + u(t, x-1, y)) + CY * (u(t, x, y+1) - 2
         * u(t, x, y) + u(t, x, y-1)) + u(t, x, y);
14   Pochoir_Kernel_End

15   for (int x = 0; x < X; ++x)
16     for (int y = 0; y < Y; ++y)
17       u(0, x, y) = rand();

18   heat.Run(T, heat_fn);

19   for (int x = 0; x < X; ++x)
20     for (int y = 0; y < Y; ++y)
21       cout << u(T, x, y);


23   return 0;
24 }
```

Figure 6: The Pochoir stencil source code for a periodic 2D heat equation. Pochoir keywords are boldfaced.

Pochoir's novel two-phase compilation strategy allowed us to build significant domain-specific optimizations into the Pochoir compiler without taking on the massive job of parsing and type-checking the full C++ language. Knowing that the source program compiles error-free with the Pochoir template library during Phase 1 allows the Pochoir compiler in Phase 2 to treat portions of the source as uninterpreted text, confident that the Intel compiler will compile it correctly in the optimized postsource. Moreover, the Pochoir template library allows the programmer to debug his or her code using a comfortable native C++ tool chain without the complications of the Pochoir compiler.

Figure 6 shows the Pochoir source code for the periodic 2D heat equation. We leave the specification of the Pochoir language to Section 2, but outline the salient features of the language using this code as an example.

Line 7 declares the ***Pochoir shape*** of the stencil, and line 8 cre-

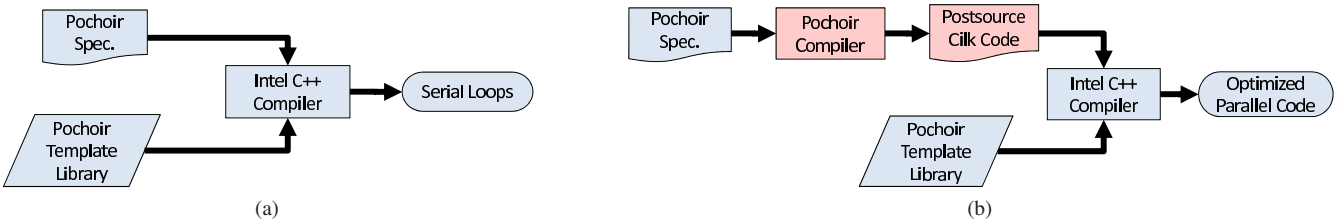


Figure 4: Pochoir's two-phase compilation strategy. (a) During Phase 1 the programmer uses the normal Intel C++ compiler to compile his or her code with the Pochoir template library. Phase 1 verifies that the programmer's stencil specification is Pochoir compliant. (b) During Phase 2 the programmer uses the Pochoir compiler, which acts as a preprocessor to the Intel C++ compiler, to generate optimized multithreaded Cilk code.

ates the 2-dimensional ***Pochoir object*** `heat` having that shape. The Pochoir object will contain all the state necessary to perform the computation. Each triple in the array `2D_five_pt` corresponds to a relative offset from the space-time grid point (t,x,y) that the stencil kernel (declared in lines 12–14) will access. The compiler cannot infer the stencil shape from the kernel, because the kernel can be arbitrary code, and accesses to the grid points can be hidden in subroutines. The Pochoir template library complains during Phase 1, however, if an access to a grid point during the kernel computation falls outside the region specified by the shape declaration.

Line 9 declares `u` as an `X×Y` ***Pochoir array*** of double-precision floating-point numbers representing the spatial grid. Lines 2–4 define a ***boundary function*** that will be called when the kernel function accesses grid points outside the computing domain, that is, if it tries to access `u(t, x, y)` with $x < 0$, $x \geq X$, $y < 0$, or $y \geq Y$. The boundary function for this periodic stencil performs calculations modulo the dimensions of the spatial grid. (Section 2 shows how nonperiodic stencils can be specified, including how to specify Dirichlet and Neumann boundary conditions [14].) Line 10 associates the boundary function `heat_bv` with the Pochoir array `u`. Each Pochoir array has exactly one boundary function to supply a value when the computation accesses grid points outside of the computing domain. Line 11 registers the Pochoir array `u` with the `heat` Pochoir object. A Pochoir array can be registered with more than one Pochoir object, and a Pochoir object can have multiple Pochoir arrays registered.

Lines 12–14 define a ***kernel function*** `heat_fn`, which specifies how the stencil is computed for every grid point. This kernel can be an arbitrary piece of code, but accesses to the registered Pochoir arrays must respect the declared shape(s).

Lines 15–17 initialize the Pochoir array `u` with values for time step 0. If a stencil depends on more than one prior step as indicated by the Pochoir shape, multiple time steps may need to be initialized. Line 18 executes the stencil object `heat` for T time steps using kernel function `heat_fn`. Lines 19–21 prints the result of the computation by reading the elements `u(T, x, y)` of the Pochoir array. In fact, Pochoir overloads the "<<" operator so that the Pochoir array can be pretty-printed by simply writing "`cout << u;`".

The remainder of this paper is organized as follows. Section 2 provides a full specification of the Pochoir embedded language. Section 3 describes the cache-oblivious parallel algorithm used by the compiled code and analyzes its theoretical performance. Section 4 describes four important optimizations employed by the Pochoir compiler. Section 5 describes related work, and Section 6 offers some concluding remarks.

2. THE POCHOIR SPECIFICATION LANGUAGE

This section describes the formal syntax and semantics of the Pochoir language, which was designed with a view to offer as much expressiveness as possible without violating the Pochoir Guarantee. Since we wanted to allow third-party developers to implement their own stencil compilers that could use the Pochoir specification language, we avoided to the extent possible making the language too specific to the Pochoir compiler, the Intel C++ compiler, and the multicore machines we used for benchmarking.

The static information about a Pochoir stencil computation, such as the computing kernel, the boundary conditions, and the stencil shape, is stored in a ***Pochoir object***, which is declared as follows:

- **`Pochoir_`***dim***`D`** *name* `(` *shape* `);`

This statement declares *name* as a Pochoir object with *dim* spatial dimensions and computing shape *shape*, where *dim* is a small positive integer and *shape* is an array of arrays which describes the shape of the stencil as elaborated below.

We now itemize the remaining Pochoir constructs and explain the semantics of each.

- **`Pochoir_Shape_`***dim***`D`** *name* `[] = {`*cells*`}`

This statement declares *name* as a ***Pochoir shape*** that can hold shape information for *dim* spatial dimensions. The Pochoir shape is equivalent to an array of arrays, each of which contains $dim+1$ integer numbers. These numbers represent the offset of each memory footprint in the stencil kernel relative to the space-time grid point $\langle t,x,y,\cdots\rangle$. For example, suppose that the computing kernel employs the following update equation:

$$\begin{aligned} u_t(x,y) &= u_{t-1}(x,y) \\ &\quad + \frac{\alpha\Delta t}{\Delta x^2}(u_{t-1}(x-1,y) + u_{t-1}(x+1,y) - 2u_{t-1}(x,y)) \\ &\quad + \frac{\alpha\Delta t}{\Delta y^2}(u_{t-1}(x,y-1) + u_{t-1}(x,y+1) - 2u_{t-1}(x,y)) \ . \end{aligned}$$

The shape of this stencil is $\{\{0,0,0\}, \{-1,1,0\}, \{-1,0,0\}, \{-1,-1,0\}, \{-1,0,1\}, \{-1,0,-1\}\}$.

The first cell in the shape is the ***home*** cell, whose spatial coordinates must all be 0. During the computation, this cell corresponds to the grid point being updated. The remaining cells must have time offsets that are smaller than the time coordinate of the home cell, and the corresponding grid points during the computation are read-only.

The ***depth*** of a shape is the time coordinate of the home cell minus the minimum time coordinate of any cell in the shape. The depth corresponds to the number of time steps on which a grid point depends. For our example stencil, the depth of the shape is 1, since a point at time t depends on points at time $t-1$.. If a stencil shape has depth k, the programmer must initialize all Pochoir arrays for time steps $0,1,\ldots,k-1$ before running the computation.

- **`Pochoir_Array_`***dim***`D`** (*type*, *depth*) *name* (*size*$_{dim-1}$, …, *size*$_1$, *size*$_0$)

This statement declares *name* as a ***Pochoir array*** of type *type* with *dim* spatial dimensions and a temporal dimension. The size of the

ith spatial dimension, where $i \in \{0, 1, \ldots, dim\}$, is given by $size_i$. The temporal dimension has size $k+1$, where k is the depth of the Pochoir shape, and are reused modulo $k+1$ as the computation proceeds. The user may not obtain an alias to the Pochoir array or its elements.

- **Pochoir_Boundary**_*dim*`D`$(name, array, idx_t, idx_{dim-1}, \ldots, idx_1, idx_0)$
  $\langle definition \rangle$
  Pochoir_Boundary_End

This construct defines a ***boundary function*** called *name* that will be invokeda to supply a value when the stencil computation accesses a point outside the domain of the Pochoir array *array*. The Pochoir array *array* has *dim* spatial dimensions, and $\langle idx_{dim-1}, \ldots, idx_1, idx_0 \rangle$ are the spatial coordinates of the given point outside the domain of *array*. The coordinate in the time dimension is given by idx_t. The function body $\langle definition \rangle$) is C++ code that defines the values of *array* on its boundary. A current restriction is that this construct must be declared outside of any function, that is, the boundary function is declared global.

- **Pochoir_Kernel**_*dim*`D`$(name, array, idx_t, idx_{dim-1}, \ldots, idx_1, idx_0)$
  $\langle definition \rangle$
  Pochoir_Kernel_End

This construct defines a ***kernel function*** named *name* for updating a stencil on a spatial grid with *dim* spatial dimensions. The spatial coordinates of the point to update are $\langle idx_{dim-1}, \ldots, idx_1, idx_0 \rangle$, and idx_t is the coordinate in time dimension. The function body $\langle definition \rangle$ may contain arbitrary C++ code to compute the stencil. Unlike boundary functions, this construct can be defined in any context.

- *name*.**Register_Array**$(array)$

A call to this member function of a Pochoir object *name* informs *name* that the Pochoir array *array* will participate in its stencil computation.

- *name*.**Register_Boundary**$(bdry)$

A call to this member function of a Pochoir array *name* associates the declared boundary function *bdry* with *name*. The boundary function is invoked to supply a value whenever an off-domain memory access occurs. Each Pochoir array is associated with exactly one boundary function at any given time, but the programmer can change boundary functions by registering a new one.

- *name*.**Run**$(T, kern)$

This function call runs the stencil computation on the Pochoir object *name* for T time steps using computing kernel function *kern*.

After running the computation for T steps, the results of the computation can be accessed by indexing its Pochoir arrays at time $T+k-1$, where k is the depth of the stencil shape. The programmer may resume the running of the stencil after examining the result of the computation by calling *name*.`Run`$(T', kern)$, where T' is the number of additional steps to execute. The result of the computation is then in the computation's Pochoir arrays indexed by time $T+T'+k-1$.

Rationale

The Pochoir language is a product of many design decisions, some of which were influenced by the current capabilities of the Intel 12.0.0 C++ compiler. We now discuss some of the more important design decisions.

Although we chose to pass a kernel function to the `Run` method of a Pochoir object, we would have preferred to simply store the kernel function with the Pochoir object. The kernel function is a C++ lambda function [5], however, whose type is not available to us. Thus, although we can pass the lambda function as a template type, we cannot store it unless we create a `std::function` to capture its type. Since the Intel compiler does not yet support `std::function`, this avenue was lost to us. There is only one kernel function per Pochoir object, however, and so we decided as a second-best alternative that it would be most convenient for users if they could declare a kernel function in any context and we just pass it as an argument to the `Run` member function.

The lack of support for function objects also had an impact on the declaration of boundary functions. We wanted to store each boundary function with a Pochoir array so that whenever an access to the array falls outside the computing domain, we can call the boundary function to supply a value. The only way to create a function that can be stored is to use an ordinary function, which must be declared in a global scope. We hope to improve Pochoir's linguistic design when function objects are fully supported by the compiler.

We chose to specify the kernel function imperatively rather than as a pure function or as an expression that returns a value for the grid point being updated. This approach allows a user to write multiple statements in a kernel function and provides flexibility on how to specify a stencil formula. For example, the user can choose to specify a stencil formula as `a(t, i, j) = ...` or `a(t+1, i, j) = ...`, whichever is more convenient.

We chose to make the user copy data in and out of Pochoir internal data structures, rather than operate directly on the user's arrays. Since the user is typically running the stencil computation for many time steps, we decided that the copy-in/copy-out approach would not cause much overhead. Moreover, the layout of data is now under the control of the compiler, allowing it to optimize the storage for cache efficiency.

3. POCHOIR'S CACHE-OBLIVIOUS PARALLEL ALGORITHM

This section describes the parallel algorithm at the core of Pochoir's efficiency. TRAP is a cache-oblivious algorithm based on "trapezoidal decompositions" [16, 17], but which employs a novel "hyperspace-cut" strategy to improve parallelism without sacrificing cache-efficiency. On a d-dimensional spatial grid with all "normalized" spatial dimensions equal to w and the time dimension a power-of-2 multiple of w, TRAP achieves $\Theta(w^{d-\lg(d+2)+1}/d^2)$ parallelism, whereas Frigo and Strumpen's original parallel trapezoidal decomposition algorithm [17] achieves $\Theta(w^{d-\lg(2^d+1)+1}/2^d) = O(w)$ parallelism. Both algorithms exhibit the same asymptotic cache complexity of $\Theta(hw^d/\mathcal{M}^{1/d}\mathcal{B})$ proved by Frigo and Strumpen, where h is the height of the time dimension, $\mathcal{M}$ is the cache size, and $\mathcal{B}$ is the cache-block size.

TRAP uses a cache-oblivious [15] divide-and-conquer strategy based on a recursive trapezoidal decomposition of the space-time grid, which was introduced by Frigo and Strumpen [16]. They originally used the technique for serial stencil computations, but later extended it to parallel stencil computations [17]. Whereas Frigo and Strumpen's parallel algorithm cuts the spatial dimensions of a hypertrapezoid, or "zoid," one at a time with "parallel space cuts," TRAP performs a ***hyperspace cut*** where it applies parallel space cuts simultaneously to as many dimensions as possible, yielding asymptotically more parallelism when the number of spatial dimensions is 2 or greater. As we will argue later in this section, TRAP achieves this improvement in parallelism while attaining the

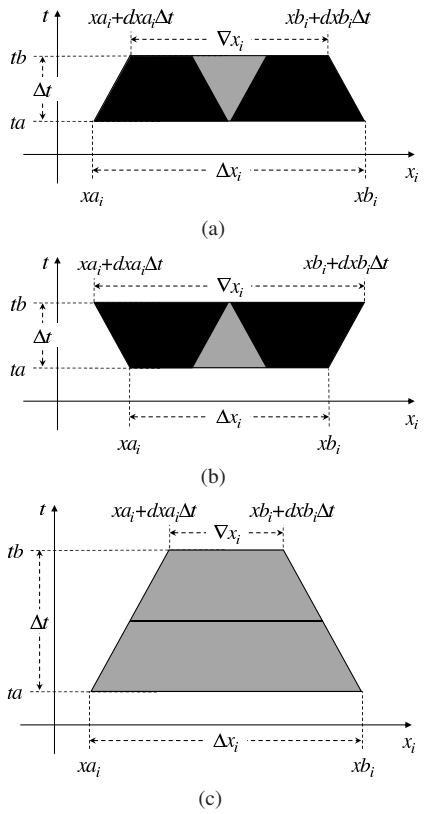


Figure 7: Cutting projection trapezoids. The spatial dimension increases to the right, and the time runs upward. (a) Trisecting an upright trapezoid using a parallel space cut produces two black trapezoids that can be processed in parallel and a gray trapezoid that must be processed after the black ones. (b) Trisecting an inverted trapezoid using a parallel space cut produces two black trapezoids that can be processed in parallel and a gray trapezoid that must be processed before the black ones. (c) A time cut produces a lower and an upper trapezoid where the lower trapezoid must be processed before the upper.

same cache complexity as Frigo and Strumpen's original parallel algorithm.

TRAP operates as follows. Line 5 of Figure 2 determines whether the x-dimension of the zoid can be cut with a parallel space cut, and if so, line 6 trisects the zoid, as we shall describe later in this section and in Figure 7, but it does not immediately spawn recursive tasks to process the subzoids, as Frigo and Strumpen's algorithm would. Instead, the code attempts to make a "hyperspace cut" by proceeding to the y-dimension, and if there were more dimensions, to those, cutting as many dimensions as possible before spawning recursive tasks to handle the subzoids. The counter k keeps track of how many spatial dimensions are cut. If $k > 0$ spatial dimensions are trisected, as tested for in line 11, then line 12 assigns each subzoid to one of $k+1$ dependency levels such that the subzoids assigned to the same level are independent and can be processed in parallel, as we describe later in this section and in Figure 8. Lines 13–15 recursively walk all subzoids level by level in parallel. Lines 17–19 perform a time cut if no space cut can be performed. Lines 20–28 perform the base-case computation if the zoid is sufficiently small that no space or time cut is productive.

We first introduce some notations and definitions, many of which have been borrowed or adapted from [16, 17]. A $(d+1)$-dimensional ***space-time hypertrapezoid***, or $(d+1)$-***zoid***, $Z = (ta, tb;\ xa_0, xb_0, dxa_0, dxb_0;\ xa_1, xb_1, dxa_1, dxb_1;\ \ldots;\ xa_{d-1}, xb_{d-1}, dxa_{d-1}, dxb_{d-1})$, where all variables are integers, is the set of integer grid points $\langle t, x_0, x_1, \ldots, x_{d-1}\rangle$ such that $ta \le t < tb$ and $xa_i + dxa_i(t-ta) \le x_i < xb_i + dxb_i(t-ta)$ for all $i \in \{0,1,\ldots,d-1\}$. The ***height*** of Z is $\Delta t = tb - ta$. Define the ***projection trapezoid*** Z_i of Z along spatial dimension i to be the 2D trapezoid that results from projecting the zoid Z onto the dimensions x_i and t. The projection trapezoid Z_i has two ***bases*** (sides parallel to the x_i axis) of lengths $\Delta x_i = xb_i - xa_i$ and $\nabla x_i = (xb_i + dxb_i\Delta t) - (xa_i + dxa_i\Delta t)$. We define the ***width***[1] w_i of Z_i to be the length of the longer of the two bases (parallel sides) of Z_i, that is $w_i = \max\{\Delta x_i, \nabla x_i\}$. The value w_i is also called the ***width*** of Z along spatial dimension i. We say that Z_i is ***upright*** if $w_i = \Delta x_i$ — the longer base corresponds to time ta — and ***inverted*** otherwise. A zoid Z is ***well-defined*** if its height is positive, its widths along all spatial dimensions are positive, and the lengths of its bases along all spatial dimensions are nonnegative. A projection trapezoid Z_i is ***minimal*** if Z_i is upright and $\nabla x_i = 0$, or Z_i is inverted and $\Delta x_i = 0$. A zoid Z is ***minimal*** if all its Z_i's are minimal.

Given the shape S of a d-dimensional stencil (as described in Section 2), define t_{home} be the time index of the home cell. We define the ***slope***[2] of a cell $c = (t, x_0, x_1, \ldots, x_{d-1}) \in S$ along dimension $i \in \{0,1,\ldots,d-1\}$ as $\sigma_i(c) = |x_i/(t_{\text{home}} - t)|$, and we define the ***slope*** of the stencil along spatial dimension i as $\sigma_i = \max_{c\in S} \lceil \sigma_i(c) \rceil$. (Pochoir assumes for simplicity that the stencil is symmetric in each dimension.) We define the ***normalized width*** of a zoid Z along dimension i by $\widehat{w}_i = w_i/2\sigma_i$.

Parallel space cuts

Our trapezoidal decomposition differs from that of Strumpen and Frigo in the way we do parallel space cuts. A ***parallel space cut*** can be applied along a given spatial dimension i of a well-defined zoid Z provided that the projection trapezoid Z_i can be trisected into 3 well-defined subtrapezoids, as shown in Figures 7(a) and 7(b). The triangle-shaped gray subtrapezoid that lies in the middle is a minimal trapezoid. The larger base of Z_i is split in half with each half forming the larger base of a black subtrapezoid. These three subtrapezoids of Z_i correspond to three subzoids of Z. Since the two black subzoids have no interdependencies, they can be processed in parallel. As shown in Figure 7(a), for an upright projection trapezoid, the subzoids corresponding to the black trapezoids are processed first, after which the subzoid corresponding to the gray subtrapezoid can be processed. For an inverted projection trapezoid, as shown in Figure 7(b), the opposite is done. In either case, the 3 subzoids can be processed in parallel in the time to process 2 of them, what we shall call 2 ***parallel steps***. The following lemma describes the general case.

LEMMA 1. *All 3^k subzoids created by a hyperspace cut on $k \ge 1$ of the $d \ge k$ spatial dimensions of a $(d+1)$-zoid Z can be processed in $k+1$ parallel steps.*

PROOF. Assume without loss of generality that the hyperspace cut is applied to the first k spatial dimensions of Z. For each such dimension i, label the projection subtrapezoids in 2D space-time resulting from the parallel space cut (see Figures 7(a) and 7(b)) with the numbers 1, 2, and 3, where the black trapezoids are labeled 1 and 3 and the gray trapezoid is labeled 2. When the hyperspace cut consisting of all k parallel space cuts is applied, it creates a set S of 3^k subzoids in $(k+1)$-dimensional space-time. Each subzoid can be identified by a unique k-tuple $\langle u_0, u_1, \ldots, u_{k-1}\rangle$, where $u_i \in \{1,2,3\}$ for $i = 0,1,\ldots,k-1$. Let $I_i = 1$ if the projection trapezoid

[1] Frigo and Strumpen [16, 17] define width as the average of the two bases.
[2] Actually, the reciprocal of slope, but we follow Frigo and Strumpen's terminology.

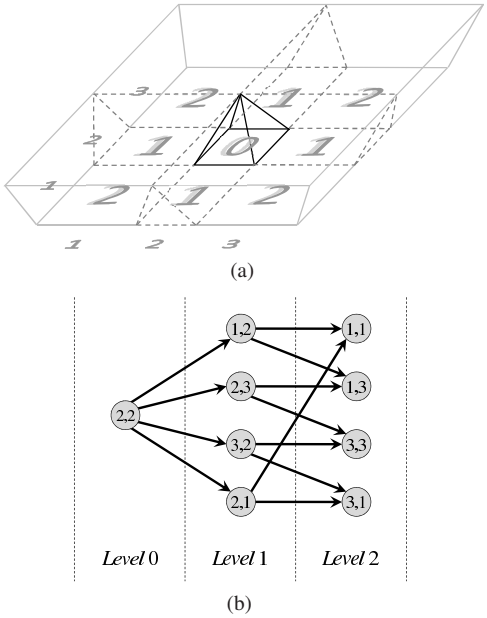


Figure 8: Dependency levels of subzoids resulting from a hyperspace cut along both spatial dimensions of a 3-zoid. (a) Labeling of coordinates of subzoids and their dependency levels. (b) The corresponding dependency graph.

$\mathcal{Z}_i$ along the ith dimension is upright and $I_i = 0$ if $\mathcal{Z}_i$ is inverted. The ***dependency level*** of a zoid $\langle u_0, u_1, \ldots, u_{k-1} \rangle \in S$ is given by

$$dep(\langle u_0, u_1, \ldots, u_{k-1} \rangle) = \sum_{i=0}^{k-1} ((u_i + I_i) \bmod 2) \ .$$

Observe that this equation implies exactly $k+1$ dependency levels, since each term of the summation may be either 0 or 1. Figure 8(a) shows the dependency levels for the subzoids of a 3-zoid, both of whose projection trapezoids are inverted, generated by a hyperspace cut with $k = 2$.

We claim that all zoids in S with the same dependency level are independent, and thus all of S can be processed in $k+1$ parallel steps. As illustrated in Figure 8(b), we can construct a directed graph $G = (S, E)$ that captures the dependency relationships among the subzoids of S as follows. Given any pair of zoids $\langle u_0, u_1, \ldots, u_{k-1} \rangle, \langle u'_0, u'_1, \ldots, u'_{k-1} \rangle \in S$, we include an edge $(\langle u_0, u_1, \ldots, u_{k-1} \rangle, \langle u'_0, u'_1, \ldots, u'_{k-1} \rangle) \in E$, meaning that a grid point in $\langle u'_0, u'_1, \ldots, u'_{k-1} \rangle$ directly depends on a grid point in $\langle u_0, u_1, \ldots, u_{k-1} \rangle$, if there exists a dimension $i \in \{0, 1, \ldots, k-1\}$ such that the following conditions hold:

- $u_j = u'_j$ for all $j \in \{0, 1, \ldots, i-1, i+1, \ldots, k-1\}$,
- $(I_i + u_i) \bmod 2 = 0$,
- $(I_i + u'_i) \bmod 2 = 1$.

Under these conditions, we have $dep(\langle u'_0, u'_1, \ldots, u'_{k-1} \rangle) = dep(\langle u_0, u_1, \ldots, u_{k-1} \rangle) + 1$. Thus, along any path in G, the dependency levels are strictly increasing, and no two nodes with the same dependency level can lie on the same path. As a result, all zoids in S with the same dependency level form an antichain and can be processed simultaneously. Thus, all zoids in S can be processed in $k+1$ parallel steps with step $s \in \{0, 1, \ldots, k\}$ processing all zoids having dependency level s. □

Pochoir's cache-oblivious parallel algorithm

Given a well-defined zoid $\mathcal{Z}$, the algorithm TRAP from Figure 2 works by recursively decomposing $\mathcal{Z}$ into smaller well-defined zoids as follows.

Hyperspace cut. Lines 4–10 in Figure 2 apply a hyperspace cut involving all dimensions on which a parallel space cut can be applied, as shown in Figures 7(a) and 7(b). If the number k of dimensions of $\mathcal{Z}$ on which a space cut can be applied is at least 1, as tested for in line 11 of Figure 2, then dependency levels are computed for all resulting subzoids in line 12, and then lines 13–15 recursively process them in order according to dependency level as described in the proof of Lemma 1.

Time cut. If a hyperspace cut is not applicable and $\mathcal{Z}$ has height greater than 1, as tested for in line 16, then lines 17–19 cut $\mathcal{Z}$ in the middle of its time dimension and recursively process the lower subzoid followed by the upper subzoid, as shown in Figure 7(c).

Base case. If neither a hyperspace cut nor a time cut can be applied, lines 20–28 processes $\mathcal{Z}$ directly by invoking the stencil-specific kernel function. In practice, the base case is ***coarsened*** (see Section 4) by choosing a suitable threshold larger than 1 for Δt in line 16, which cuts down on overhead due to the recursion.

Analysis

We can analyze the parallelism using a work/span analysis [7, Ch. 27]. The ***work*** T_1 of a computation is its serial running time, and the ***span*** T_∞ is the longest path of dependencies, or equivalently, the running time on an infinite number of processors assuming no overheads for scheduling. The ***parallelism*** of a computation is the ratio T_1/T_∞ of work to span.

The next lemma provides a tight bound on the span of TRAP algorithm on a minimal zoid.

LEMMA 2. *Consider a minimal $(d+1)$-zoid $\mathcal{Z}$ with height h and normalized widths $\widehat{w}_i = h$ for $i \in \{0, 1, \ldots, d-1\}$. Then the span of* TRAP *when processing $\mathcal{Z}$ is $\Theta(dh^{\lg(d+2)})$.*

PROOF. For simplicity we assume that a call to the kernel function costs $O(1)$, as in [17]. As TRAP processes $\mathcal{Z}$, some of the subzoids generated recursively have normalized widths equal to their heights and some have twice that amount. Let us denote by $T_\infty(h, k, d-k)$ the span of TRAP processing a $(d+1)$-zoid with height h where $k \geq 0$ of the d spatial dimensions have normalized width $2h$ and $d-k$ spatial dimensions have normalized width h. Using Lemma 1, the span of TRAP processing a zoid $\mathcal{Z}$ when it undergoes a hyperspace cut can be described by the recurrence

$$\begin{aligned} T_\infty(h, k, d-k) &= (k+1) T_\infty(h, 0, d) + \Theta\left(\sum_{i=0}^{k} \lg(3^k)\right) \\ &= (k+1) T_\infty(h, 0, d) + \Theta(k^2) \ , \end{aligned}$$

where $T(1, 0, d) = \Theta(1)$ is the base case. The summation in this derivation represents the span due to spawning. A parallel **for** with r iterations adds $\Theta(\lg r)$ to the span, and since the number of zoids at all levels is 3^k, this value upper-bounds the number of iterations at any given level. Moreover, the lower bound on the number of zoids on a given level is at least the average $3^k/(k+1)$, whose logarithm is asymptotically the same as $\lg(3^k)$, and hence the bound is asymptotically tight.

A time cut can be applied when the zoid $\mathcal{Z}$ is minimal. Assume that $k \geq 0$ projection trapezoids $\mathcal{Z}_i$'s are upright and the rest are inverted. Then for each upright projection trapezoid $\mathcal{Z}_i$, the normalized width of the lower zoid generated by the hyperspace cut is $\widehat{w}_i = h$, the same as for $\mathcal{Z}$, and for each inverted projection trapezoid $\mathcal{Z}_i$, the lower zoid has normalized width $\widehat{w}_i - h/2 = h/2$. Similarly, for each upright projection trapezoid $\mathcal{Z}_i$, the normalized width of the upper zoid is $\widehat{w}_i - h/2 = h/2$, and for each inverted projection trapezoid $\mathcal{Z}_i$, the upper zoid has normalized width $\widehat{w}_i$. Thus, the

recurrence for the span of TRAP when a minimal $\mathcal{Z}$ undergoes a time cut can be written as follows:

$$T_\infty(h,0,d) = T_\infty(h/2,k,d-k) + T_\infty(h/2,d-k,k) + \Theta(1) .$$

Applying hyperspace cuts to the subzoids on the right-hand side of this recurrence yields

$$\begin{aligned} T_\infty(h,0,d) &= (d+2)T_\infty(h/2,0,d) + \Theta(k^2) + \Theta((d-k)^2) \\ &= (d+2)T_\infty(h/2,0,d) + \Theta(d^2) \\ &= \Theta(d^2(d+2)^{\lg h-1}) + \Theta((d+2)^{\lg h}) \\ &= \Theta(dh^{\lg(d+2)}) . \quad \square \end{aligned}$$

THEOREM 3. *Consider a $(d+1)$-dimensional grid $\mathcal{Z}$ with $\widehat{w}_i = w$ for $i \in \{0,1,\ldots,d-1\}$ and height $h = 2^r w$. Then the parallelism of* TRAP *when processing $\mathcal{Z}$ using a stencil with constant slopes is $\Theta(w^{d-\lg(d+2)+1}/d^2)$.*

PROOF. Assume without loss of generality that the stencil is periodic. (As will be discussed in Section 4, Pochoir implements TRAP so that the control structure for nonperiodic stencils is the same as that for periodic.) The algorithm first applies a series of r time cuts, dividing the original time dimension into $h/w = 2^r$ subgrids with $\widehat{w}_i = w$ with height w. These grids are processed serially. The next action of TRAP applies a hyperspace cut to all d spatial dimensions of $\mathcal{Z}$, dividing the grid into $d+1$ minimal zoids which are then processed serially. Applying Lemma 2 yields a span of

$$\begin{aligned} T_\infty &= (h/w)(d+1) \cdot \Theta(dw^{\lg(d+2)}) \\ &= \Theta((d^2h)w^{\lg(d+2)-1}) . \end{aligned}$$

The work is the volume of $\mathcal{Z}$, which is $T_1 = \Theta(hw^d)$, since the stencil has constant slopes. Thus, the parallelism is

$$T_1/T_\infty = \Theta(w^{d-\lg(d+2)+1}/d^2) . \quad \square$$

We can compare TRAP with a version of Frigo and Strumpen's parallel stencil algorithm [17] we call STRAP, which performs the space cuts serially as in Figures 7(a) and 7(b). Each space cut results in one synchronization point, and hence a sequence of k space cuts applied by STRAP introduces 2^k parallel steps compared to the $k+1$ parallel steps generated by TRAP (see Lemma 1). Thus, each space cut virtually doubles STRAP's span. Figure 8(a) shows a simple example where STRAP produces $2^2 - 1 = 3$ synchronization points while TRAP introduces only 2. The next lemma and theorem analyze STRAP, mimicking Lemma 2 and Theorem 3. Their proofs are omitted.

LEMMA 4. *Consider a minimal $(d+1)$-zoid $\mathcal{Z}$ with height h and normalized widths $\widehat{w}_i = h$ for $i \in \{0,1,\ldots,d-1\}$. Then the span of* STRAP *when processing $\mathcal{Z}$ is $\Theta(h^{\lg(2^d+1)})$.* $\square$

THEOREM 5. *Consider a $(d+1)$-dimensional grid $\mathcal{Z}$ with $\widehat{w}_i = w$ for $i \in \{0,1,\ldots,d-1\}$ and height $h = 2^r w$. Then the parallelism of* STRAP *when processing $\mathcal{Z}$ using a stencil with constant slopes is $\Theta(w^{d-\lg(2^d+1)+1}/2^d)$.* $\square$

Discussion

As can be seen from Theorems 3 and 5, both TRAP and STRAP have the same asymptotic parallelism $\Theta(w^{2-\lg 3})$ for $d = 1$, but for $d = 2$, TRAP has $\Theta(w^2)$ while STRAP has $\Theta(w^{3-\lg 5})$, and the difference grows with the number of dimensions.

The cache complexities of TRAP and STRAP are the same, which follows from the observation that both algorithms apply exactly the same time cuts in exactly the same order, and immediately before each time cut, both are in exactly the same state in terms of the spatial cuts applied. Thus, they arrive at exactly the same configuration — number, shape, and size — of subzoids before each time cut.

Frigo and Strumpen's parallel stencil algorithm is actually slightly different from STRAP. For any fixed integer $r > 1$, a space cut in their algorithm produces r black zoids and between $r-1$ and $r+1$ gray zoids. STRAP is a special case of that algorithm with $r = 2$ for upright projection trapezoids and $r = 1$ for inverted projection trapezoids. For larger values of r, Frigo and Strumpen's algorithm achieves more parallelism but the cache efficiency drops. It is straightforward to extend TRAP to perform r multiple cuts along each dimension to match the cache complexity of Frigo and Strumpen's algorithm while providing asymptotically more parallelism.

Empirical results

Figure 9 shows the results of using the Cilkview scalability analyzer [20] to compare the parallelism of TRAP and STRAP on two typical benchmarks. We measured the two algorithms with uncoarsened base cases. As can be seen from the figure, TRAP's asymptotic advantage in parallelism is borne out in practice for these benchmarks.

We used the Linux perf tool [29] to verify that TRAP does not suffer any loss in cache efficiency compared to the STRAP algorithm. Figure 10 also plots the cache-miss ratio of the straightforward parallel loop algorithm, showing that it exhibits poorer cache performance than the two cache-oblivious algorithms.

4. COMPILER OPTIMIZATIONS

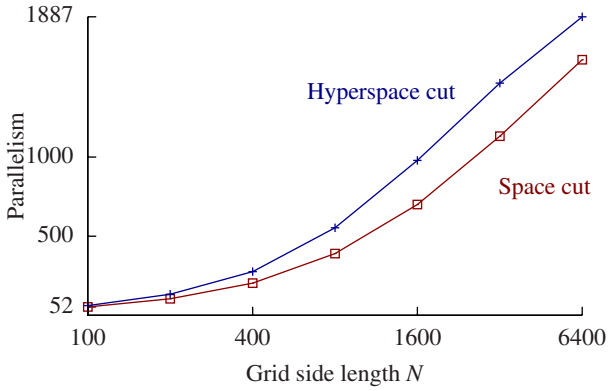
The Pochoir compiler transforms code written in the Pochoir specification language into optimized C++ code that employs the Intel Cilk multithreading extensions [23]. The Pochoir compiler is written in Haskell [37], and it performs numerous optimizations, the most important of which are code cloning, loop-index calculations, unifying periodic and nonperiodic boundary conditions, and coarsening the base case of recursion. This section describes how the Pochoir compiler implements these optimizations.

Before a programmer compiles a stencil code with the Pochoir compiler, he or she is expected to perform Phase 1 of Pochoir's two-phase methodology which requires that it be compiled using the Pochoir template library and debugged. This C++ template library is employed by both Phases 1 and 2 and includes both loop-based and trapezoidal algorithms. Differences between stencils, such as dimensionality or data structure, are incorporated into these generic algorithms at compile-time via C++ template metaprogramming.

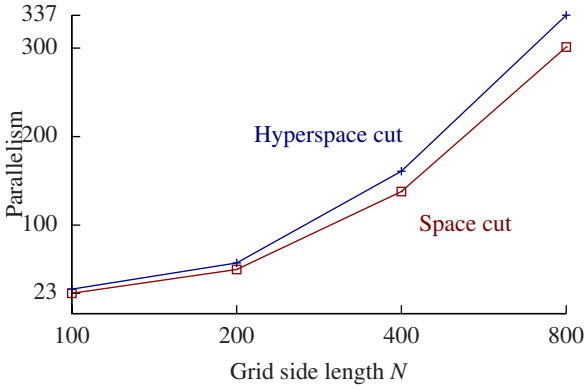
Handling boundary conditions by code cloning

The handling of boundary conditions can easily dominate the runtime of a stencil computation. For example, we coded the 2D heat equation on a periodic torus using Pochoir, and we compared it to a comparable code that simply employs a modulo operation on every array index. For a 5000^2 spatial grid over 5000 time steps, the runtime of the modular-indexing implementation degraded by a factor of 2.3.

For nonperiodic stencil computations, where a value must be provided on the boundary, performance can degrade even more if a test is made at every point to determine whether the index falls off the grid. Stencil implementers often handle constant nonperiodic boundary conditions with the simple trick of introducing ***ghost cells*** [8] that form a ***halo*** around the periphery of the grid. Ghost cells are read but never written. The stencil computation can apply the kernel function to the grid points on the real grid, and accesses

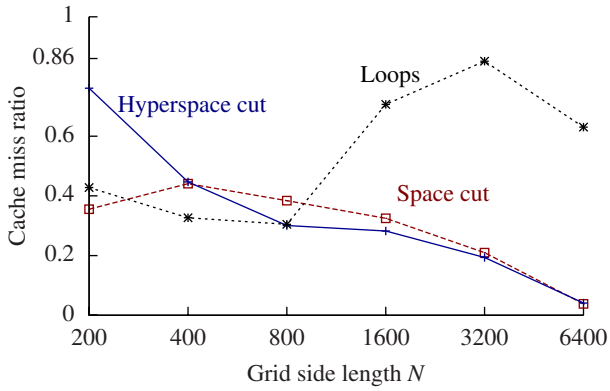


(a)

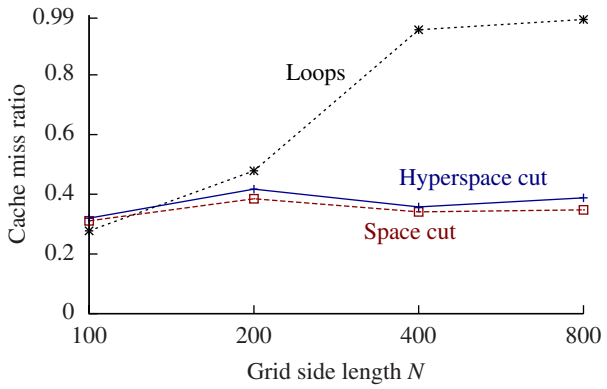


(b)

Figure 9: Parallelism comparison on two benchmarks between TRAP, which employs hyperspace cuts, and STRAP, which uses serial space cuts. Measurements are of code without base-case coarsening. (a) 2D nonperiodic heat equation. Space-time size is $1000N^2$. (b) 3D nonperiodic wave equation. Space-time size is $1000N^3$.



(a)



(b)

Figure 10: Cache-miss ratios for two benchmarks using TRAP, STRAP, and a parallel-loop algorithm. The cache-miss ratio is the ratio of the cache misses to the number of memory references. Measurements are of code without base-case coarsening. (a) 2D nonperiodic heat equation. Space-time is $1000N^2$. (b) 3D nonperiodic wave equation. Space-time is $1000N^3$.

```
1  Pochoir_Boundary_2D(dirichlet, arr, t, x, y)
2        return 100 + 0.2*t;
3  Pochoir_Boundary_End
```

(a)

```
1  Pochoir_Boundary_2D(neumann, arr, t, x, y)
2      int newx = x;
3      if (x < 0) newx = 0;
4      if (x >= arr.size(1)) newx = arr.size(1);
5      int newy = y;
6      if (y < 0) newy = 0;
7      if (y >= arr.size(0)) newy = arr.size(0);
8      return arr.get(t, newx, newy);
9  Pochoir_Boundary_End
```

(b)

Figure 11: Pochoir code for specifying nonperiodic boundary conditions. (a) A Dirichlet condition with constrained boundary value (set equal to a function of t). (b) A Neumann condition with constrained derivative at the boundary (set equal to 0).

that "fall off" the edge into the halo obtain their values from the ghost cells without any need to check boundary conditions.

In practice, however, nonperiodic boundary conditions can be more complicated than simple constants, and we wanted to allow Pochoir users flexibility in the kinds of boundary conditions they could specify. For example, Dirichlet boundary conditions may specify boundary values that change with time, and Neumann boundary conditions may specify the value the derivative should take on the boundary [14]. Figure 11(a) shows a Pochoir specification of a Dirichlet boundary condition, and Figure 11(b) shows the Pochoir specification of a Neumann boundary condition.

To handle boundaries efficiently, the Pochoir compiler generates two code clones of the kernel function: a slower ***boundary*** clone and a faster ***interior*** clone. The boundary clone is used for ***boundary*** zoids: those that contain at least one point whose computation requires an off-grid access. The interior clone is used for ***interior*** zoids: those all of whose points can be updated without indexing off the edge of the grid. Whether a zoid is interior or boundary is determined at runtime.

In the base case of the recursive trapezoidal decomposition, the boundary clone invokes the user-supplied boundary function to perform the relatively expensive checks on the coordinates of each point in the zoid to see whether they fall outside the boundary. If so, the user-supplied boundary function determines what value to use. The base case of the interior clone avoids this calculation, since it knows that no such test is necessary, and it simply accesses the necessary grid points.

The trapezoidal-decomposition algorithm exploits the fact that all subzoids of an interior zoid remain interior. If all the dimensions of the grid are approximately the same size, the boundary of the grid is much smaller than its (hyper)volume. Consequently, the faster interior clones dominate the running time, and the slower boundary clones contribute little.

```
1  Pochoir_Kernel_1D(heat_1D_fn, t, i)
2      a(t+1, i) = 0.125 * (a(t, i-1) + 2 * a(t, i) +
           a(t, i+1));
3  Pochoir_Kernel_End
```

(a)

```
1  /* a.interior() is a function to dereference the
       value without checking
2   * boundary conditions
3   */
4  #define a(t, i) a.interior(t, i)
5  Pochoir_Kernel_1D(heat_1D_fn, t, i)
6      a(t + 1, i) = 0.125 * (a(t, i - 1) + 2 * a(t, i
           ) + a(t, i + 1));
7  Pochoir_Kernel_End
8  #undef a(t, i)
```

(b)

```
1  Pochoir_Kernel_1D(heat_1D_fn, t, i)
2  /* The base address of the Pochoir array 'a' */
3  double *a_base = a.data();
4  /* Pointers to be used in the innermost loop */
5  double *iter0, *iter1, *iter2, *iter3;
6  /* Total size of the Pochoir array 'a' */
7  const int l_a_total_size = a.total_size();
8  int gap_a_0;
9  const int l_stride_a_0 = a.stride(0);
10 for (int t = ta; t < tb; ++t) {
11     double * baseIter_1;
12     double * baseIter_0;
13     baseIter_0 = a_base + ((t + 1) & 0xb) *
           l_a_total_size + (l_grid.xa[0]) *
           l_stride_a_0;
14     baseIter_1 = a_base + ((t) & 0xb) *
           l_a_total_size + (l_grid.xa[0]) *
           l_stride_a_0;
15     iter0 = baseIter_0 + (0) * l_stride_a_0;
16     iter1 = baseIter_1 + (-1) * l_stride_a_0;
17     iter2 = baseIter_1 + (0) * l_stride_a_0;
18     iter3 = baseIter_1 + (1) * l_stride_a_0;
19     for (int i = l_grid.xa[0]; i < l_grid.xb[0];
           ++i, ++iter0, ++iter1, ++iter2, ++iter3) {
20     (*iter0) = 0.125 * ((*iter1) + 2 * (*iter2) +
           (*iter3));  }
21 }
22 Pochoir_Kernel_End
```

(c)

Figure 12: Pochoir's loop-indexing optimizations illustrated on a 1D heat equation. (a) The original Pochoir code for the kernel function. (b) The code as transformed by `-split-macro-shadow`. (c) The code as transformed by `-split-pointer`.

Loop indexing

Because the interior zoids asymptotically dominate the computing time, most of the optimizations performed by Pochoir compiler focus on the interior clone. Two important optimizations relate to loop indexing. The particular optimization is chosen automatically by the Pochoir compiler, or it can be mandated by user as a command-line option. Consistent with their command-line names, the optimizations are called `-split-macro-shadow` and `-split-pointer`.

The `-split-macro-shadow` option causes the Pochoir compiler to employ macro tricks on the interior clone to eliminate the boundary-checking overhead. Consider the code snippet in Figure 12(a) which defines the kernel function for a 1D heat equation. Figure 12(b) shows the postsource code generated by the Pochoir compiler using `-split-macro-shadow`. Line 4 defines a macro that replaces the original accessing function `a`, which also does boundary checking, with one that performs the address calculation but without boundary checking.

The `-split-pointer` command-line option causes the Pochoir compiler to transform the indexing of Pochoir arrays in the interior clone into C-style pointer manipulation, as illustrated in Figure 12(c). A C-style pointer represents each term in the stencil formula. The resulting array indexing appears on line 20. For each consecutive iteration, the code increments each pointer. When iterating outer loops, the code adds a precomputed constant to each pointer as shown in lines 15–18.

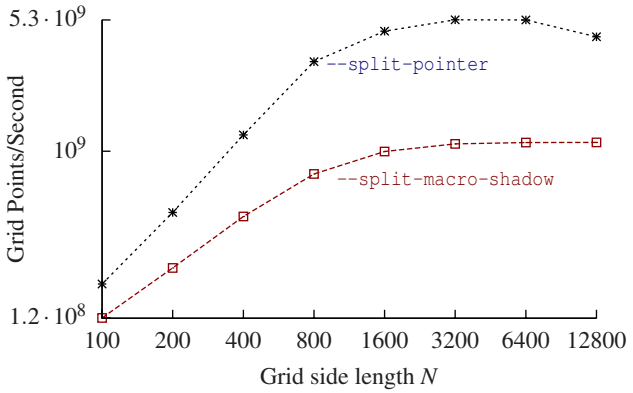


Figure 13: The performance of different loop-index optimizations on a 2D heat equation on torus. The grid is N^2 with 1000 time steps.

The Pochoir compiler tries to use the `-split-pointer` optimization if possible. It can do so if it can parse and "understand" the C++ syntax of the user's specification. Because our prototype Haskell compiler does not contain a complete C++ front end, however, it sometimes may not understand unusually complex C++ code written by the user, in which case, it employs the `-split-macro-shadow` optimization, relying on Phase 1 to ensure that the code is Pochoir-compliant.

Figure 13 compares the performances of the two optimizing options for a 2D heat equation on a torus. Other benchmarks show similar relative performances.

Unifying periodic and nonperiodic boundary conditions

Typical stencil codes discriminate between periodic and nonperiodic stencils, implementing them in different ways. To make the specification of boundary functions as flexible as possible, we investigated how periodic and nonperiodic stencils could be implemented using the same algorithmic framework, leaving the choice of boundary function up to the user. Our unified algorithm allows the user to program boundary functions with arbitrary periodic/nonperiodic behavior, providing support, for example, for a 2D cylindrical domain, where one dimension is periodic and the other is nonperiodic.

The key idea is to treat the entire computation as if it were periodic in all dimensions and handle nonperiodicity and other boundary conditions in the base case of the boundary clone where the kernel function is invoked. When a zoid wraps around the grid in a given dimension i, meaning that $xa_i > xb_i$, we represent the lower- and upper-bound coordinates of the zoid in dimension i by ***virtual*** coordinates $(xa_i, N_i + xb_i)$, where N_i is the size of the periodic grid in dimension i. In the base of the recursion of the boundary clone, Pochoir calls the kernel function and supplies it with the true coordinates of the grid point being updated by performing a modulo computation on each coordinate. Within the kernel function, accesses to the Pochoir arrays now call the boundary function, which provides the correct value for grid points that are outside the true grid. Of course, no such checking is required for interior zoids, which are always represented by true coordinates.

Coarsening of base cases

Previous work [9, 26, 27, 34] has found that although trapezoidal decomposition dramatically reduces cache-miss rates, overall performance can suffer from function-call overhead unless the base

case of the recursion is coarsened. For example, proper coarsening of the base case of the 2D heat-equation stencil (running for 5000 time steps on a 5000×5000 toroidal grid) improves the performance by a factor of 36 over running the recursion down to a single grid point.

Since choosing the optimal size of the base case can be difficult, we integrated the ISAT autotuner [22] into Pochoir. Despite the advantage of finding the optimal coarsening factor on any specific platform, this autotuning process can take hours to find the optimal value, which may be unacceptable for some users.

In practice, Pochoir employs some heuristics to choose a reasonable coarsening. One principle is that to maximize data reuse, we want to make the spatial dimensions all about the same size. Another principle is that to exploit hardware prefetching, we want to avoid cutting the unit-stride spatial dimension and avoid odd-shaped base cases. For example, for 2D problems, a square-shaped computing domain often offers the best performance. We have found that for 3D problems, the effect of hardware prefetching can often be more important than cache efficiency for reasonably sized base cases. Consequently, for 3 or more dimensions, Pochoir adopts the strategy of never cutting the unit-stride spatial dimension, and it cuts the rest of the spatial dimensions into small hypercubes to ensure that the entire base case stays in cache. Given all that potential complexity, the compiler's heuristic is actually fairly simple. For 2D problems, Pochoir stops the recursion at 100×100 space chunks with 5 time steps. For 3D problems, the recursion stops at $1000 \times 3 \times 3$ with 3 time steps.

5. RELATED WORK

Attempts to compile stencils into highly optimized code are not new. This section briefly reviews the history of stencil compilers and discusses some of the more recent innovative strategies for optimizing stencil codes.

Special-purpose stencil compilers for distributed-memory machines first came into existence at least two decades ago [3, 4, 39]. The goal of these researchers was generally to reduce interprocessor data transfer and improve the performance of loop-based stencil computations through loop-level optimizations. The compilers expected the stencils to be expressed in some normalized form.

More recently, Krishnamoorthy *et al.* [28] have considered automatic parallelization of loop-based stencil codes through loop tiling, focusing on load-balancing the execution of the tiles. Kamil *et al.* [25] have explored automatic parallelization and tuning of stencil computations for chip multiprocessors. The stencils are specified using a domain-specific language which is a subset of Fortran 95. An abstract syntax tree is built from the stencil specified in the input language, from which multiple formats of output can be generated, including Fortran, C, and CUDA. The parallelization is based on blocked loops.

We have discussed Frigo and Strumpen's seminal trapezoidal-decomposition algorithms [16, 17] at length, since they form the foundation of the Pochoir algorithm. Nitsure [34] has studied how to use Frigo and Strumpen's parallel algorithm to implement 2D and 3D lattice Boltzmann methods. In addition to several other optimizations, Nitsure employs two code clones for the kernel to reduce the overhead of boundary checking, which Pochoir does as well. Nitsure's stencil code is parallelized with OpenMP [35], and data dependencies among subdomains are maintained by locking.

Cache-aware techniques have been used extensively to improve the stencil performance. Datta *et al.* [9] and Kamil *et al.* [26, 27] have applied both algorithmic and coding optimizations to loop-based stencil computations. Their algorithmic optimizations include an explicitly blocked time-skewing algorithm which overlaps subregions to improve parallelism at the cost of redundant memory storage and computation. Their coding optimizations include processor-affinity binding, kernel inlining, an explicit user stack, early cutoff, indirection instead of modulo, and autotuning.

Researchers at the University of Southern California [11, 12, 36] have performed extensive studies on how to improve the performance of high-order stencil computations though parallelization and optimization. Their techniques, which apply variously to multicore and cluster machines, include intranode, internode, and data-parallel optimizations, such as cache blocking, register blocking, manual SIMD-izing, and software prefetching.

6. CONCLUDING REMARKS

It is remarkable how complex a simple computation can be when performance is at stake. Parallelism and caching make stencil computations interesting. As discussed in Section 5, many researchers have investigated how various other features of modern machines — such as prefetching units, graphical processing units, and clustering — can be exploited to provide even more performance. We see many ways to improve Pochoir by taking advantage of these machine capabilities.

In addition, we see ample opportunity to enhance the linguistic features of the Pochoir specification language to provide more generality and flexibility to the user. For example, we are considering how to allow the user to specify irregularly shaped domains. As long as the boundary of a region, however irregular, is small compared to the region's interior, special-case code to handle the boundary should not adversely impact the overall performance. Even more challenging is coping with boundaries that change with time. We believe that such capabilities will dramatically speed up the PSA, RNA, and LCS benchmarks which operate on diamond-shaped space-time domains.

Pochoir's two-phase compilation strategy introduces a new method for building domain-specific languages embedded in C++. Historically, the complexity of parsing and type-checking C++ has impeded such separately compiled domain-specific languages. C++'s template programming does provide a good measure of expressiveness for describing special-purpose computations, but it provides no ability to perform the domain-specific optimizations such as those that Pochoir employs. Pochoir's compilation strategy offers a new way to build optimizing compilers for domain-specific languages embedded in C++ where the compiler can parse and "understand" only as much of the programmer's C++ code as it is able, confident that code it does not understand is nevertheless correct.

The Pochoir compiler can be downloaded from `http://supertech.csail.mit.edu/pochoir`.

7. ACKNOWLEDGMENTS

Thanks to Matteo Frigo of Axis Semiconductor and Volker Strumpen of the University of Linz, Austria, for providing us with their code for trapezoidal decomposition of the 2D heat equation which served as a model and inspiration for Pochoir. Thanks to Kaushik Datta of Reservoir Labs and Sam Williams of Lawrence Berkeley National Laboratory for providing us with the Berkeley autotuner code and help with running it. Thanks to Geoff Lowney of Intel for his support and critical appraisal of the system and to Robert Geva of Intel for an enormously helpful discussion that led to a great simplification of the Pochoir specification language. Many thanks to the Intel Cilk team for support during the development of Pochoir, and especially Will Leiserson for his responsiveness as the SPAA submission deadline approached. Thanks to

Will Hasenplaugh of Intel and to members of the MIT Supertech Research Group for helpful discussions.

8. REFERENCES

[1] T. Akutsu. Dynamic programming algorithms for RNA secondary structure prediction with pseudoknots. *Discrete Applied Mathematics*, 104:45–62, 2000.

[2] R. Bleck, C. Rooth, D. Hu, and L. T. Smith. Salinity-driven thermocline transients in a wind- and thermohaline-forced isopycnic coordinate model of the North Atlantic. *Journal of Physical Oceanography*, 22(12):1486–1505, 1992.

[3] R. G. Brickner, W. George, S. L. Johnsson, and A. Ruttenberg. A stencil compiler for the Connection Machine models CM-2/200. In *Workshop on Compilers for Parallel Computers*, 1993.

[4] M. Bromley, S. Heller, T. McNerney, and G. L. Steele Jr. Fortran at ten Gigaflops: The Connection Machine convolution compiler. In *PLDI*, pages 145–156, Toronto, Ontario, Canada, June 26–28 1991.

[5] C++ Standards Committee. Working draft, standard for programming language C++. available from `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf`, 2011. ISO/IEC Document Number N3242=11-0012.

[6] R. A. Chowdhury, H.-S. Le, and V. Ramachandran. Cache-oblivious dynamic programming for bioinformatics. *TCBB*, 7(3):495–510, July-Sept. 2010.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[8] K. Datta. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2009.

[9] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC*, pages 4:1–4:12, Austin, TX, Nov. 15–18 2008.

[10] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

[11] H. Dursun, K.-i. Nomura, L. Peng, R. Seymour, W. Wang, R. K. Kalia, A. Nakano, and P. Vashishta. A multilevel parallelization framework for high-order stencil computations. In *Euro-Par*, pages 642–653, Delft, The Netherlands, Aug. 25–28 2009.

[12] H. Dursun, K.-i. Nomura, W. Wang, M. Kunaseth, L. Peng, R. Seymour, R. K. Kalia, A. Nakano, and P. Vashishta. In-core optimization of high-order stencil computations. In *PDPTA*, pages 533–538, Las Vegas, NV, July13–16 2009.

[13] J. F. Epperson. *An Introduction to Numerical Methods and Analysis*. Wiley-Interscience, 2007.

[14] H. Feshbach and P. Morse. *Methods of Theoretical Physics*. Feshbach Publishing, 1981.

[15] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, pages 285–297, New York, NY, Oct. 17–19 1999.

[16] M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *ICS*, pages 361–366, Cambridge, MA, June 20–22 2005.

[17] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. *Theory of Computing Systems*, 45(2):203–233, 2009.

[18] M. Gardner. Mathematical Games. *Scientific American*, 223(4):120–123, 1970.

[19] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982.

[20] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview scalability analyzer. In *SPAA*, pages 145–156, Santorini, Greece, June 13–15 2010.

[21] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4), December 1996.

[22] Intel software autotuning tool. `http://software.intel.com/en-us/articles/intel-software-autotuning-tool/`, 2010.

[23] Intel Corporation. *Intel Cilk Plus Language Specification*, 2010. Document Number: 324396-001US. Available from `http://software.intel.com/sites/products/cilk-plus/cilk_plus_language_specification.pdf`.

[24] C. John. *Options, Futures, and Other Derivatives*. Prentice Hall, 2006.

[25] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *IPDPS*, pages 1–12, 2010.

[26] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *MSPC*, pages 51–60, San Jose, CA, 2006.

[27] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *MSP*, pages 36–43, Chicago, IL, June 12 2005.

[28] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *PLDI*, San Diego, CA, June 10–13 2007.

[29] `https://perf.wiki.kernel.org/index.php/Main_Page`.

[30] R. Mei, W. Shyy, D. Yu, and L. Luo. Lattice Boltzmann method for 3-D flows with curved boundary. *J. of Comput. Phys*, 161(2):680–699, 2000.

[31] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37:316–344, December 2005.

[32] P. Micikevicius. 3D finite difference computation on GPUs using CUDA. In *GPPGPU*, pages 79–84, Washington, DC, Mar. 8 2009.

[33] A. Nakano, R. Kalia, and P. Vashishta. Multiresolution molecular dynamics algorithm for realistic materials modeling on parallel computers. *Computer Physics Communications*, 83(2-3):197–214, 1994.

[34] A. Nitsure. Implementation and optimization of a cache oblivious lattice Boltzmann algorithm. Master's thesis, Institut für Informatic, Friedrich-Alexander-Universität Erlangen-Nürnberg, July 2006.

[35] OpenMP application program interface, version 2.5. OpenMP specification, May 2005.

[36] L. Peng, R. Seymour, K.-i. Nomura, R. K. Kalia, A. Nakano, P. Vashishta, A. Loddoch, M. Netzband, W. R. Volz, and C. C. Wong. High-order stencil computations on multicore clusters. In *IPDPS*, pages 1–11, Rome, Italy, May 23–29 2009.

[37] S. Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 1998.

[38] H. Prokop. Cache-oblivious algorithms. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

[39] G. Roth, J. Mellor-Crummey, K. Kennedy, and R. G. Brickner. Compiling stencils in High Performance Fortran. In *SC*, pages 1–20, San Jose, CA, Nov. 16–20 1997. ACM.

[40] A. Taflove and S. Hagness. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech House, Norwood, MA, 2000.

[41] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Lattice Boltzmann simulation optimization on leading multicore platforms. In *IPDPS*, pages 1–14, Miami, FL, Apr. 2008.

Hybrid strategy for stencil computations on the APU

Pacôme Eberhart Issam Said Pierre Fortin

UPMC Univ Paris 06 and CNRS UMR 7606, LIP6
4 place Jussieu, F-75252, Paris cedex 05, France
Contact : Pacome.Eberhart@lip6.fr

Henri Calandra
Total
Avenue Larribau, 64000 Pau,
France

ABSTRACT

Stencil computations are very regular and well adapted to GPU execution. However, the PCI-E bus that connects a discrete GPU to the system memory has a relatively low bandwidth when compared to the GPU compute power. The AMD APU architecture contains both CPU and GPU on the same chip and shared memory between them, which enables to bypass this PCI-E bus. In this paper, we devise a strategy for hybrid deployments on the CPU and the integrated GPU of the APU. For the task-parallel deployment, we rely on the CPU to process the diverging parts of the application. For the data-parallel deployment, we balance the workloads of the CPU and the GPU to achieve the best performance. Our strategy is tested on different stencil computations and we achieve a 20 to 30% gain in performance in the best cases.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles—*heterogeneous (hybrid) systems*; G.1.8 [**Numerical Analysis**]: Partial Differential Equations—*Finite difference methods*; G.4 [**Mathematical Software**]: Parallel and vector implementations

Keywords

APU, heterogeneous architecture, OpenCL programming, finite difference stencil, divergence, task parallelism, data parallelism

1. INTRODUCTION

In the industrial context of oil exploration for Total, the RTM (Reverse Time Migration [3]) allows to check for the presence of oil fields without the high cost of drilling. First, data is gathered in a prospect area by sending vibrations into the ground and capturing reflected waves. Then, by comparing simulated wave propagation and the captured data, the composition of underground layers is deduced.

Finite difference stencil computations are the core of the RTM, as well as of many other physics simulations. They approximate the derivation operators in physics equations into linear combinations on a discrete domain. Due to their regularity, they are very well suited for efficient executions on the massively parallel architecture of GPUs (Graphics Processing Units). Several works have thus studied the deployment of the RTM on GPUs [5, 15, 5]. However, when executed on a GPU, the RTM needs to bring data back from the GPU to the CPU. When this *snapshotting* gets very frequent, the slow PCI-E bus data transfer rate has a negative impact on snapshotting times [1], thus slowing down the overall RTM performance.

The AMD APU (*Accelerated Processing Unit*) has both CPU and GPU cores, and shared memory between them. This gives the APU an interesting potential for high performance parallel computing [9, 8]. Other works [6, 7] have shown the potential of APU integrated GPUs for the RTM, especially when considering energy consumption. The memory shared between CPU and GPU on the APU allows indeed to completely bypass the PCI-E bus for the snapshotting.

In this paper, we will try to use both the CPU and the GPU of the APU to gain further performance. The partial SIMD (Single Instruction Multiple Data) execution of a GPU will force diverging control flows to be serialized (compute divergence). Similarly, the cost of memory divergence on a GPU is due to the fact that memory accesses are optimal only when contiguous and aligned. As the costs of computation and memory divergence are lower on CPU, specific treatments on the borders of the domain could be executed on the CPU, while keeping the regular execution within the domain on the GPU. We will call this deployment task-parallel. The CPU could also simply be used for its additional compute power and memory bandwidth in a data-parallel deployment. To decide between those two alternatives, we devise a strategy for hybrid deployments on the APU.

Using a single parametrizable OpenCL source code, we will first select the best performing version for each architecture. We will then detail how to ensure efficient hybrid deployments on the APU according to our strategy.

Section 2 will provide an overview of the APU architecture and of the OpenCL model. Section 3 explains our strat-

HiStencils 2014
First International Workshop on High-Performance Stencil Computations
January 21, 2014, Vienna, Austria
In conjunction with HiPEAC 2014.

http://www.exastencils.org/histencils/2014/

egy for hybrid deployments on the APU. In Section 4, we study several strategies for deploying stencil computations on only the CPU or only the GPU, in order to prepare for hybrid deployments. In Section 5, we deploy hybrid stencil computations on the APU and present performance results. Section 6 concludes and discusses future work.

2. APU ARCHITECTURE AND OPENCL PROGRAMMING

The APU A10-5700 (family codename Trinity) we have at our disposal has 4 CPU cores at a 3400 MHz frequency with SSE instructions and 4 MB L2 caches. The integrated GPU has 96 processing elements at a 760 MHz frequency and has no cache. Each of these processing elements can compute four 32 bit floating operations at the same time. The GPU has a maximum theoretical peak performance of 546 GFlop/s and the CPU has 108.8 GFlop/s. As such 83% of the compute power of the APU is contained in the GPU and 17% in the CPU.

The memory of the APU is not entirely shared between the CPU and the GPU. The sytem memory and the GPU memory are only accessible from, respectively, the CPU and the GPU, and can each contain shared locations (resp. device-visible host memory and host-visible device memory) (see Fig. 1). Accessing the memory from the CPU is done through L2 caches or Write Combine (WC) buffers. There are two memory buses for the GPU: *Garlic*, fast (maximum theoretical bandwidth of 25.6 GB/s) but without a coherence protocol with CPU caches, and *Onion*, slower (8 GB/s) but cache coherent. We classify the different locations in memory for buffers according to [6] as:

- c, the system memory, accessible from the CPU only;
- g, the GPU memory, not accessible from the CPU, accessed by the *Garlic* bus;
- p, the GPU memory shared with the CPU;
- u, the CPU memory accessible from the GPU by the *Garlic* bus;
- z, the CPU memory accessible from the GPU by the *Onion* bus.

p, u and z are refered to as *zero-copy* memory buffers. *Garlic* having no coherence protocol with CPU caches, u is read-only for the GPU. The WC buffers on the CPU ensure that writes from the CPU to u are propagated to system memory and visible from the GPU. p and z are read-write enabled locations for both CPU and GPU.

In order to use the CPU and the GPU concurrently, we use OpenCL [11], as OpenCL kernels can be executed on CPU and on GPU, and synchronized through the runtime. Using the SPMD (Single Program, Multiple Data) programming model, the kernel is executed independantly on multiple work-groups, each made of work-items that are executed on hardware threads. On an AMD OpenCL GPU, work-items of a given work-group are executed in several *wavefronts* (similar to *warps* in NVIDIA CUDA), each one being processed synchronously (SIMD). Wavefronts in which work-items execute different control flows will have their executions serialized over the different control flows. On an AMD OpenCL CPU, a thread will execute one by one each

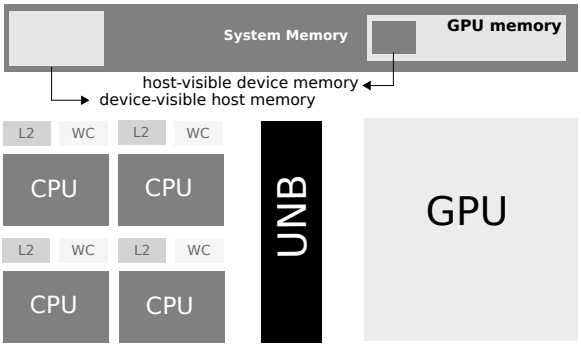


Figure 1: Architecture of the APU (UNB : Unified North Bridge)

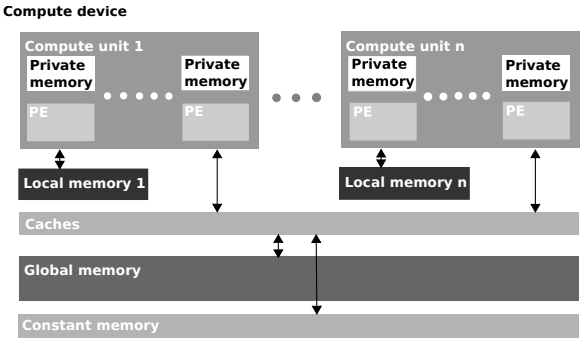


Figure 2: OpenCL memory model

work-item of a given work-group until completion or encountering a barrier. As the work-items are executed one at time, there is no overhead on CPU due to diverging control flows among the work-items.

Here, it has to be noticed that the current AMD OpenCL SDK does not provide implicit vectorization among multiple work-items on CPU. To use the SSE instructions on the CPU, and the vector instructions on the GPU, we therefore need to rely on explicit vectorization by means of vector types (`float4`).

The memory model of OpenCL allows different levels of sharing between work-items (see Fig. 2). The global memory is accessible by all work-items, the local memory is shared within a work-group and registers belong to a single work-item. On a GPU, global memory accesses are optimal when work-items in a given wavefront access a contiguous and aligned memory area at the same time (*coalesced* memory accesses on NVIDIA GPUs). Since the work-items are processed one by one on a CPU, the CPU is also not sensitive to the divergence in memory accesses among multiple work-items. Moreover the CPU caches enable to offset the overhead of irregular memory accesses. Besides, local memory is redundant on CPU with the caches of each core, especially since a work-group is executed on a single core.

An OpenCL program is launched from a host machine. Kernels are compiled for the chosen hardware, and placed in an execution queue for that hardware. Data transfers and barriers can also be enqueued. Finally, the host will synchronize with these queues to ensure that an execution is over.

3. HYBRID STRATEGY FOR THE APU

In stencil computations, the borders of the domain may require specific memory access patterns and may induce compute divergence. This will cause an increase in the cost of computation on a GPU. We here propose the use of the APU CPU for the borders, where the computation will not be serialized. We call this deployment task-parallel. We can also divide the domain into two regions to be executed on the CPU and the GPU, in a data-parallel fashion (see Fig. 3).

To choose between these two alternatives, we propose the strategy presented in Fig. 4. First, we determine if the application is compute-bound or memory-bound. For this purpose, we compare the arithmetic intensity (ratio of the number of memory accesses over the number of arithmetic operations) and the hardware specifications of the integrated GPU. We then try to quantify the divergence, either through a profiler, or through code analysis or even thanks to programmer indications. For a memory-bound application, we only consider divergent memory accesses, whereas we look only at divergent computations in the compute-bound case. If the divergence is high enough according to empirical thresholds, we choose the task-parallel deployment. Otherwise, the data-parallel one will be preferred.

Such strategy could thus be implemented in a generic software platform. In this paper, we will only apply and try to validate this strategy on our stencil computations.

The hybrid strategy applied to stencil computations requires to read from an input buffer and write to an output buffer, for both CPU and GPU. As the buffers will swap between input and output at every iteration, buffers need to be zero-copy and read-write enabled. u memory, being read-only from the GPU, is not a valid choice, only z and p are both zero-copy and read-write enabled from CPU and GPU. Previous work [6] has shown p to be significantly slower than z for CPU reads. The input and output buffers will thus be allocated as z buffers.

Also shown in [6] is the fact that z memory is only more interesting than g memory (along with explicit data copies) when snapshotting is performed at every iteration in the RTM. Memory accesses via *Garlic* (g memory) are indeed much more efficient than via *Onion* (z memory) on current APUs. In the future APU architectures, this performance gap between z and g memory locations should however be reduced, which will widen the interest of using z memory and justify our hybrid approach based on zero-copy buffers.

Before we try to apply this strategy, we study different deployments on CPU-only and GPU-only computations. We will determine which ones are the most efficient and use them as the basis for our task-parallel and data-parallel deployments.

4. STENCIL COMPUTATIONS ON CPU OR GPU

When simulating infinite or semi-infinite domains with stencils, the borders act as reflectors. To correct this, PMLs (Perfectly Matched Layers) [4] are usually used on the borders. This technique creates divergence in the computation on the borders of the domain.

To study the effect of memory access divergence, we use here a basic stencil with a parametrizable size. This enables us to control the amount of divergence by changing its size.

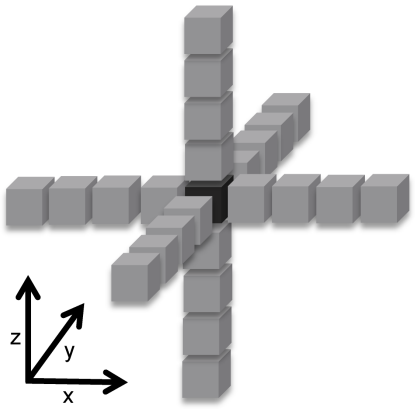


Figure 5: Memory access pattern of an 8th order (size 4) laplacian stencil

To ensure the best control over divergence we therefore do not use PMLs here. We do not use halos either (additional points on the borders of the domain containing only zero values) on the buffers, as they would cancel memory divergence. We will use a laplacian stencil (see Fig. 5), as used in the RTM. For low divergence, we will use a size 4 stencil, and a size 32 for high divergence.

We also examine different deployment strategies following [14]:

- **complete**, one single kernel executed on the 3D whole domain;
- **inout**, one kernel executed on the inside of the domain and another one encompassing all the borders;
- **sides**, one kernel executed on the inside of the domain and one for each of the six borders in 3D.

complete will be the basis for the data-parallel deployment, and the task-parallel deployment will be developed on the most performant between **inout** and **sides**.

The kernels have different versions to exploit at best the different hardware specifications of the CPU and of the GPU. A **scalar** version will allocate work-items on a 2D grid where each of them will iterate along the Z-axis of the 3D domain. The **vectorized** version adds the explicit use of OpenCL vector types (float4) and instructions to compute 4 points at the same time. The **local vectorized** uses the local memory of the work-groups to share the values of points on an XY-plan [13, 12]: this makes the execution benefit from the higher bandwidth of the local memory on common values needed by several work-items.

It has to be noticed that our kernels source codes are fully parametrizable. This enables us to define the size of the stencil and the size of the work-group at kernel compilation time by using the C pre-processor. We can then easily optimize the size of the work-groups for the A10-5700 APU model via an exhaustive search. The domains are three-dimensional and of size N^3.

Due to the existence of vectorized instructions on both the CPU and GPU, the **vectorized** kernel performs consistently better than the **scalar** one thanks to the SSE instructions and the GPU vector processing units (performance tests not presented). The **local vectorized** version does not offer performance gain over **vectorized** either. On the CPU, the

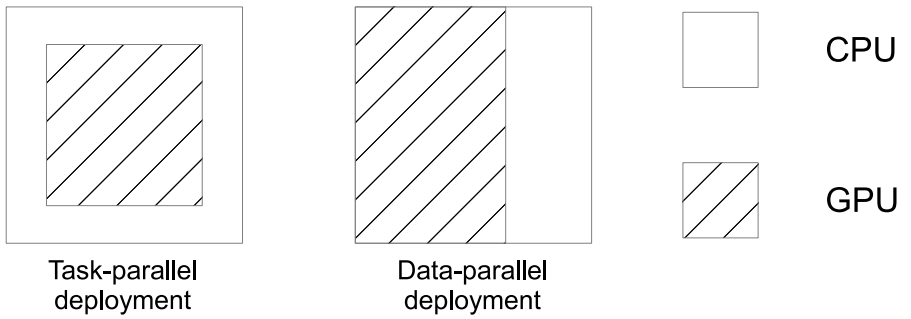


Figure 3: Example of data-parallel and task-parallel deployments on a 2D domain

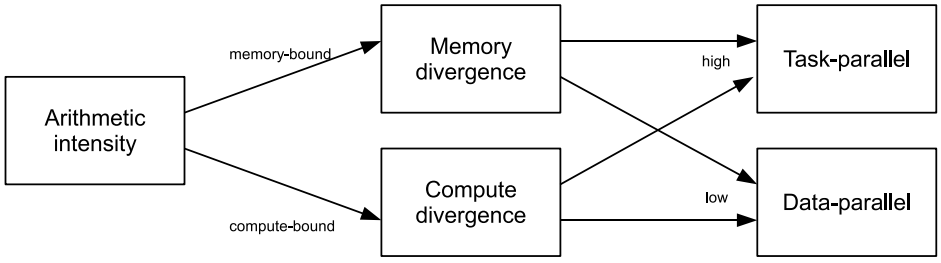


Figure 4: Hybrid strategy for the APU

redundancy of local memory with caches implies indeed an overhead. On the GPU, there is also no performance gain for **local vectorized** with a stencil of size 4. For a stencil of size 32, the amount of local memory required decreases the occupancy, hence the low performance. We will therefore consider only the **vectorized** kernel through the rest of this paper.

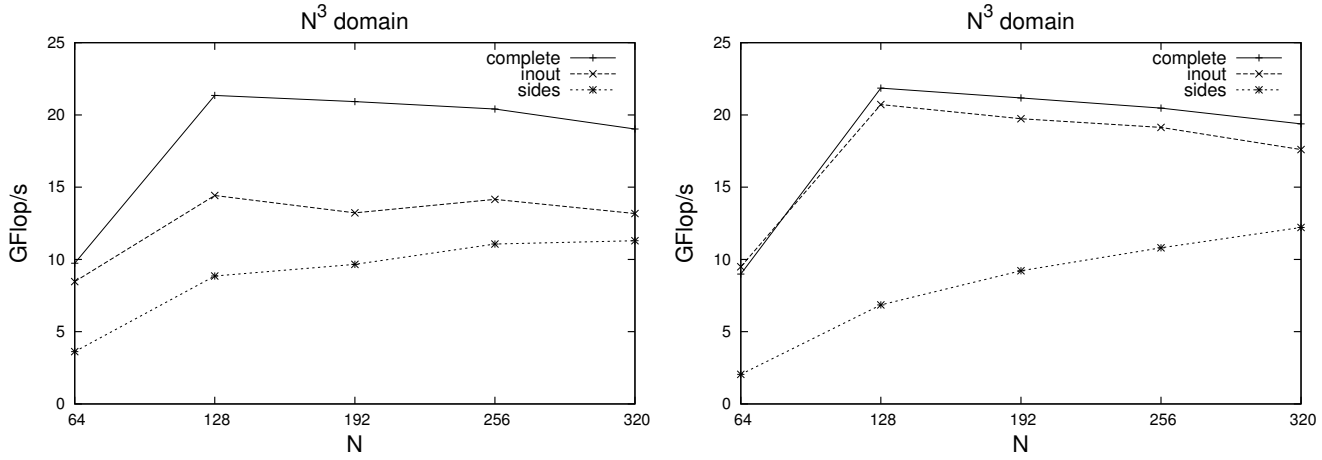
Figures 6a, 6b, 7a and 7b, present performance results of our different deployments on GPU only and on CPU only. This enables us to study the divergence impact on each architecture separately.

On GPU, **sides** is topped by the other strategies, due to the cost of launching the multiple kernels and the low occupancy of the kernels on the borders. On a size 4 stencil (see Fig. 6a), **complete** performs better than **inout**. When the size of the stencil is increased to 32 (see Fig. 6b), memory divergence rises and **inout** almost catches up with **complete** but does not outperform it: this is due to the fact that the AMD GPUs are only slightly sensitive to memory divergence ([2], section 6.4).

On CPU, performance drops very fast when the domain enlarges (see Figs. 7a and 7b), because a smaller domain, fitting almost entirely in cache, allows for a better memory bandwidth. There is nearly no difference between the performances of the three strategies, as they were designed to handle divergence and CPU hardware is not sensitive to divergence. Domains with sizes which are multiples of 128 offer low performance, probably due to the under-utilization of the interleaved memory banks. The effects are especially noticeable for the size 32 stencil computations, which are subject to higher cache pressure.

We have observed so far the effect of memory divergence on stencil computations, but the little sensivity of AMD hardware to memory divergence has lessened its impact. To more clearly witness the impact of our strategy, we have also studied artificial compute-bound stencils with compute divergence. In this purpose, we first include compute divergence in our kernels by dividing the computation into seven distinct parts: the inside of the domain, and each of the six sides. To ensure that the compute divergence will not be made negligible by the memory-bound nature of our stencils, we artificially raise the arithmetic intensity. To do so, we compute each point of the domain several times and accumulate all the results, while keeping the values necessary for the computation in registers.

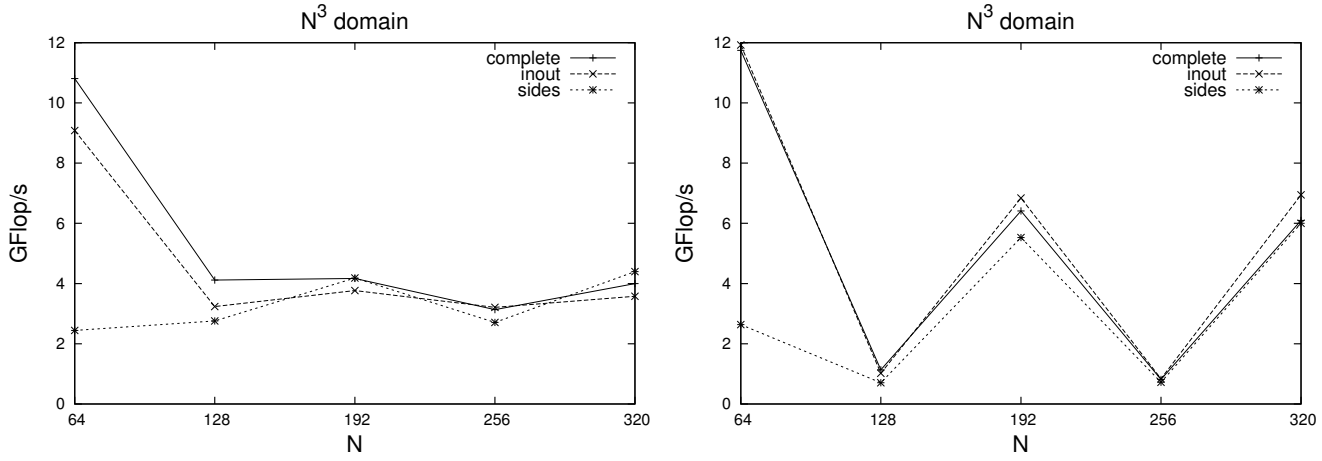
In the memory-bound stencil, for each point in our domain, there were 26 memory accesses (25 reads and one write) for 36 arithmetic operations. For the compute-bound stencil, the number of memory accesses remains the same, whereas the number of arithmetic operations is multiplied by the number of iterations. The APU integrated GPU has



(a) Size 4 stencil

(b) Size 32 stencil

Figure 6: Performance results of stencil computations on the GPU



(a) Size 4 stencil

(b) Size 32 stencil

Figure 7: Performance results of stencil computations on the CPU

a maximum memory bandwidth of 25.6 GB/s, meaning it can access 6.4G floats per second. With a peak performance at 546 GFlop/s, its theoretical threshold between memory-bound and compute-bound applications is around 85 floating point operations per memory access. The arithmetic intensity of our previous basic stencil computation was 1.4, which is clearly memory-bound. With 128 iterations our new artificial stencil computation has an arithmetic intensity of 177 which is clearly compute-bound.

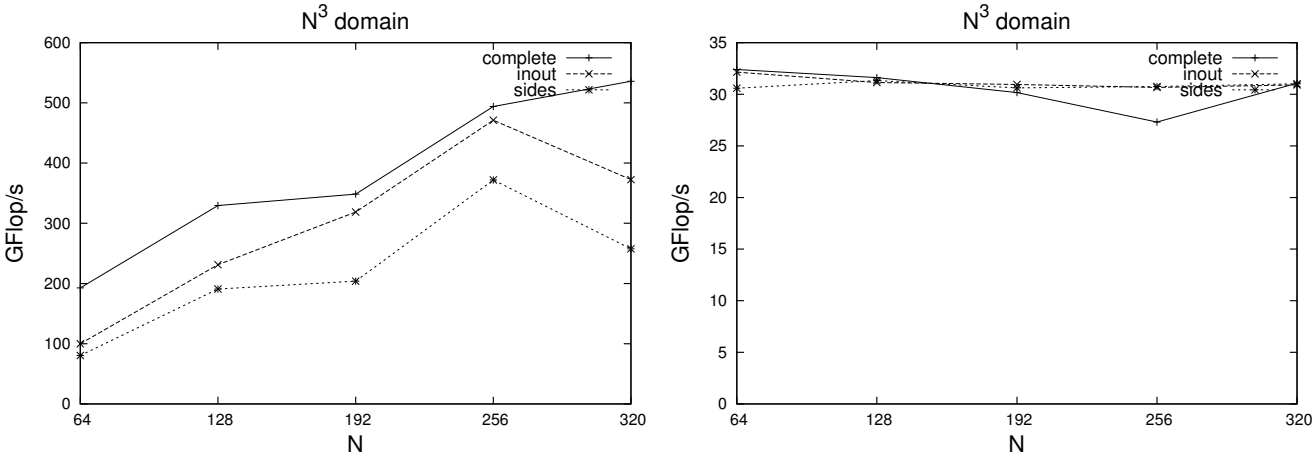
For these compute-bound stencils, we also study our three deployment strategies and we set the stencil size to 4. The relative performance of the three strategies are then similar to the previous ones on both the GPU and the CPU (see Figs. 8a and 8b). On the GPU, *sides* is again performing worse than *inout*, *inout* being closer to *complete*. On the CPU, there is no difference in the performance results of the three strategies, since the CPU is again not sensitive to divergence.

5. HYBRID DEPLOYMENT

Our two strategies for hybrid deployments, task-parallel and data-parallel, will execute one or several kernels on both CPU and GPU. For the task-parallel deployment, we choose the **inout** strategy, as it consistently performs better than **sides**. We modify it to make the CPU execute a divergent kernel on the borders of the domain, while the GPU executes a regular kernel on the inside. For the data-parallel one, we divide the domain into two subdomains and execute the same kernel on them. We divide the domain along the Y axis, as cutting along the X axis could split up `float4` values, and as the Z axis is the axis work-items iterate along.

We synchronize the execution of the kernels by using a blocking function, `clFinish`, to ensure that the two execution queues (CPU and GPU) are completed. According to the OpenCL specification [11], `clFinish` also ensures that memory writes are visible to both the CPU and GPU.

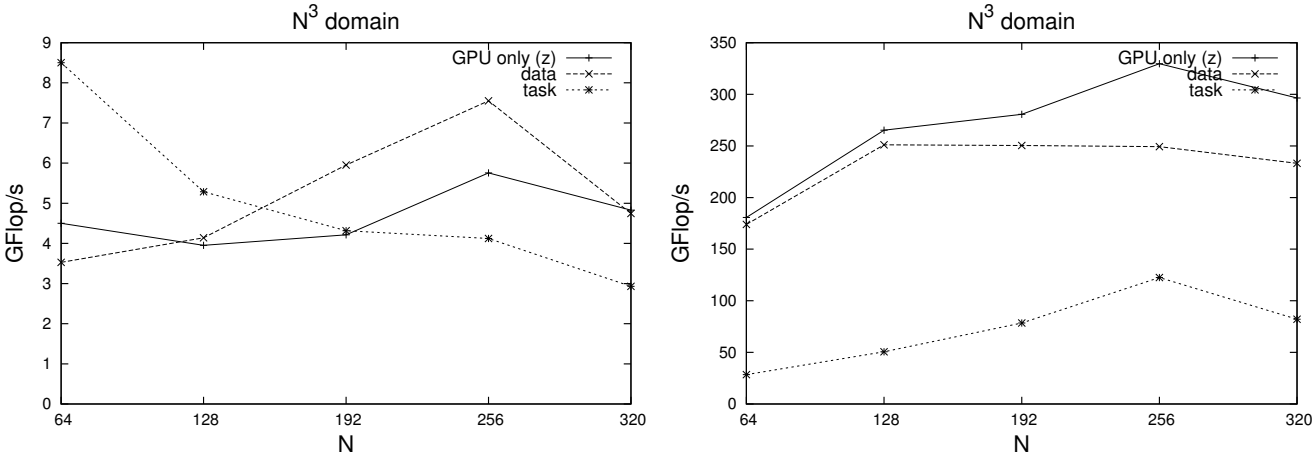
As noted by [10], the OpenCL standard does not specify a way to share a buffer for concurrent accesses by multiple devices (on distinct data within this buffer). However,



(a) On the GPU

(b) On the CPU

Figure 8: Performance results of compute-bound stencil computations (size 4)



(a) Memory-bound stencil

(b) Compute-bound stencil

Figure 9: Performance results of hybrid deployments of stencil computations

the AMD implementation of OpenCL makes it possible by not deleting the reference to the physical location of the buffer when unmapping. In their case, simply unmapping the buffer from the device allowed them to use it simultaneously on the GPU, as a device, and on the CPU, as a host. In our case, we similarly write to the shared buffer concurrently by the CPU and the GPU, both used here as OpenCL devices, and we have consistently checked the correctness of our hybrid computations.

For the data-parallel deployment, we also need to determine the optimal ratio between the parts of the domain that will be computed by the CPU and by the GPU. At first, we had chosen the theoretical peak performance ratio of the CPU and the GPU, but the use of the profiler has shown an imbalance in the execution times. We have then chosen the ratio of the actual performance of the CPU-only and of the GPU-only vectorized (complete) kernels: these have been confirmed empirically as nearly optimal, by running several performance tests with various ratios. For the memory-bound case, we have thus a 0.8 ratio for the GPU, and 0.95 in the compute-bound case.

According to the AMD profiler, executing a kernel on the CPU used as a device preempts the CPU and hence prevents the CPU host from enqueuing GPU kernel executions. A possible solution would have been the OpenCL device fission of the CPU, separating it into different devices, keeping one core for the host and executing the kernel on the remaining three. In our case, enqueuing the GPU kernel first was adequate, as we have to synchronize at each iteration and cannot enqueue more than one GPU kernel at a time.

We now compare the APU hybrid performance (for stencils of size 4) with an execution on the APU integrated GPU only with buffers allocated in z memory.

For the memory-bound case (see Fig. 9a), we see a better performance in the task-parallel deployment on smaller domains, but this drops when the domain gets larger and the GPU occupancy rises. When the GPU is full enough, the CPU will not perform fast enough to keep up, and overall performance will decrease. The data-parallel deployment performs better when the domain gets larger, as the relative cost of synchronization gets lower when computation

time increases. We obtain a 20 to 30% better performance compared to GPU only (with buffers in z memory).

For the compute-bound case (see Fig. 9b), the task-parallel deployment is slowed down by the CPU's lower compute power. Even for the data-parallel deployment, the cost of synchronization is too high due to the imbalance in the ratio between the CPU and the GPU. This may be due to the lack of optimization of the source code for the CPU: whereas its compute power is 17% of the APU, its kernel performance is only 5% of that of the integrated GPU.

Finally, it has to be noticed that the GPU-only performance (with buffers in z memory) is here lower than the GPU-only performance (with buffers in g memory) presented in Figs. 6a and 8a. g memory offers indeed a better bandwidth than z memory on current APUs. However, those previous results with g memory do not take into account the cost of the snapshotting necessary to applications such as the RTM. In [6], g memory has been shown to perform currently better than z memory even when the snapshotting frequency is high, but future generations of APUs will significantly improve the bandwidth of such zero-copy (z) memory.

6. CONCLUSIONS

In this paper, we studied how to use both the CPU and the GPU of the APU for stencil computations. We provided a strategy for hybrid deployments that takes into account the compute-bound or memory-bound nature of the application and its amount of divergence. We proposed two possible deployments, a task-parallel one and a data-parallel one, and balanced the use of the CPU and the integrated GPU in order to exploit at best the APU. When considering a classic memory-bound stencil, we obtained an up to 30% performance gain compared to a GPU only deployment.

Our strategy seems to be valid in the memory-bound case, but requires further investigation for the compute-bound case on the APU. More precisely, we are currently developping a specific OpenCL kernel for the CPU: this kernel should better exploit the CPU caches as well as the CPU *prefetch* feature, and hence offer much better performance. In the future, we will first try to confirm the validity of our strategy on the complete RTM, as well as on various applications. We also plan to apply this strategy on other hybrid hardware, such as the Intel multicore CPUs with integrated GPUs or the future Denver architecture from NVIDIA with both GPU and ARM CPU on the same chip.

7. ACKNOWLEDGEMENT

The authors would like to thank Total for funding this work, as well as AMD for providing us with the surveyed hardware.

8. REFERENCES

[1] R. Abdelkhalek, H. Calandra, O. Coulaud, J. Roman, and G. Latu. Fast seismic modeling and reverse time migration on a GPU cluster. In *International Conference on High Performance Computing Simulation, 2009. HPCS '09*, pages 36–43, 2009.

[2] AMD. Accelerated parallel processing OpenCL programming guide, May 2012.

[3] E. Baysal, D. D. Kosloff, and J. W. C. Sherwood. Reverse time migration. *Geophysics*, 48(11):1514–1524, 1983.

[4] J.-P. Berenger. A perfectly matched layer for the absorption of electromagnetic waves. *Journal of Computational Physics*, 114(2):185–200, 1994.

[5] J. Cabezas, M. Araya-Polo, I. Gelado, N. Navarro, E. Morancho, and J. Cela. High-performance reverse time migration on gpu. In *Chilean Computer Science Society (SCCC), 2009 International Conference of the*, pages 77–86, 2009.

[6] H. Calandra, R. Dolbeau, P. Fortin, J.-L. Lamotte, and I. Said. Evaluation of successive CPUs/APUs/GPUs based on an OpenCL finite difference stencil. In *2013 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 405–409, 2013.

[7] H. Calandra, R. Dolbeau, P. Fortin, J.-L. Lamotte, and I. Said. Forward seismic modeling on AMD accelerated processing unit. In *Rice Oil & Gas HPC Workshop*, 2013.

[8] G. Cocco and A. Cisternino. Device specialization in heterogeneous multi-GPU environments. In *2012 Imperial College Computing Student Workshop*, pages 35–41, 2012.

[9] M. Daga, A. M. Aji, and W. chun Feng. On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing. In *Symposium on Application Accelerators in High-Performance Computing*, 2011.

[10] M. C. Delorme, T. S. Abdelrahman, and C. Zhao. Parallel radix sort on the AMD fusion accelerated processing unit. In *International Conference on Parallel Processing, 2013. ICPP 2013*, pages 339–348, 2013.

[11] Khronos Group. The OpenCL Specification version 1.2, November 2012.

[12] D. Michéa and D. Komatitsch. Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards. 182(1):389–402, 2010.

[13] P. Micikevicius. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, page 79–84, New York, NY, USA, 2009. ACM.

[14] J. I. Toivanen, T. P. Stefanski, N. Kuster, and N. Chavannes. Comparison of CPML implementations for the GPU-accelerated FDTD solver. *Progress in Electromagnetics Research M*, 19:61–75, 2011.

[15] K. Yoon, S. Suh, J. Ji, J. Cai, and B. Wang. Stability and speedup issues in TTI RTM implementation. In *SEG Technical Program Expanded Abstracts 2010*, pages 3221–3225, 2010.

Toward efficient distribution of MPDATA stencil computation on Intel MIC architecture

Lukasz Szustak
Czestochowa University of Technology
lszustak@icis.pcz.pl

Krzysztof Rojek
Czestochowa University of Technology
krojek@icis.pcz.pl

Roman Wyrzykowski
Czestochowa University of Technology
roman@icis.pcz.pl

Pawel Gepner
Intel Corporation
pawel.gepner@intel.com

ABSTRACT

The multidimensional positive definite advection transport algorithm (MPDATA) belongs to the group of nonoscillatory forward-in-time algorithms, and performs a sequence of stencil computations. MPDATA is one of the major parts of the dynamic core of the EULAG geophysical model.

The Intel Xeon Phi coprocessor is the first product based on the Intel Many Integrated Core (Intel MIC) architecture. This architecture offers notable performance advantages over traditional processors, and supports practically the same traditional parallel programming model.

In this work, we outline an approach to adaptation of the 3D MPDATA algorithm to the Intel MIC architecture. This approach is based on combination of temporal and space blocking techniques, and allows us to ease memory and communication bounds and better exploit the theoretical floating point efficiency of target computing platforms. In order to utilize computing resources available in Intel Xeon Phi, the proposed approach employs two main levels of parallelism: (i) task parallelism which allows for utilization of more than 200 logical cores, and (ii) data parallelism to use efficiently 512-bit vector processing units.

An important method of improving the efficiency of the block decomposition is partitioning of available cores/threads into teams. It allows us to reduce inter-cache communication overheads. Also, this method increases opportunities for the efficient distribution of MPDATA computation onto available resources. The purpose is to provide the trade-off between two coupled criteria: load balancing and intra-cache communication.

We discuss performance results obtained on two platforms, including either two Intel Xeon E5-2643 CPUs and Intel Xeon Phi 3120A, or two Intel Xeon E5-2697 v2 CPUs and Intel Xeon Phi7120P. The top-of-the-line Intel Xeon Phi 7120P gives the best performance results for all tests.

The achieved performance results provide a basis for further research on optimizing the distribution of MPDATA computations across all the computing resources of the Intel MIC architecture, taking into consideration features of its on-board memory, cache hierarchy, computing cores, and vector units.

Keywords

Stencil computation, MPDATA algorithm, temporal/space blocking techniques, Intel MIC, OpenMP, task scheduler

1. INTRODUCTION

MPDATA [7] is one of the two major parts of the dynamic core of the EULAG model. EULAG (Eulerian/semi-Lagrangian fluid solver) is an established geophysical model for simulating thermo-fluid flows across a wide range of scales and physical scenarios, including the numerical weather prediction (NWP).

The newest research of EULAG parallelization have been carried out using IBM Blue Gene/Q and CRAY XE6 [4]. 3D MPI parallelization has been used for running EULAG on these systems with tens of thousands of cores, or even with more than 100K cores. When parallelizing EULAG computation on supercomputers and CPU clusters, the efficiency is declined below 10%. We propose to rewrite the EULAG dynamical core and replace standard HPC systems by small heterogeneous clusters with accelerators such as GPU [5] and Intel MIC [3].

Preliminary studies of porting anelastic numerical models to modern architectures, including hybrid CPU-GPU architectures, were carried out in works [5, 11, 10]. The results achieved for porting selected parts of EULAG to non-traditional architectures revealed potential in running scientific applications, including anelastic numerical models, on novel hardware architectures.

In this work, we outline an approach to adaptation of the 3D MPDATA algorithm to the Intel MIC architecture. This approach is based on combination of temporal and space blocking techniques, and allows us to ease memory and communication bounds, and better exploit the theoretical floating point efficiency of target computing platforms. We show some of the optimization methods that we found effective, and demonstrate their impact on the performance of both the Intel CPU and MIC architectures. We mainly focus on the use of MPDATA in NWP, where the size of grids is limited by $n \leq 2048$, $m \leq 1024$, and $l = [64, 128]$. The starting

HiStencils 2014
First International Workshop on High-Performance Stencil Computations
January 21, 2014, Vienna, Austria
In conjunction with HiPEAC 2014.

http://www.exastencils.org/histencils/2014/

point in these research is an unoptimized parallel implementation of the MPDATA algorithm. In our work, we use the OpenMP standard for multi-/many-core programming.

2. ARCHITECTURE OVERVIEW

2.1 Intel MIC architecture

The Intel MIC architecture combines many Intel CPU cores onto a single chip [2, 3]. The Intel Xeon Phi coprocessor is the first product based on this architecture. The main advantage of these accelerators is that it is built to provide a general-purpose programming environment similar to that provided for Intel CPUs. This coprocessor is capable of running applications written in industry-standard programming languages such as Fortran, C, and C++.

The Intel Xeon Phi coprocessor includes processing cores, caches, memory controllers, PCIe client logic, and a very high bandwidth, bidirectional ring interconnect [3]. Each coprocessor contains of more than 50 cores clocked at 1 GHz or more. These cores support four-way hyper-threading, which gives more than 200 logical cores. The actual number of cores depends on the generation and model of a specific coprocessor. Each core features an in-order, dual-issue x86 pipeline, 32 KB of L1 data cache, and 512 KB of L2 cache that is kept fully coherent by a global-distributed tag directory. As a result, the aggregate size of L2 caches can exceeds 25 MB. The memory controllers and the PCIe client logic provide a direct interface to the GDDR5 memory on the coprocessor and the PCIe bus, respectively. The coprocessor has over 6 GB of on-board memory (maximum 16 GB). The high-speed bidirectional ring connects together all the cores, caches, memory controllers and PCIe client logic of Intel Xeon Phi coprocessors.

An important component of each Intel Xeon Phi processing core is its vector processing unit (VPU) [2], that significantly increases the computing power. Each VPU supports a new 512-bit SIMD instruction set called Intel Initial Many-Core Instructions. The new ability to work with 512-bit vectors enables operating on 16 float or 8 double elements per iteration, instead of a single element.

The Intel Phi coprocessor is delivered in form factor of a PCI express device, and cannot be used as a stand-alone processor. Since the Intel Xeon Phi coprocessor runs Linux operating system, any user can access the coprocessor as a network node, and directly run individual applications in the native mode. These coprocessors also support heterogeneous applications wherein a part of the application is executed on the host (CPU), while another part is executed on the coprocessor (offload mode).

2.2 Target platforms

In this study, we use two platforms containing a single Intel Xeon Phi coprocessor. The first platform is equipped with two newest Intel Xeon E5-2697 v2 CPUs (totally 2×2 cores), based on the Ivy Bridge architecture, and the Intel Xeon Phi 3120A card (57 cores). The second one includes two Sandy Bridge-EP Intel Xeon E5-2643 CPUs (2×4 cores in total), and the top-of-the-line Intel Xeon Phi 7120P coprocessor (61 cores). The peak performances of these platforms are 1521 (2x259 + 1003) GFlop/s and 1419 (2x105.5 + 1208) GFlop/s. These values are given for the double precision arithmetic, with taking into account the usage of SIMD vectorization. The important feature of Intel Xeon Phi coprocessors is the high memory bandwidth. In particular, Intel Xeon Phi 7120P provides 352 GB/s of memory bandwidth, as compared with 2×51.2 GB/s for both CPU platforms.

A summary of key features of tested platforms can be found in [1].

3. OUTLINE OF MPDATA

MPDATA belongs to the group of nonoscillatory forward-in-time algorithms, and performs a sequence of stencil computations. The 3D MPDATA algorithm is based on the first-order-accurate advection equation:

$$\frac{\partial \Psi}{\partial t} = -\frac{\partial}{\partial x}(u\Psi) - \frac{\partial}{\partial y}(v\Psi) - \frac{\partial}{\partial z}(w\Psi), \tag{1}$$

where x, y and z are space coordinates, t is time, $u, v, w = const$ are flow velocities, and Ψ is a nonnegative scalar field. Eqn. (1) is approximated according to the donor-cell scheme, which for the $(n+1)$-th time step $(n = 0, 1, 2, \ldots)$ gives the following equation:

$$\begin{aligned} \Psi^*_{i,j,k} = \Psi^n_{i,j,k} \quad - \quad & [F(\Psi^n_{i,j,k}, \Psi^n_{i+1,j,k}, U_{i+1/2,j,k}) - \\ & F(\Psi^n_{i-1,j,k}, \Psi^n_{i,j,k}, U_{i-1/2,j,k})] - \\ & [F(\Psi^n_{i,j,k}, \Psi^n_{i,j+1,k}, V_{i,j+1/2,k}) - \\ & F(\Psi^n_{i,j-1,k}, \Psi^n_{i,j,k}, V_{i,j-1/2,k})] - \\ & [F(\Psi^n_{i,j,k}, \Psi^n_{i,j,k+1}, W_{i,j,k+1/2}) - \\ & F(\Psi^n_{i,j,k-1}, \Psi^n_{i,j,k}, W_{i,j,k-1/2})]. \end{aligned} \tag{2}$$

$$U \equiv \frac{u\delta t}{\delta x};\ [U]^+ \equiv 0,5(U + |U|);\ [U]^- \equiv 0,5(U - |U|). \tag{3}$$

The same definition is true for the local Courant numbers V and W.

The first-order-accurate advection equation is approximated to the second order in δx, δy and δt, through defining the advection-diffusion equation. Such a transformation is widely described in the literature. For the full description of the most important aspects of the second-order-accurate formulation of MPDATA, the reader is referred to [6, 7].

The whole MPDATA computation in each time step are decomposed into a set of 17 stencil sweeps, called further stages. Each stage is responsible for calculating elements of a certain matrix, based on the corresponding stencil. The stages dependent on each other. Results of stages are usually input data for the next one.

A single MPDATA time step requires 5 input and 1 output matrices, other 16 matrices are temporary ones. In the basic, unoptimized implementation of the MPDATA algorithm, every stage reads a required set of matrices from the main memory, and writes results to the main memory after computation. This scheme is repeated for all the stages.

In consequence, a significant traffic to the main memory is generated. Moreover, compute units (cores/threads, and VPUs) have to wait for data transfers from the main memory to the cache hierarchy. In order to better utilize features of novel accelerators, the adaptation of MPDATA computation to the Intel MIC architecture is considered in this work, taking into account the memory-bounded character of the algorithm.

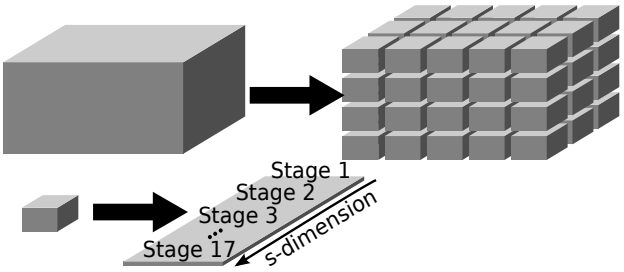


Figure 1: Idea of block decomposition of MPDATA computation

4. BLOCK DECOMPOSITION OF MPDATA

4.1 Basic idea

Since the 3D MPDATA algorithm includes so many intermediate computation, one of the primary methods for reducing the saturation of memory traffic is to avoid data transfers associated with these computation. For this aim, all the intermediate results must be kept in the cache memory, which increases the cache reusing. The memory traffic is generated only to transfer the required input and output data. Such an approach is commonly called the temporal blocking [8, 9].

In order to implement this approach efficiently, the loop tiling technique is applied. As a result, the grid is partitioned into blocks. Every block is responsible for computing all the 17 stages within the part of grid assigned to it. Within a single block, each stage provides computation for the adequate chunk of the corresponding matrix. Executing of a sequence of blocks determines the final output result for a single MPDATA time step.

The main requirement for this approach is to keep in the cache hierarchy all the data required for MPDATA computation within each block. Therefore, the size $nB \times mB \times lB$ of each block has to be selected in an appropriate way. The idea of block decomposition of the MPDATA algorithm is shown in Fig. 1. This decomposition determines four dimensions of distribution of MPDATA calculation across computing resources: i-, j-, and k-dimensions are related to the grid partitioning, while s-dimension is associated with the order of executing MPDATA stages.

Computing each MPDATA block requires extra calculation and communication for every stage because of data dependencies between stages. Extra calculations and communication have to take place on the borders between adjacent blocks, in order to ensure the correct results of the whole algorithm. Therefore, blocks are extended by adequate halo areas. As a result, blocks are independent of each other, and there are no communication between blocks within a single MPDATA time step.

The sizes of halo areas are determined in all the four dimensions (i, j, k and s), according to data dependencies between MPDATA stages. Thus, each of 5 input, one output, and 16 temporary matrices, is partitioned into chunks of size $nB \times mB \times lB$, which further is expanded by adequate halo areas with sizes iL, iR, and jL, jR, as well as kL, kR.

This approach allows us to avoid memory transfers for intermediate computation at the cost of extra computation associated with halo areas in chunks of temporary matrices, as well as extra communication between the main and cache memories, corresponding to halo areas in chunks of the input matrices. Another advantage of this approach is reducing the main memory consumption because all the intermediate results are stored in the cache memory only. In the case of coprocessors, it plays an important role because the size of on-board main memory is fixed, and significantly smaller than for traditional CPU solutions.

The requirement of expanding halo areas is one of the major difficulties when applying the proposed approach, taking into account data dependencies between MPDATA stages. It requires to develop a dedicated task scheduling for the MPDATA block decomposition.

4.2 Improving efficiency of block decomposition

Although the block decomposition of MPDATA allows for reducing the memory traffic, it still does not guarantee a satisfying utilization of target platforms. The main difficulty here is associated with extra computation and communication, which have impact on the performance degradation. In particular, there are three groups of extra computation and communication, corresponding to i-, j-, and k-dimensions. Some of them can be reduced or even avoided by applying the following rules:

1. The additional calculation and communication in k-dimension can be avoided if $lB = l$, and the size $nB \times mB \times lB$ of block is small enough to save in cache all the required data. This rule is especially useful when the value of l is relatively small, as it is in the case of NWP, where l is in range $[64, 128]$.

2. The overheads associated with j-dimension is avoided by leaving partial results in the cache memory. It becomes possible when extra computation are repeated by adjacent blocks. In this case, some results of intermediate computation have to reside in cache for executing the next block. This rule requires to develop a flexible management of computation for all the stages, as well as an adequate mapping of partial results onto the cache space. In consequence, all the chunks are still expanded by their halo areas, but only some portions of these chunks are computed within the current block. It means that this approach does not increase the cache consumption. The idea of improving the efficiency of block decomposition is shown in Fig. 2.

3. In order to reduce additional calculations in i-dimension, the size nB should be as large as possible to save in the cache hierarchy all the data required to compute a single block.

5. MPDATA PARALLELIZATION

5.1 Partitioning of cores/threads into independent teams

Another method of improving the efficiency of the proposed block decomposition is partitioning of available cores/ threads into teams. Each team corresponds to a piece of the MPDATA grid, and executes calculation according to the block decomposition strategy. As a results, the MPDATA

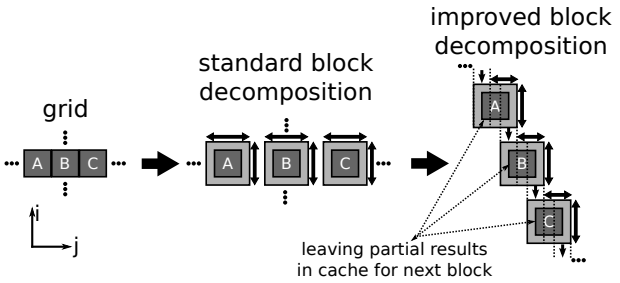


Figure 2: Idea of leaving partial results in cache memory

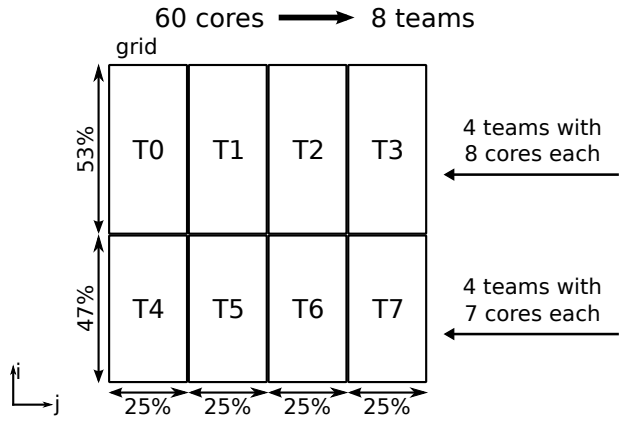


Figure 3: Partitioning of Intel MIC processing cores into teams when performing MPDATA computation

grid is distributed into pieces, and then each piece is decomposed into MPDATA blocks. Computation executed within teams are independent within each time step.

The proposed method allows us to reduce inter-cache communication overheads due to communication between neighbour threads, as well as their synchronization. What is also important, this method increases opportunities for the efficient load distribution of MPDATA computation onto available resources. These advantages are achieved at the cost of some extra computation performed by teams.

In general, pieces of the grid corresponding to different teams are characterized by various sizes. Numbers of cores/ threads assigned to teams are different, as well. Fig. 3 shows an example of partitioning 60 Intel MIC processing cores into 8 teams, and distribution of the MPDATA grid onto teams. To provide load balancing, we distinguish 4 teams with 8 cores each, and 4 teams 7 cores each. Moreover, pieces of the MPDATA grid corresponding to these teams have different sizes along i-dimension.

5.2 Task and data parallelisms

In order to utilize computing resources available in the Intel Xeon Phi coprocessor, the proposed approach employs two main levels of parallelism:

- task parallelism which allows for utilization of more than 200 logical cores;
- data parallelism to use efficiently 512-bit vector processing units.

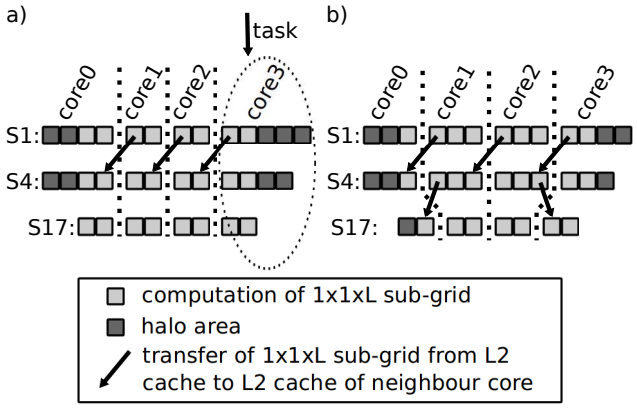


Figure 4: Example of distribution of calculation within a team of cores: (i) first scenario sacrifices load balancing for reducing intra-cache communication; (ii) second scenario improves load balancing at the cost of increasing intra-cache communication

All computation executed within every MPDATA block of size $nB \times mB \times lB$ are distributed across available threads in each team. Each block is partitioned into sub-blocks of size $nB^* \times mB^* \times lB$, where partitioning takes place along i and j dimensions. Within an MPDATA block, tasks are assigned to sub-blocks, where each sub-block is computed by a corresponding thread.

Another level of parallelization is vectorization applied within each thread, so the resulting SIMDification is performed within k-dimension. In consequence, the value of size lB has to be adjusted to the vector size.

At the same time, for a fixed MPDATA block, a sequence of stages is executed, taking into account the adequate sizes of halo areas. Due to the data dependencies of MPDATA, appropriate synchronizations between MPDATA stages are necessary. Finally, within each team its MPDATA blocks are processed sequentially, following the order proposed for the block decomposition in Section 4. Naturally, teams perform all computation in parallel.

5.3 Distribution of calculation within team of cores

An appropriate distribution of calculation within team of cores is crucial for optimizing the overall system performance. The purpose is to provide the trade-off between two coupled criteria: load balancing and intra-cache communication. In fact, aiming at improving the load balancing within a team, we have to take into account the possibility of undesirable effect of increasing the communication between cores/threads, implemented through the cache hierarchy.

Fig. 4 illustrates an example of two scenarios of distributing MPDATA calculation within a given team of cores for the block of size $1 \times 8 \times l$. In this example, a single team corresponds to 4 cores (one thread per core is assumed). The first scenario (Fig. 4a) features less amount of intra-cache communication between tasks (threads) than the second one. However, the load imbalance within the team of cores is noticeable in this scenario. The second scenario provides a better load balance across available resources assigned to team, but it requires more intra-cache communication.

Because of intra-cache communication between tasks, the

overall system performance depends strongly on a chosen task placement onto available cores (threads). Therefore, the physical core affinity plays a significant role in optimizing the system performance. In this work, the affinity is adjusted manually, to force communication between tasks placed onto the closest adjacent cores, as much as possible. This increases the sustained intra-cache bandwidth, as well as reduces cache misses, and the latency of access to the cache memory.

6. PERFORMANCE RESULTS

In this section, we present preliminary performance results obtained for the double precision 3D MPDATA algorithm on the platforms introduced in Section 2. In all the tests, we use the icc compiler as a part of Intel Parallel Studio 2013, with the same optimization flags. The best configurations, including number of teams, size of block, and distribution of computation within team, are chosen in an empirical way, individually for each platform. Moreover, we use Intel Xeon Phi in the native mode.

Currently, only the first four stages of MPDATA are implemented and tested. These four stages correspond to the linear version of MPDATA. Since all the input matrices are required to provide the correctness of calculation, the overall performance for this part of MPDATA is strongly limited by the memory traffic between the main memory and cache memory.

Fig. 5 presents the normalized execution time of the 3D MPDATA algorithm, for 500 time steps and the grid of size $1022 \times 512 \times 63$.

Fig. 5a shows a performance gain for the improved version of block decomposition. The proposed method of reducing extra computation allows us to speedup MPDATA block version from 2 to 4 times, depending on the platform used and size of the grid.

The impact of block size on the overall performance is illustrated in Fig. 5b. In general, the larger block size the higher performance is achieved. However, the limiting factor is the cache size.

According to Fig. 5c, among five tested configurations the best results are obtained for the configuration containing 14 teams, with 16 threads per each team. Rather surprisingly this configuration uses only 56 cores. These configurations are highly distinguished with respect to load balancing of MPDATA computation and intra-cache communication. In consequence, significant performance differences are observed in these tests.

The advantages of using vectorization is observed for all the platforms. In particular, for Intel Xeon Phi 7120P, it allows us to accelerate computation more than 3 times using all the available threads/cores (Fig. 5d).

Fig. 5e shows the performance obtained for different numbers of threads per core, using Intel Xeon Phi 7120P. The best efficiency of computation is achieved when running 4 threads per each core.

The performance comparison of all the platforms is shown in Fig. 5f. For each platform, we use all the available cores with vectorization enabled. As expected, the best performance result is obtained using Intel Xeon Phi 7120P. This coprocessor executes the MPTADA algorithm 2 times faster than two Intel Xeon E5-2697 v2 CPU, totally containing 24 cores. The both models of the Intel Xeon Phi coprocessor give similar performance results.

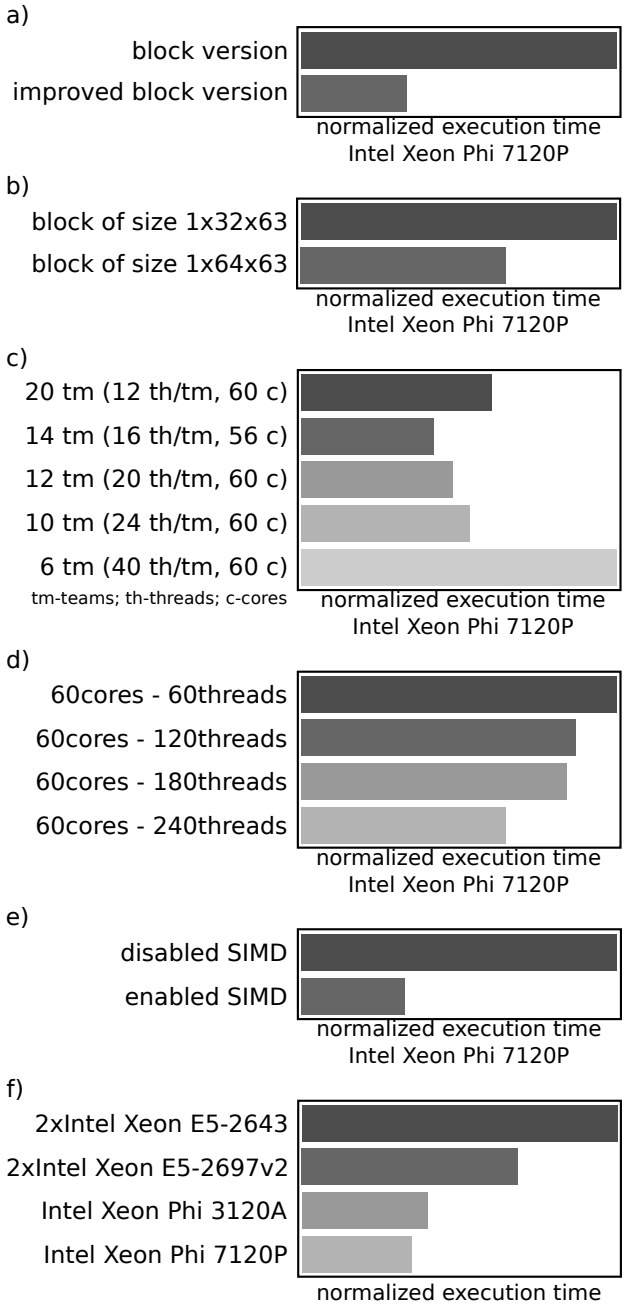


Figure 5: Preliminary performance results: (a) comparison of block and improved block versions; (b) performance for different block sizes (c) performance for different configurations of teams; (d) advantages of using vectorization; (e) performance for different numbers of threads per core; (f) comparison of Intel Xeon CPU and Intel Xeon Phi (best configurations with SIMD)

7. CONCLUSIONS

Using the Intel Xeon Phi coprocessor to accelerate computations in the 3D MPDATA algorithm is a promising direction for developing the parallel implementation of the EULAG model. Rewriting the EULAG code, and replacing conventional HPC systems with heterogeneous clusters using accelerators such as Intel MIC is a perspective way to improve the efficiency of using this model in practical simulations.

The main challenge of the proposed parallelization is to take advantage of many- and multi-core, vectorization, and cache reusing. For this aim, we propose the block version of the 3D MPDATA algorithm, based on combination of temporal and space blocking techniques. Such an approach gives us the possibility to ease memory bounds by increasing the efficient cache reusing, and reducing the memory traffic associated with intermediate computations. Furthermore, the proposed method of reducing extra computation allows us to accelerate the MPDATA block version up to 4 times, depending on the platform used and size of the grid.

Another method of improving the efficiency of the proposed block decomposition is partitioning of available cores/ threads into teams. Each team corresponds to a piece of the MPDATA grid, and executes calculation according to the block decomposition strategy. It allows us to reduce inter-cache communication overheads due to communication between neighbour threads, and their synchronization. Also, this method increases opportunities for the efficient load distribution of MPDATA computation on available resources.

An appropriate distribution of calculation within team of cores is crucial for optimizing the overall system performance. The purpose is to provide the trade-off between two coupled criteria: load balancing and intra-cache communication. Aiming at improving the load balancing within a team, the possibility of undesirable effect of increasing the communication between cores/threads has to be taken into account.

In all the performed tests, the Intel Xeon Phi 7120P coprocessor gives the best performance results. Both the many-core and vectorization features of the Intel MIC architecture play the leading role in the performance exploitation. The other important features are the block size, number of teams, number of threads per core, as well as an adequate thread placement onto physical cores. All these features have a significant impact on the sustained performance.

The achieved performance results provide the basis for further research on optimizing the distribution of MPDATA computation across all the computing resources of the Intel MIC architecture, taking into consideration features of its on-board memory, cache hierarchy, computing cores, and vector units. Additionally, the proposed approach requires to develop a flexible data and task scheduler, supported by adequate performance models. Another direction of future work is adaptation to heterogeneous clusters with Intel MICs, with a further development and optimization of code.

8. ACKNOWLEDGMENTS

This work was supported by the Polish National Science Centre under grant no. UMO-2011/03/B/ST6/03500, and by the Polish Ministry of Science and Education under grant no. BS/MN-1-112-304/2013/P.

We gratefully acknowledge the help and support provided by Jamie Wilcox from Intel EMEA Technical Marketing HPC Lab.

9. REFERENCES

[1] *Intel Architectures Comparison*. http://ark.intel.com/compare/75799,75797,64587,75283.

[2] *Intel Xeon Phi Coprocessor System Software Developers Guide*. Intel Corporation, 2013.

[3] *Parallel Programming and Optimization with Intel Xeon Phi Coprocessors, Handbook on the Development and Optimization of Parallel Applications for Intel Xeon Processors and Intel Xeon Phi Coprocessors*. Colfax International, 2013.

[4] Z. Piotrowski, A. Wyszogrodzki, and P. Smolarkiewicz. Towards Petascale Simulation of Atmospheric Circulations with Soundproof Equations. *Acta Geophysica*, 59:1294–1311, 2011.

[5] K. Rojek and L. Szustak. Parallelization of EULAG Model on Multicore Architectures with GPU Accelerators. *Lect. Notes in Comp. Sci.*, 7204:391–400, 2012.

[6] K. Rojek, L. Szustak, and R. Wyrzykowski. Performance analysis for stencil-based 3D MPDATA algorithm on GPU architecture. *Proc. PPAM 2013, Lect. Notes in Comp. Sci.*, in print:11, 2013.

[7] P. Smolarkiewicz. Multidimensional Positive Definite Advection Transport Algorithm: An Overview. *Int. J. Numer. Meth. Fluids*, 50:1123–1144, 2006.

[8] J. Treibig, G. Wellein, and G. Hager. Efficient multicore-aware parallelization strategies for iterative stencil computations. *Journal of Computational Science*, 2:130–137, 2011.

[9] M. Wittmann, G. Hager, J. Treibig, and G. Wellein. Leveraging shared caches for parallel temporal blocking of stencil codes on multicore processors and clusters. *Parallel Process. Lett.*, 20 (4):359–376, 2010.

[10] R. Wyrzykowski, K. Rojek, and L. Szustak. Model-Driven Adaptation of Double-Precision Matrix Multiplication to the Cell Processor Architecture. *Parallel Computing*, 38:260–276, 2012.

[11] R. Wyrzykowski, K. Rojek, and L. Szustak. Using Blue Gene/P and GPUs to Accelerate Computations in the EULAG Model. *Lect. Notes in Comp. Sci.*, 7116:662–670, 2012.

Hardware-aware block size tailoring on adaptive spacetree grids for shallow water waves

Tobias Weinzierl*
School of Engineering and Computing Sciences
Durham University
Durham DH13 LE, GBR
tobias.weinzierl@durham.ac.uk

Roland Wittmann
Informatics, TUM
wittmanr@in.tum.de

Kristof Unterweger
Department of Informatics
Technische Universität München (TUM)
85748 Garching, GER
unterweg@in.tum.de

Michael Bader
Informatics, TUM
bader@in.tum.de

Alexander Breuer
Informatics, TUM
breuera@in.tum.de

Sebastian Rettenberger
Informatics, TUM
rettenbs@in.tum.de

ABSTRACT

Spacetrees are a popular formalism to describe dynamically adaptive Cartesian grids. Though they directly yield an adaptive spatial discretisation, i.e. a mesh, it is often more efficient to augment them by regular Cartesian blocks embedded into the spacetree leaves. This facilitates stencil kernels working efficiently on homogeneous data chunks. The choice of a proper block size, however, is delicate. While large block sizes foster simple loop parallelism, vectorisation, and lead to branch-free compute kernels, they bring along disadvantages. Large blocks restrict the granularity of adaptivity and hence increase the memory footprint and lower the numerical-accuracy-per-byte efficiency. Large block sizes also reduce the block-level concurrency that can be used for dynamic load balancing. In the present paper, we therefore propose a spacetree-block coupling that can dynamically tailor the block size to the compute characteristics. For that purpose, we allow different block sizes per spacetree node. Groups of blocks of the same size are identified automatically throughout the simulation iterations, and a predictor function triggers the replacement of these blocks by one huge, regularly refined block. This predictor can pick up hardware characteristics while the dynamic adaptivity of the fine grid mesh is not constrained. We study such characteristics with a state-of-the-art shallow water solver and examine proper block size choices on AMD Bulldozer and Intel Sandy Bridge processors.

*Corresponding author.

HiStencils 2014
First International Workshop on High-Performance Stencil Computations
January 21, 2014, Vienna, Austria
In conjunction with HiPEAC 2014.

http://www.exastencils.org/histencils/2014/

1. INTRODUCTION

In this paper, we address an important conflict of interest faced by numerical simulations on modern architectures: while many algorithms strive to reduce the number of unknowns and required operations per accuracy via adaptivity in space and time, the latest computing architectures ask for regular data access patterns. Our objective is to team up the advantages of adaptive, octree-type meshes with regularly refined patches (blocks). We propose to merge multiple small blocks into bigger though regular blocks wherever possible, while the size of the merged blocks is chosen with respect to hardware characteristics. If adaptivity criteria refine parts of the composed regular grid regions later, the big blocks can be decomposed again.

Plain shallow water equations act as test bed for our approach well-suited for hyperbolic partial differential equations (PDEs) in general. The latter are used to model a wide range of problems of great societal and technical relevance: examples include tsunamis or earthquakes on the continental scale, radiation-sensitive cooling processes in manufacturing, as well as flow in blood vessels on the cell scale. Hyperbolic PDE models are often characterised by a multitude of scales in space and time, such that accurate solutions demand for very fine meshes—at least in certain critical regions that change in time. At the same time, the respective applications often demand for a low time to solution. Simulation-based tsunami prediction systems, for example, have to yield reasonable results within minutes.

The multitude of scales of interest for hyperbolic solvers and the local behaviour in time (reflected by the use of explicit time stepping methods) imply that efficient computational meshes for these problems need to be dynamically adaptive: they should follow the characteristic features of the solution. Furthermore, local time stepping is important where individual subgrids march in time with different time step sizes determined by the varying wave propagation speed, e.g. The finer the granularity of the adaptivity both in space and time, the "better" is the algorithm—at least in terms of the required number of unknowns and arithmetic operations.

If we express solvers with fine granular, unconstrained adaptivity in stencil notation, a large variety of stencils matching all occurring local mesh refinement situations is

required. An application of a series of such stencils in turn exhibits non-uniform data access. However, modern multi- and manycore systems offering large amount of hardware threads and vector facilities with increasing register width yield the best performance for algorithms with low memory footprint and high arithmetic intensity that are split into a vast number of homogeneous tasks. Hence, invariant stencils should be applied to big homogeneous data structures. This conflict of interest renders hyperbolic solvers on block-structured adaptive Cartesian grids a prototype challenge for novel and upcoming high-performance computing architectures.

In the presented work, we address block-structured adaptive Cartesian meshes for shallow water simulations. Our meshes result from a k-spacetree formalism [15, 17] with $k = 2$ yielding a quadtree (in 2D) or octree (in 3D), where regular Cartesian grids—we denote them as *SWE blocks*—are embedded into the leaves of the spacetree. Such a scheme facilitates dynamic block adaptivity. And adaptivity facilitates a low computational effort/memory footprint per accuracy ratio. In the present paper, we however study a different selling point of spacetree adaptivity. We use it to tune the stencil code performance: on the blocks, we apply state-of-the-art Riemann kernels yielding uniform vectorised stencils [1, 3, 9, 10]. The inter-block coupling is realised through bilinear conservative stencils from [14]. A similar technique has been proposed later in [4] for the same type of equations or in [6, 7, 11], e.g., for other challenges. Adaptive time stepping, dynamic adaptivity, and local time stepping follow [14] but are beyond scope here. Instead, our approach yields a methodology to select well-suited sizes of the Cartesian blocks for a given global adaptivity pattern automatically.

Each spacetree leaf induces a regular Cartesian block. If the size of these blocks is fixed, an adaptive spacetree induces a distinct adaptive Cartesian grid. If the size of these blocks can be configured, multiple spacetrees induce the same adaptive Cartesian grid—with a regular grid being a special case of an adaptive one. Big blocks facilitate aggressive vector optimisations, loop fusion, uniform memory access patterns, and straightforward shared memory parallelisation. Small blocks mirror loop tiling, which may improve cache usage [8, 18], but also facilitates fine-grid adaptive meshes and high block concurrency if blocks can be processed in parallel. The latter gains importance when the application faces hard memory constraints, if local time stepping is realised on a per-block basis, and if concurrency and load-balancing rely on atomic blocks. In practice, one has to choose a block size compromise. In the present paper, we make the block size a technical degree of freedom, i.e. we allow a different choice of the block size per spacetree leaf. At the same time, we follow [5] and identify regular subgrids consisting of multiple regular Cartesian grids of the same size on-the-fly. Given a performance model of the stencil operations on a regular mesh with respect to the total block size, we can then dynamically coarsen the spacetree and replace multiple spacetree leaves with one leaf hosting one capacious Cartesian mesh. This optimisation is hidden from the compute kernels, i.e. the user, and does not restrict the adaptivity pattern. Simple case studies reveal its potential impact on simulations, sketch how such a performance model can guide spacetree block configurations and give estimates for the efficiency improvements.

The proposed techniques fall into the class of autotuning of stencil codes for streaming-friendly, multicore, SIMD architectures where the stencil application is tailored to a given adaptive mesh that might change dynamically rather than making the mesh follow performance considerations. On the long term, we expect the tuning facility to be become particularly interesting when we can determine throughout the application run, i.e. online, whether few processors with high frequency and wide vector registers outperform massively parallel lower frequency configurations or the other way round. Switching on and off vector facilities, changing clock speeds, or adding cores then are not any longer showstoppers but transform into energy-aware tuning parameters, as long as the stencil operation schemes follow hardware changes.

The remainder is organised as follows: We first introduce our mesh formalism and then present our application's solver together with its stencils in Section 3. These two building blocks merge into a single block-based application that is capable to adapt the mesh-to-block mapping at hands of a performance model predicting the impact on the runtime (Section 4). In Section 5, we study the block configuration-performance interplay and derive which blocks should be merged or not in the spacetree. A brief outlook and summary in Section 6 close the discussion.

2. SPACETREE MESHES WITH REGULAR CARTESIAN BLOCKS

Let $(0,1) \times (0,1) \subset \mathbb{R}^2$ be the bounding box of the computational domain. We cut this domain equidistantly into k parts along each coordinate axis. This yields k^2 non-overlapping cubes of the same size. If we continue this splitting recursively while we decide per cube autonomously whether to refine or not, we end up with an adaptive Cartesian grid.

Let $c_0 := (0,1) \times (0,1)$, and make $\mathcal{C}$ the set of all cubes resulting from the construction process. $\sqsubseteq$ is the parent-child relation on $\mathcal{C}$. If $c_i, c_j \in \mathcal{C} : c_i \sqsubseteq c_j$, c_i is one of the k^2 subcubes resulting from the refinement of c_j. Each cube has either k^2 or no children at all. Cubes without children are *leaves* from the set $\mathcal{C}_L \subseteq \mathcal{C}$, and c_0 is the *root*.

$\sqsubseteq$ induces a directed tree graph on $\mathcal{C}$. As the nodes of this graph are cubes, i.e. spatial elements, this tree is a *k-spacetree* [15]. $k = 2$ gives the special case of a quadtree. The *height* of a spacetree is the length of the shortest path in the graph. For the trivial spacetree with $\mathcal{C} = \mathcal{C}_L = \{c_0\}$, we end up with height zero. All experiments of the present work are based upon the PDE framework Peano [16] and thus use $k = 3$. We hence omit the parameter k from now on and refer to that data structure variant as spacetree (Figure 1).

Volume-based discretisations of hyperbolic equations—or partial differential equations in general—such as finite volumes or finite elements directly yield stencils on any adaptive Cartesian grid induced by a spacetree formalism. While a direct spacetree-based stencil or system matrix derivation offers great flexibility with respect to the adaptivity, efficiency considerations as well as the intention to reuse existing software fragments suggest to add an additional mapping $f : \mathcal{C}_L \mapsto \mathbb{N}$ that embeds an equidistant Cartesian mesh with $f(c) \times f(c)$ cells into each spacetree leaf c. $f \equiv 1$ embeds a trivial grid of one cell into each leaf, i.e. each spacetree leaf

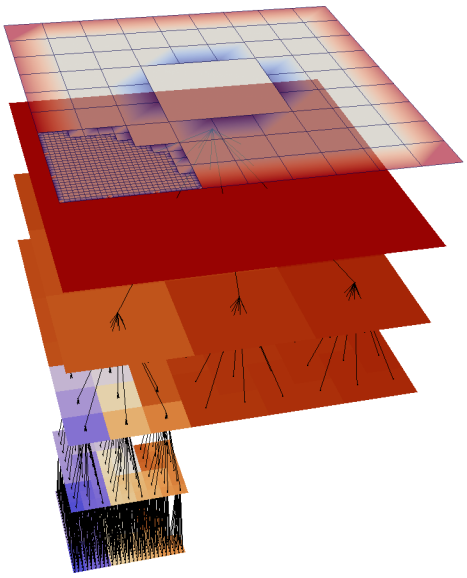


Figure 1: Adaptive Cartesian spacetree grid (top layer, transparent) with $k = 3$. The non-transparent layers below visualise the individual refinement steps, i.e. all elements of $\mathcal{C}$, with the tree relation $\sqsubseteq_{child\ of}$ as black lines.

is a cell of the computational grid Ω_h. In return,

$$f(c) = k^h \tag{1}$$

can be read as spacetree where a regular subtree of height h within the total spacetree is replaced by one spacetree node c with (1). We discuss in [5, 12, 13] how to exploit this tree replacement formalism to improve performance and introduce red-black Gauß-Seidel concurrency for direct spacetree-based PDE discretisations. From a performance point of view, it often pays off to make f return multiples of four or eight, respectively, as this fits to vector processing units—if the stencil codes exploit this fact.

Here, we start from a fixed $f(c) = n \geq 2\ \forall c \in \mathcal{C}_L$, and call the embedded regular Cartesian grids *blocks*. The spacetree then not only defines a block-structured adaptive Cartesian grid Ω_h, it also yields a non-overlapping domain decomposition of Ω_h. If we extend each $n \times n$ block by a halo layer of $\hat{n}$ cells, we obtain an overlapping domain decomposition.

Given a stencil code mapping a $(n + 2\hat{n}) \times (n + 2\hat{n})$ grid onto new values within the $n \times n$ grid and intergrid operators mapping a $n \times n$ grid onto the halo layer of another grid, we can run over the spacetree's finest level and write down any explicit time-stepping as follows:

A Run over each element of $\mathcal{C}_L$ and copy/interpolate the values of the $3^d - 1$ adjacent cells of time t onto the local halo layer. Each halo layer now holds up-to-date copies of the grid values.

B Run over each element of $\mathcal{C}_L$ and advance the values of the corresponding block from t to $t + \Delta t$.

The scheme exhibits concurrency on the block level, if we simultaneously hold simulation snapshots at t and $t + \Delta t$ per block. Then, we can update two blocks in parallel independent of each other, if they share no common vertex or face—if a neighbouring block has not advanced in time yet, simulation data of t acts as preimage of the halo initialisation, otherwise, we use the simulation data of $t + \Delta t$. For local time stepping, these two snapshots have to be interpolated anyway while the decision whether and how to advance in time also comprises wave speed considerations [14]. This scheme mirroring red-black Gauß-Seidel in linear algebra yields our *inter-block parallelisation*. It is a task-based approach with each task comprising both halo layer initialisation and unknown update.

The halo layer initialisation is pure copying of grid values into halo layers if two adjacent blocks prescribe the same grid resolution as the spacetree cubes are aligned. Their stencil is the identity. Otherwise, we realise bilinear interpolation or update fluxes according to [14] to preserve mass. Halo layer updates are cheap with respect to required floating point operations per unknown but require high memory throughput. Compared to the internal block updates, they are cheap with respect to total floating point operations as they work only on a one-dimensional submanifolds. While halo layer updates induce an overhead compared to a plain algorithm working on one regular Cartesian grid, the unknown update within the blocks dominates the overall computational workload.

3. SHALLOW WATER STENCILS

As stencil code working on the regular blocks in each leaf cell of our spacetree grid we use the SWE package [1] developed originally for teaching purposes. It processes regular Cartesian blocks of arbitrary n with $\hat{n} = 1$ halo layers. While it offers MPI parallelism and CUDA support for clusters of GPUs, we focus here on the vectorised kernels that can process a block with initialised halo data in parallel due to OpenMP. OpenMP yields our *inter-block parallelisation*.

SWE solves the basic shallow water equations given as

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = S(t,x,y),$$

where h denotes the height of the water column (water depth), u and v encode the momentum in x- and y-direction and g is the gravitational constant ($g := 9.81\,\mathrm{m/s^2}$). The source term $S(t,x,y)$ models effects of varying ocean depth (bathymetry) or frictional or Coriolis forces. In this paper, however, we set $S(t,x,y) := 0$. Solutions are characterised by water waves (traveling at a speed of $\approx \sqrt{gh}$) triggered by initial displacements of the surface, i.e., changes in the water height h (cf. Figure 2 for some artificial settings).

SWE realises an explicit Finite Volume scheme. It leads to two computational kernels executed in each time step per block as soon as the halo layer also describing global boundary conditions is initialised:

B.1 *Computation of net updates:* For each edge, an approximate solution of the Riemann problem is computed from the quantities $Q^n_{i,j} = [h_{i,j}, (hu)_{i,j}, (hv)_{i,j}]$ in the two adjacent grid cells. Following the *wave propagation* formulation [9], we compute so-called *net updates* $\mathcal{A}^{\pm}\Delta Q_{i\mp 1/2,j}$ and $\mathcal{B}^{\pm}\Delta Q_{i,j\mp 1/2}$, which determine the impact of waves entering or leaving the respective grid cells on the cell quantities.

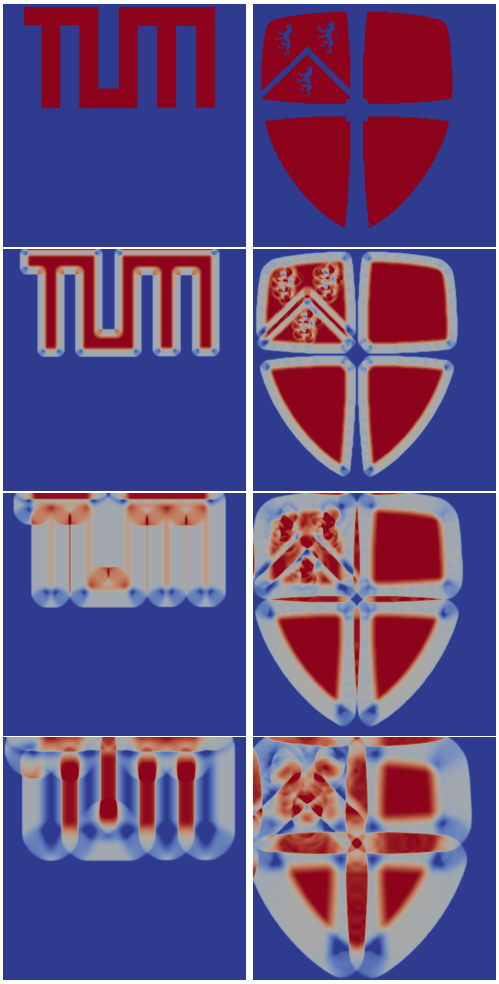


Figure 2: Two artificial water height start configurations induce waves traveling through the domain with reflecting boundary conditions. Snapshots at time $t \in \{0 \cdot 10^{-4}, 6 \cdot 10^{-4}, 1.1 \cdot 10^{-3}, 2.0 \cdot 10^{-3}\}$ resulting from a 972×972 grid.

B.2 *Updating the unknowns:* For each cell, the quantities $Q_{i,j}$ are then updated according to the balance equation

$$\begin{aligned} Q^{n+1}_{i,j} &= Q^n_{i,j} \\ &- \frac{\Delta t}{\Delta x}\left(\mathcal{A}^+ \Delta Q_{i-1/2,j} + \mathcal{A}^- \Delta Q^n_{i+1/2,j}\right) \\ &- \frac{\Delta t}{\Delta y}\left(\mathcal{B}^+ \Delta Q_{i,j-1/2} + \mathcal{B}^- \Delta Q^n_{i,j+1/2}\right), \end{aligned} \tag{2}$$

which is obtained by adding the four wave components entering the cell (i, j). The time step size Δt is restricted via the CFL condition: the maximum local wave speed (computed together with the net updates) must not exceed a fixed fraction of the mesh resolution (Δx resp. Δy) per time step.

Both, computing the net updates and updating the kernels are classical stencil-type computations, though with different characteristics. Updating the unknowns in (2) is a memory-bound loop kernel, with roughly one floating-point addition per accessed float variable as long as the steps (B.1) and (B.2) are ran one after another and not merged into one stencil. The present studies rely on a non-fused implementation. Auto-vectorisation by the compiler here can easily be achieved via a respective compiler-hint to ignore vector dependencies (`#pragma ivdep`). A similar reasoning holds for parallel for-based OpenMP parallelisation.

The loop kernel to compute the net updates runs an f-wave solver [3] on the Riemann problem. SWE provides a careful implementation of the f-wave solver that allows auto-vectorisation based on the `#pragma simd` statements introduced by the Intel compiler. Similar, it supports OpenMP concurrency. The f-wave solver requires roughly 80 floating-point operations per edge and is executed in separate loops for horizontal and vertical edges, respectively. In each loop the kernel read the quantities from two adjacent grid cells ($2 \cdot 3 = 6$ floats stored in single precision) and writes net updates for h and the normal momentum component (4 floats). Hence, the *computational intensity*, defined as the ratio of floating-point operations vs. accessed bytes of memory, of the second kernel is around two.

4. GRID TRANSITIONS

We expect the runtime per cell/stencil update to depend on the actual block size $f(c)$ whenever we update all unknowns of a block c. A naive assumption expects big blocks to be advantageous in terms of cost per unknown while small blocks allow us to tailor the grid to the solution at minimal memory cost. Given the marker

$$M(c) = \begin{cases} 0 & \text{if } c \in \mathcal{C}_L \\ n & \text{if } c \in \mathcal{C} \setminus \mathcal{C}_L \wedge \\ & \quad \exists n : \ \forall c_i \sqsubseteq c : (M(c_i) = 0 \ \wedge f(c_i) = n)) \\ \bot & \text{else} \end{cases} \tag{3}$$

on all spacetree nodes, we know due to (1) that we can replace any node c in the spacetree with $M(c) = n > 0$ and all of its children with a new node $\hat{c} \in \mathcal{C}_L$ with $f(\hat{c}) = kn$. Such a replacement searches for a $k \times k$ arrangement of blocks of the same size, merges the corresponding spacetree nodes into their spacetree parent, and replaces the original k^2 blocks by one block. The replacement preserves the fine grid Ω_h. We hence copy the values from the original blocks,

and the spacetree modification is hidden from the compute kernels. If (3) is recomputed again immediately, we may re-apply this replacement strategy.

If the start grid Ω_h is a regular Cartesian grid, such a tree replacement strategy deteriorates the spacetree after h steps with h being the height of the initial tree. On adaptive grids, it reduces the number of spacetree nodes iteratively. The interplay with dynamic adaptivity is obvious. In practice, merging always is not a good choice. Instead, it does make sense to establish a performance model $r(n)$ returning the cost per unknown for a block with $n \times n$ unknowns, and to calibrate this predictor with measurements.

The replacement of a subtree labeled with $M(c) = n$ then is advantageous if $k^2 \cdot r(n) > r(k \cdot n)$. Once a proper runtime predictor is available that takes overhead cost due to the inter-block data exchange (initialising the halo layer) into account, we end up with an autotuning approach. As each merge reduces halo layers, this autotuning also reduces the memory footprint of a given fine grid Ω_h iteratively. At the same time, each merge reduces the inter-block while it increases the intra-block concurrency.

5. RESULTS

All experiments were conducted on the Sandy Bridge and Bulldozer partitions of the CoolMAC cluster hosted at the Leibniz Supercomputing Centre. The AMD partition consists of quad-socket AMD Bulldozer Opteron 6274 nodes with 16 cores per socket, 256 GB RAM, and 2 MB exclusive L2 cache shared by two cores. They run at 2.2 GHz. The Intel partition consists of dual socket Intel Sandy Bridge-EP Xeon E5-2670 nodes with 8 real cores per socket, 128 GB RAM, and 256 KByte L2 cache per core at 2.6 GHz up to 3.3 GHz. All figures illustrate the cell updates per second for the whole simulation, i.e. include any setup or administration cost. They hence show the algorithmic throughput corresponding to the actual runtime directly.

We restrict ourselves to regular grid case studies where the block size transitions have a major impact. Statements on adaptive grids derive from the histogram of regular subtrees. The Opteron experiments are driven by the GNU compiler. Due to the pragmas, we study the vectorisation impact only on the Intel system instructed by the Intel compiler. Inter-block parallelisation is done via Intel's TBB on both systems. The intra-block parallelisation relies on OpenMP.

We first study the single core performance for grids of different size that are split into blocks of $n = 3^k \cdot 12$ (Figure 3). The Sandy Bridge system outperforms Bulldozer by up to a factor of 3.5 due to the use of single precision vector facilities and its higher frequency. For Sandy Bridge, block sizes smaller than 108 are not reasonable. Block sizes bigger than 972 also do not yield sufficient performance. The latter degradation is not observed if vectorisation is disabled. For Bulldozer, block sizes smaller than 108 also suffer from overhead. Yet, if we select bigger block sizes than 108, we observe a performance breakdown. Another breakdown is observed if we go beyond blocks of $n = 972$. The impact of Intel's turbo boost increasing the clock rate from 2.6 GHz to 3.3 GHz is not analysed further here. If interfering, it changes the results quantitatively but does not qualitatively alter the curve shape that is important to the proposed methodology.

Both systems suffer from the per-block overhead (halo layer setup and administration) for tiny block sizes. Since

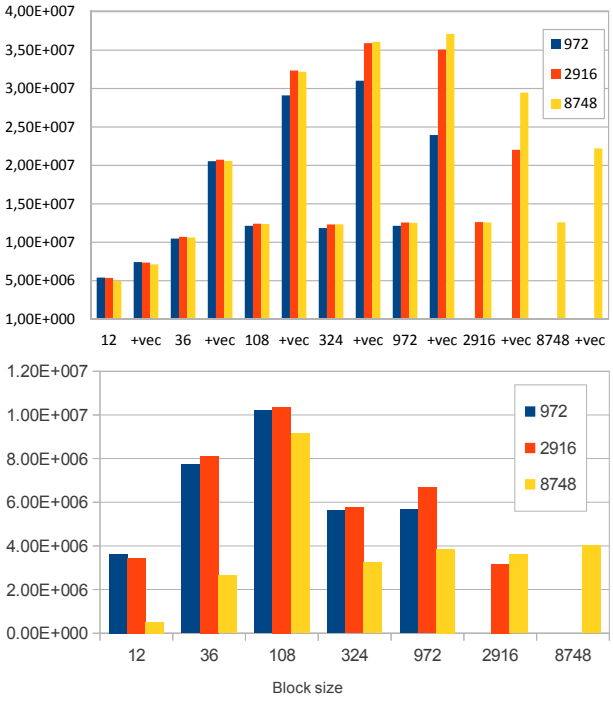


Figure 3: Throughput, i.e. cell updates per second, on a single core of Sandy Bridge (top) and Bulldozer (bottom). Each run tackles a grid of different total size split up into blocks of different size (x-axis). The Sandy Bridge measurements also compare a SIMD-loop vectorisation (`+vev`) to a plain implementation.

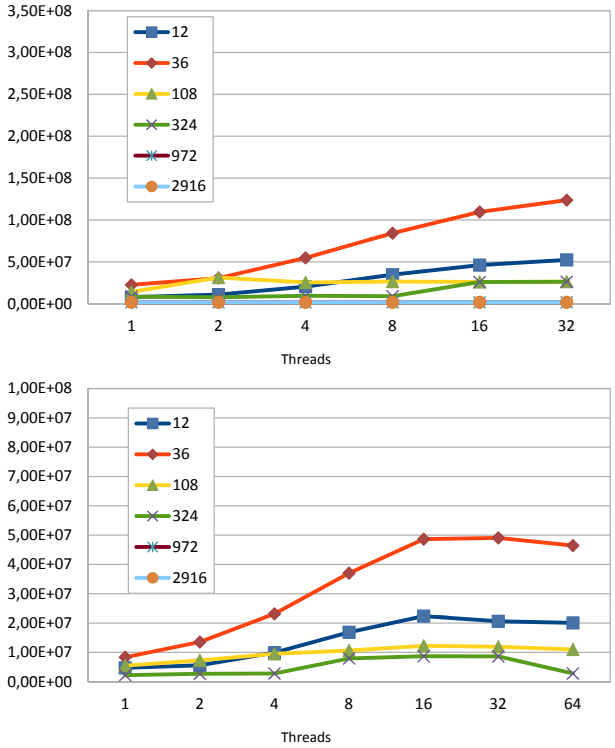


Figure 4: Inter-block parallel throughput, i.e. cell updates per second, with TBB for fixed grid of resolution 2916×2916 for Sandy Bridge (top) and Bulldozer (bottom). The grid is broken down into blocks of different size for the individual measurements, and the experiment enables different numbers of threads.

block sizes of 324 with four unknowns per cell do not fit into the L2 cache if we hold two time steps, the corresponding Bulldozer performance degradation results from the fact that the tiling cache optimisation induced implicitly by the blocking then does not avoid memory accesses anymore. Another cache threshold is hit at 972. Sandy Bridge is not that sensitive to tiling. However, its performance also degenerates for huge blocks.

If we run both codes with inter-block parallelisation where the blocks are processed in parallel (Figure 4 for $n = 2916$—other block sizes yield similar results), we observe that Sandy Bridge's hyperthreading does pay off and that both algorithms scale. While the single core performance suffers from very small block sizes, small block sizes induce a higher level of inter-block concurrency. This level in turn yields better parallel efficiency.

For the intra-block parallelisation (Figure 5), we observe a performance saturation before all cores come into play. Furthermore, the larger the blocks the better the scalability. Obviously, the inter-block parallelisation can fuse the different phases of the block updates (halo layer initialisation and the two update sweeps). However, a parallelisation exclusively of the compute loops with a serial halo layer initialisation still performs better—in particular on Intel.

Tests with a hybrid inter-/intra-block parallelisation (Figure 6) finally reveal that such a combination can not compete

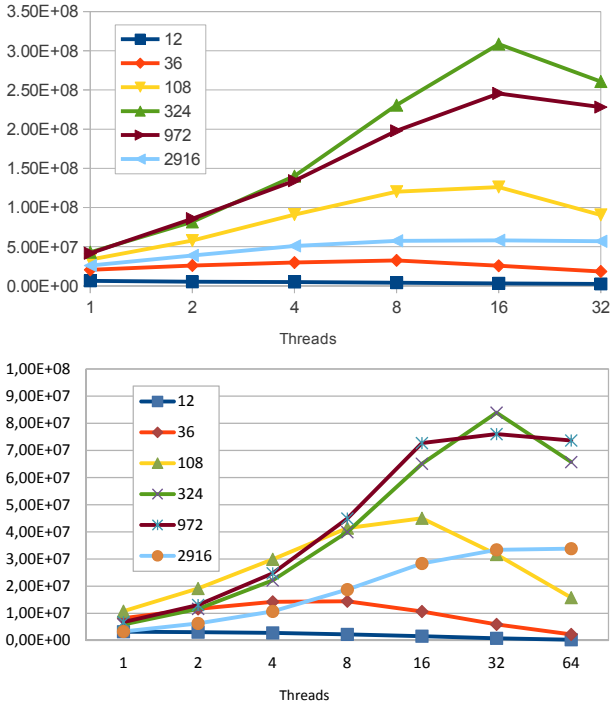


Figure 5: Intra-block parallel throughput, i.e. cell updates per second, due to OpenMP for fixed grid with 2916×2916 grid cells broken down into blocks of different size. Sandy Bridge (top) and Bulldozer (bottom). Exclusively the algorithmic phases B.1 and B.2 run in parallel due to parallel-for pragmas.

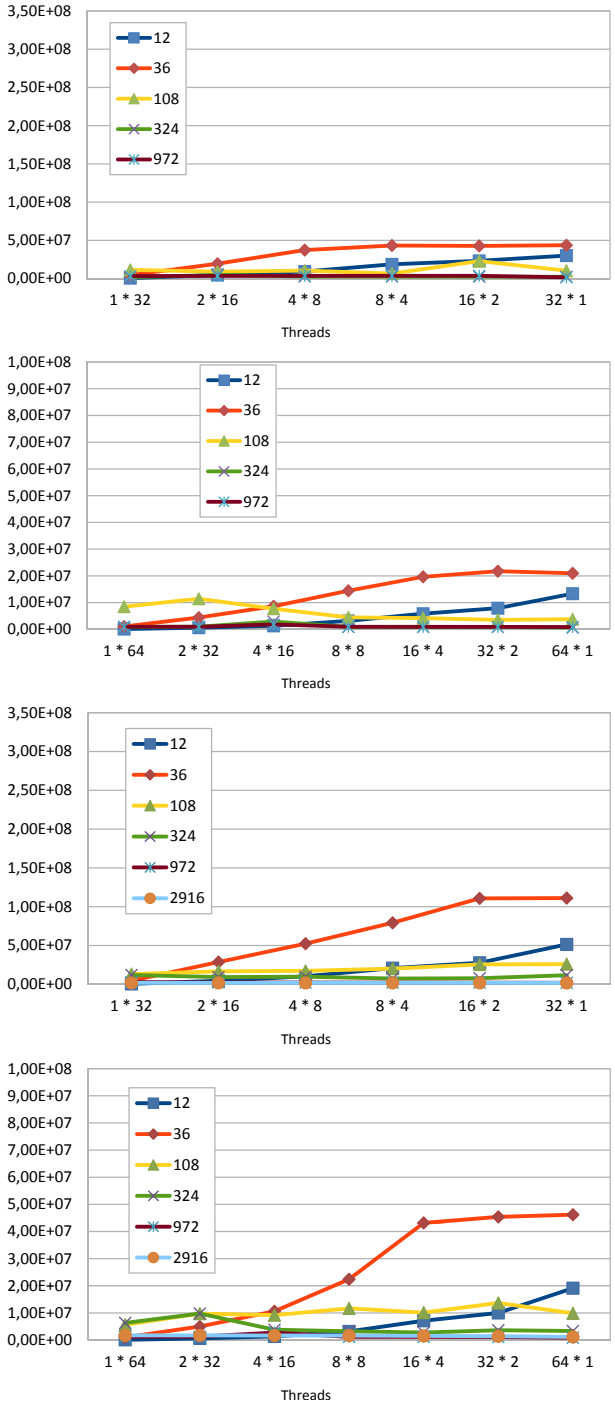


Figure 6: Throughput, i.e. cell updates per second, with intra-block and inter-block parallelisation combined. Top down: Sandy Bridge with $n = 972$, Bulldozer with $n = 972$, Sandy Bridge with $n = 2916$, and Bulldozer with $n = 2916$. Horizontally, the first number specifies the cardinality of the TBB threads whereas the number right of $*$ gives the number of enabled OpenMP threads per TBB task.

with pure inter-block concurrency as long as our grid yields sufficiently big blocks. Obviously, the stencil kernels benefit more from uniform streaming data access and a continuous filling of the compute facilities than from different algorithmic phases running concurrently. Only if the grid blocks have to remain very small (36, e.g.), the hybrid version outperforms a parallel-for variant. These experiments do not realise explicit OpenMP-to-TBB pinning, thus might be too pessimistic on some architectures for future TBB versions offering explicit pinning.

As a summary, an on-the-fly block configuration predictor should, in the present case, select a block size of around 36 at startup—this allows a reasonable per unknown performance, fine granular adaptivity, and an advantageous unknown-increase-to-runtime-reduction ratio: Whereas $n \in \{108, 324\}$ yields even higher throughput than 36, refining the grid once already yields $(k = 3)^2 = 9$ times more grid cells whereas an initial choice of $n = 36$ facilitates a finer control of the adaptivity pattern. A single core optimisation on Sandy Bridge then coarsens the spacetree as aggressively as possible as long as no block exceeds a size of 972. A Bulldozer equivalent stops already at a block size of 108. If multiple cores are available, the maximum block size should be around 324.

6. SUMMARY AND OUTLOOK

The present paper introduces a mechanism that tailors the block size of a block-structured adaptive mesh refinement solver for hyperbolic differential equations to the architecture. It starts from a given grid, preserves the fine grid, reorganises the underlying spacetree, and exploits the fact that spacetrees simplify dynamic remeshing. Basically, the approach can be rewritten as sophisticated autotuning approach for loop tiling on adaptive Cartesian grids.

While the block formalism introduces a task concurrency, using the inter-block parallelisation rarely pays off. Instead, it is more beneficial to focus on a shared memory parallelisation of the compute-intensive stencil kernels. This changes if we apply solvers with a significantly higher Flops-per-record rate—they tend to underbook the memory bus and benefit if other blocks stream in data in parallel—solvers with branches and realisations that do not exploit vector registers, or architectures with a weaker bandwidth per core. The latter also tend to undersubscribe the memory subsystem tailored to streaming applications. In the present paper, inter-block parallelism pays off if and only if the meshing enforces very small blocks. Its interplay with hard memory constraints and local time stepping one of the future challenges to study.

We want to highlight that our methodology fits perfectly to dynamically adaptive meshes, as the block structure follows the given mesh. Furthermore, it is able to react to changes in the environment. If additional cores become available, vector capabilities increase or reduce, or suddenly machine subparts with hard memory constraints shall be used in addition to/as replacement of other nodes, it is able to adapt the mesh organisation and enable the algorithm to invade such environments.

Next steps comprise the study of non-standard hardware, studies on the bigger scale, as well as shared memory and distributed memory parallelisation in combination. Of special interest however is the impact of the present ideas on different kernels with other compute characteristics.

Acknowledgements

This work partially is based on work supported by Award No. UK-c0020, made by the King Abdullah University of Science and Technology (KAUST). The support of the Leibniz Supercomputing Centre under award pr63no is highly appreciated. All software is freely available at [2, 16].

7. REFERENCES

[1] M. Bader and A. Breuer. Teaching parallel programming models on a shallow-water code. In *ISPDC 2012 – 11th International Symposium on Parallel and Distributed Computing*, pages 301–308. IEEE Computer Society, 2012.

[2] M. Bader, A. Breuer, and S. Rettenberger. SWE—the Shallow Water Equations teaching code, 2013. https://github.com/TUM-I5/SWE.

[3] D.S. Bale, R.J. LeVeque, S. Mitran, and J.A. Rossmanith. A wave propagation method for conservation laws and balance laws with spatially varying flux functions. *SIAM Journal on Scientific Computing*, 24(3):955–978, 2003.

[4] K. Mandli C. Burstedde, D. Calhoun and A. R. Terrel. Forestclaw: Hybrid forest-of-octrees amr for hyperbolic conservation laws. Technical report, Universität Bonn, 2013. Preprint.

[5] W. Eckhardt and T. Weinzierl. A Blocking Strategy on Multicore Architectures for Dynamically Adaptive PDE Solvers. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics, PPAM 2009*, volume 6068 of *Lecture Notes in Computer Science*, pages 567–575. Springer-Verlag, 2010.

[6] C. Feichtinger, S. Donath, H. Köstler, J. Götz, and U. Rüde. WaLBerla: HPC software design for computational engineering simulations. *Journal of Computational Science*, 2(2):105–112, 2011.

[7] J. Frisch, R.-P. Mundani, and E. Rank. Adaptive distributed data structure management for parallel CFD applications. In *Proc. of the 15th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2013. accepted.

[8] M. Kowarschik and C. Weiß. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In U. Meyer, P. Sanders, and J. F. Sibeyn, editors, *Algorithms for Memory Hierarchies 2002*, pages 213–232. Springer-Verlag, 2003.

[9] R. J. LeVeque. Wave propagation algorithms for multidimensional hyperbolic systems. *Journal of Computational Physics*, 131(2):327–353, 1997.

[10] R. J. LeVeque, D. L. George, and M. J. Berger. Tsunami modelling with adaptively refined finite volume methods. *Acta Numerica*, 20:211–289, 2011.

[11] P. Neumann. *Hybrid Multiscale Simulation Approaches For Micro- and Nanoflows*. Verlag Dr. Hut, München, 2013.

[12] M. Schreiber, T. Weinzierl, and H.-J. Bungartz. Cluster optimization and parallelization of simulations with dynamically adaptive grids. In F. Wolf, B. Mohr, and D. an Mey, editors, *Euro-Par 2013*, volume 8097 of *Lecture Notes in Computer Science*, pages 484–496, Berlin Heidelberg, 2013. Springer-Verlag.

[13] M. Schreiber, T. Weinzierl, and H.-J. Bungartz. Sfc-based communication metadata encoding for adaptive mesh. In Michael Bader, editor, *Proceedings of the International Conference on Parallel Computing (ParCo)*, October 2013. accepted.

[14] K. Unterweger, T. Weinzierl, D. Ketcheson, and A. Ahmadia. Peanoclaw - a functionally-decomposed approach to adaptive mesh refinement with local time stepping for hyperbolic conservation law solvers. Technical report, Institut für Informatik, Technische Universität München, June 2013.

[15] T. Weinzierl. *A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids*. Verlag Dr. Hut, 2009.

[16] T. Weinzierl et al. Peano—a Framework for PDE Solvers on Spacetree Grids, 2012. www.peano-framework.org.

[17] T. Weinzierl and M. Mehl. Peano – A Traversal and Storage Scheme for Octree-Like Adaptive Cartesian Multiscale Grids. *SIAM Journal on Scientific Computing*, 33(5):2732–2760, October 2011.

[18] C. Weiß, W. Karl, M. Kowarschik, and U. Rüde. Memory characteristics of iterative methods. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, pages 1–31. ACM Press, 1999.

The relation between diamond tiling and hexagonal tiling

Tobias Grosser
INRIA and École Normale Supérieure, Paris
tobias.grosser@inria.fr

Sven Verdoolaege
INRIA, École Normale Supérieure and KU Leuven
sven.verdoolaege@inria.fr

Albert Cohen
INRIA and École Normale Supérieure, Paris
albert.cohen@inria.fr

P. Sadayappan
Ohio State University
saday@cse.ohio-state.edu

ABSTRACT

Iterative stencil computations are important in scientific computing and more and more also in the embedded and mobile domain. Recent publications have shown that tiling schemes that ensure concurrent start provide efficient ways to execute these kernels. Diamond tiling and hybrid-hexagonal tiling are two successful tiling schemes that enable concurrent start. Both have different advantages: diamond tiling is integrated in a general purpose optimization framework and uses a cost function to choose among tiling hyperplanes, whereas the more flexible tile sizes of hybrid-hexagonal tiling have proven to be effective for the generation of GPU code.

We show that these two approaches are even more interesting when combined. We revisit the formalization of diamond and hexagonal tiling, present the effects of tile size and wavefront choices on tile-level parallelism, and formulate constraints for optimal diamond tile shapes. We then extend the diamond tiling formulation into a hexagonal tiling one, combining the benefits of both. The paper closes with an outlook of hexagonal tiling in higher dimensional spaces, an important generalization suitable for massively parallel architectures.

1. INTRODUCTION

Stencil computations are an important computational pattern in both scientific and engineering applications and they are becoming increasingly important in the embedded and mobile domain. Computational electrodynamics [13] or partial differential equations [11] are common use cases of stencils in high performance computing, whereas image and video processing are about to become driving forces in the embedded market. Even though manual and automatic optimizations of stencil computations have been designed since many years, the generation of efficient code remains a challenge especially for higher-dimensional stencils or for platforms which allow highly parallel execution on different hardware levels. With the increased use of parallel hardware in mobile markets as well as the foreseeable increase of three dimensional processing in upcoming embedded devices, a need emerges for solutions that facilitate the automatic generation of high-performance stencil codes for different devices.

For stencil computations, the tiling strategies that enable reuse along the time dimension have shown to be most efficient. Unfortunately, the standard approach uses parallel wavefronts in a skewed index space. These skewed wavefronts reduce tile-level parallelism [9] and induce load-imbalanced prologue and epilogue phases. Split tiling [5, 9] and overlapped tiling [8, 9] address this problem by enabling concurrent start along one of the original iteration space dimension. In other words, the tile schedule allows a wavefront of tiles parallel to one of the original dimensions of the index space to be executed in parallel. However, these two tiling techniques require either periodically alternating tile shapes or induce redundant computations. In contrast, the recently published diamond tiling [2] and hybrid-hexagonal tiling [6] schemes successfully obtain concurrent start without the need for redundant computations or multiple tile shapes.

Diamond tiling is a tiling strategy that uses a single n-dimensional paralleloptope[1] that is calculated such that it is possible to create a tiling that ensures that the number of tiles executable in parallel remains consistent throughout the computation, meaning that the tile schedule enables concurrent start. The advantages of diamond tiling are its integration in a general purpose compilation framework and the use of an adaptable cost function to determine tile shapes. Hybrid hexagonal-classical tiling is a tiling scheme that uses hexagonal tile shapes to enable concurrent start and to provide flexible tile size choices on one dimension. On the remaining dimensions it uses classical parallelogram tiling. The more domain specific formulation of hybrid-hexagonal tiling does not optimize tile shapes for a certain cost function, but always uses the most narrow dependence cone to derive the tile shape. On the other side, hybrid-hexagonal tiling has the advantage that it allows to adjust the time-tile height and the width along the space dimension individually. It also permits the creation of tiles with a flat summit and can ensure that tiles do not only have the same rational shape, but their integer point placement is by construction identical—all properties that have shown to be essential for efficient GPU code generation. Besides these advantages, there are also open problems. Even though diamond tiling

HiStencils 2014
First International Workshop on High-Performance Stencil Computations
January 21, 2014, Vienna, Austria
In conjunction with HiPEAC 2014.

http://www.exastencils.org/histencils/2014/

[1] A parallelotope is a general term for what is known in 2D as parallelogram and in 3D as parallelepiped.

generally explains how to derive tiling hyperplanes that enable concurrent start, a tile schedule that includes both the tile sizes as well as the parallel wavefront coefficients necessary to obtain concurrent start was not presented. Hexagonal tiling has shown beneficial for higher dimensional stencils when combined with other tiling schemes, but the formulation of hexagonal tiling itself is limited to the 2D case (1 time dimension, 1 space dimension).

This paper combines the two tiling strategies to get the best of both worlds. Its contributions are: a) an in-depth analysis of the constraints that diamond-tiling imposes on tile-sizes and wavefront coefficients, b) a formulation of conditions that ensure identical placement of integer points within the tiles, c) an extension of the original diamond tiling algorithm to a hexagonal tiling algorithm for 2 dimensional problems (1 time dimension, 1 space dimension), d) ideas for hexagonal tiling of higher dimensional stencils.

The paper is structured as follows. In Section 2 we revisit diamond tiling, provide insights on tile size and wavefront coefficient constraints and give conditions that ensure important properties of the diamond tiles. We then introduce the unified hexagonal tiling scheme in Section 3 which includes a full formulation for two dimensional tiling as well as an outlook on hexagonal tiling for higher-dimensional cases. We discuss related work in Section 4 and conclude in Section 5.

2. DIAMOND TILING

Diamond tiling [2] is a tiling technique for stencil computations where the main contribution is the combination of affine transformations and a rectangular tiling that enables concurrent start. The idea of concurrent start is to ensure that the wavefront of tiles that are executed in parallel is aligned to a concurrent start hyperplane (normally an iteration space boundary) such that the number of tiles that are executed in parallel remains constant throughout the entire computation. This ensures that already at the beginning of the computation a sufficient amount of parallelism is available. Even though the name "diamond" suggests that the tile shapes are rhombi or rhombohedra (a.k.a. diamonds) and Figure 12 in Bandishti et al. [2] also uses edges of identical length, the tile shapes formed by diamond tiling are not restricted to diamonds, but can be more general parallelograms (parallelotopes in higher dimensions) as can be seen in Figure 3 and Figure 9a. However, some restrictions to the tile shape and sizes must be enforce to ensure that concurrent start is possible.

2.1 The Pluto optimizer

Diamond tiling was presented and implemented as an extension to Pluto [3], a general-purpose optimizer for data locality and parallelism. In contrast to other approaches that directly tile the iteration space (e.g., [5, 6]), the original Pluto tiling as well as diamond tiling are implemented as a two phase process. As a first step a program transformation is calculated that exposes sequences of loops (bands) that are tileable with rectangular tiles. In the second step a rectangular tiling is performed on these bands. Combined, this yields tiles with a possibly not rectangular, but parallelotope tile shape. There are several benefits of separating these two concerns. First, when calculating the parallel bands Pluto can and does perform other optimizations, e.g., data locality optimizations such as loop fusion. Second, tiling of the transformed program makes the tile shapes independent of the tiling hyperplanes, which makes the tiling easier to describe and analyze.

Pluto calculates program transformations on a polyhedral representation. In this representation the set of executed program statements (the iteration space) is modeled with a multi-dimensional integer set where each element represents an individual statement iteration. The execution order of elements of the iteration space is described by the schedule, an integer map that assigns a possibly multi-dimensional relative execution time to each element of the iteration space. Program transformations are performed by modifying the schedule. For a single statement and a k-dimensional execution time such a schedule has the form $S = \mathbf{x} \to (\mathbf{h}_0 \cdot \mathbf{x}, \ldots, \mathbf{h}_k \cdot \mathbf{x})$, where $\mathbf{x}$ is an element of the iteration space, $\mathbf{h}_i, i \in [0, k]$ are tiling hyperplanes and $\mathbf{h}_i \cdot \mathbf{x}$ denotes the sum of the per element products of $\mathbf{h}_i$ and $\mathbf{x}$. The result of Pluto's first step are exactly these tiling hyperplanes, selected such that the distance between two statements that depend on each other is not only lexicographically nonnegative (needed for validity of the schedule), but that the distance is also nonnegative at each individual dimension. For the exact algorithm on how to select such hyperplanes, we refer to [3]. For this paper, it is sufficient to understand that the all nonnegative dependence vectors make rectangular tiling valid. We present the Pluto rectangular tiling as a schedule only transformation which we believe is easier to understand than the actual Pluto transformation which modifies the iteration space as well. Conceptually, there should be no difference. Given a schedule S and a set of tile sizes $s_i, i \in [0, k]$ a rectangularly tiled schedule of S consists of two partial schedules. The first one, S_t, is placed at the outer level and enumerates the tiles itself. It is called the tile schedule. The second one, S_p, is placed at the inner level and enumerates the points within each tile. It is called point schedule. We define $S_t = (x_0, \ldots, x_k) \to (\lfloor(\mathbf{h}_0 \cdot \mathbf{x})/s_0\rfloor, \ldots, \lfloor(\mathbf{h}_k \cdot \mathbf{x})/s_k\rfloor)$ and $S_p = S$. This tiled schedule may already expose parallelism, but it may also be necessary to fall back to pipeline parallelism by forming a wavefront schedule at the outermost tile dimension. Then, such a wavefront schedule carries itself all dependences and ensures that the inner loops can be executed in parallel. This yields $S'_t = (x_0, \ldots, x_k) \to (\lambda_0\lfloor(\mathbf{h}_0 \cdot \mathbf{x})/s_0\rfloor + \cdots + \lambda_k\lfloor(\mathbf{h}_k \cdot \mathbf{x})/s_k\rfloor, \lfloor(\mathbf{h}_1 \cdot \mathbf{x})/s_1\rfloor, \ldots, (\mathbf{h}_k \cdot \mathbf{x})/s_k\rfloor)$ with $\lambda_i \in \mathbb{Z}_{\geq 0} : i \in [0, k]$. The coefficients λ_i allow the construction of different wavefronts. We call $\lambda_0 = \cdots = \lambda_k = 1$ the default wavefront coefficients. The hyperplanes that are calculated by the original Pluto algorithm allow the formation of such a wavefront schedule, but it is not always possible to form a tile schedule that is in the same direction as a given concurrent start face $\mathbf{f}$.

2.2 The diamond tiling extensions

Diamond tiling [2] extends the Pluto algorithm in a way that ensures that for the tiling hyperplanes computed there always exist wavefront coefficients that yield concurrent start. In the following, we identify a face or hyperplane to its orthogonal vector. This paper shows that "a transformation enables tilewise concurrent start along a face $\mathbf{f}$ if and only if the tile schedule is in the same direction as the face and carries all inter-tile dependences". It also shows that "concurrent start along a face $\mathbf{f}$ can be exposed by a set of hyperplanes if and only if $\mathbf{f}$ lies strictly inside the cone formed

by the hyperplanes, i.e., if and only if $\mathbf{f}$ is a strict conic combination of all the hyperplanes". This means it finds for a concurrent start hyperplane $\mathbf{f}$ tiling hyperplanes $\mathbf{h}_i$ such that the following equality holds:

$$m\mathbf{f} = \lambda_1\mathbf{h}_1 + \cdots + \lambda_k\mathbf{h}_k \quad (1)$$

$$\lambda_i, m \in \mathbb{Z}_{\geq 0}$$

The main focus of the diamond tiling paper is to prove the conditions necessary to ensure that the calculated hyperplanes can be used to construct a concurrent start schedule as well as to give an algorithm that actually calculates such hyperplanes. We consequently refer to this publication for details. One question that was explored less is under which conditions, especially for which tile sizes and for which wavefront coefficients, the rectangularly tiled schedule achieves concurrent start. Specifically, it is not clear for which values of λ_i, s_j the following holds:

$$m\mathbf{x}\mathbf{f} = \lambda_0\lfloor(\mathbf{h}_0\mathbf{x})/s_0\rfloor + \cdots + \lambda_k\lfloor(\mathbf{h}_k\mathbf{x})/s_k\rfloor \quad (2)$$

2.3 Relation between tile sizes and wavefronts

Even though the diamond tiling yields tiling hyperplanes that allow concurrent start, to construct the full tile schedule the tile sizes s_i as well as the wavefront coefficients λ_i still need to be chosen. Choosing the correct values is important, not only to ensure that the tiles executed within the wavefront are started concurrently, but also to control the horizontal distance between tiles of the same color relative to their tile size. We call this the density of the schedule, a property important to understand the amount of computation that can be performed in parallel. Before suggesting good values, we explore the impact of different choices.

Let us first consider a simple example with symmetric dependences:

```
for t
  for i
    A[t+1][i] = A[t][i-1] + A[t][i+1]
```

Pluto's diamond tiling implementation calculates for this kernel the transformation $(t, i) \rightarrow (t - i, t + i)$ and applies rectangular tiling in the transformed space. The default wavefront coefficients $\lambda_0 = \lambda_1 = 1$ are then used to enable parallel execution. This results in the tile schedule $(t, i) \rightarrow (\lfloor(t-i)/s_0\rfloor + \lfloor(t+i)/s_1\rfloor, \lfloor(t+i)/s_1\rfloor)$. The default square tile shapes ($s_0 = s_1$) yield both concurrent start as well as a high density of tiles. Figure 1 illustrates this for $s_0 = s_1 = 4$ with the tile wavefront highlighted in red and the concurrent start hyperplane highlighted in black. The two hyperplanes being parallel shows that the tile wavefront has concurrent start. When different tile sizes are chosen for the two dimensions the default wavefront no longer yields concurrent start. In Figure 2 we illustrate for $s_0 = 4, s_1 = 6$ that the default wavefront (red) is no longer parallel to the concurrent start hyperplane (black). It is possible to still get concurrent start using the non-default wavefront coefficients $\lambda_0 = 2, \lambda_1 = 3$, which yields the schedule $(t, i) \rightarrow (2\lfloor(t-i)/6\rfloor+3\lfloor(t+i)/4\rfloor, \lfloor(t+i)/4\rfloor)$. Unfortunately, a non-default wavefront causes a large loss in tile-level parallelism throughout the computation. This effect is illustrated by the yellow wavefront in Figure 2, which is parallel to the concurrent start hyperplane (black).

Next we analyze a kernel with asymmetric dependences:

```
for t
  for i
    A[t+1][i] = A[t][i-1] + A[t][i+2]
```

Pluto derives from this kernel the transformation $(t, i) \rightarrow (t - i, 2t + i)$. This transformation combined with square tiling and the default wavefront coefficients allows concurrent start as shown in Figure 3 for $s_0 = s_1 = 4$. The reason for this, possibly surprising, result is that for a 2 dimensional stencil (1 space, 1 time) with dependence distance 1 in the time direction, the coefficient of the space dimension in the normal will always be ± 1. This ensures that when adding the two hyperplanes together their coefficients for the space dimension cancel out and we get again the concurrent start hyperplane. Consequently, the default wavefront coefficients combined with square tile sizes yield a concurrent start wavefront. As already found earlier, non-square tile sizes will prevent concurrent start with the default wavefront coeffcients.

Another interesting observation is that even though the rational tile shapes in Figure 3 are identical throughout the original iteration space, the set of contained integer points is not. The reason for this difference is that even though we use integral tile sizes in the transformed space, the borders may become non-integral in the original space. Varying integer point placements between tiles can cause problems due to additional conditions in the generated code.

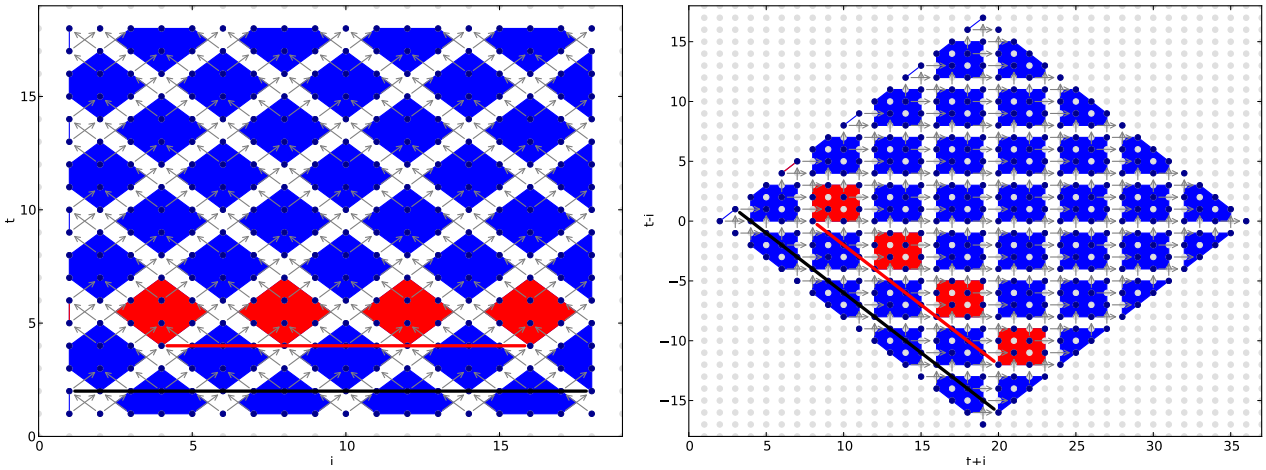
As a next step we look into a case that has dependence distances that have different lengths on the time dimension.

```
for t
  for i
    A[t+1][i] = A[t][i-1] + A[t-2][i+1]
```

For this kernel, the Pluto implementation derives the transformation $(t, i) \rightarrow (t + 3i, t + i)$. Note that this result is different from what the algorithm in [2] would produce. Apparently, the Pluto implementation is using a variation of that algorithm. It is not clear if there is a problem in this variation or that this is a mere implementation problem. As both hyperplanes have a positive coefficient for the space dimension, it is impossible to create a conic combination that eliminates the space dimension and yields a concurrent start hyperplane. According to the diamond tiling paper concurrent start is impossible and these are no valid diamond tiling hyperplanes.

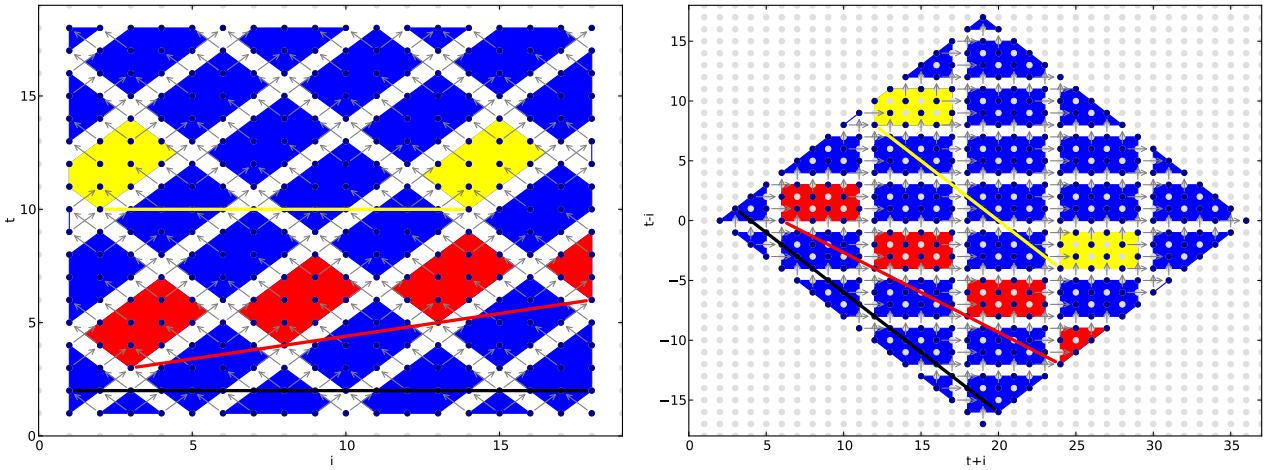
Even though the diamond tiling implementation in Pluto did not derive a valid tiling for the last kernel, there exist valid diamond tilings for it. One is the transformation $(t, i) \rightarrow (t - i, t + i)$. The same transformation was already chosen for the example illustrated in Figure 1 and according to our understanding of the cost function in Pluto, this is in fact the transformation that the algorithm of [2] would choose. The resulting tiling yields 8 computations for a per-tile memory footprint of 3.

Another valid diamond tiling transformation is $(t, i) \rightarrow (t + 3i, t - i)$. The hyperplanes in this transformation are the ones hybrid-hexagonal tiling would read off directly from the dependence cone. Given a different cost function, Pluto may also choose this transformation. The interesting point here is, that the normal of the concurrent start hyperplane in the transformed space is not anymore (1,1), but rather (1,3). In this case, the standard square tiling illustrated in Figure 4 only yields concurrent start if, instead of the default



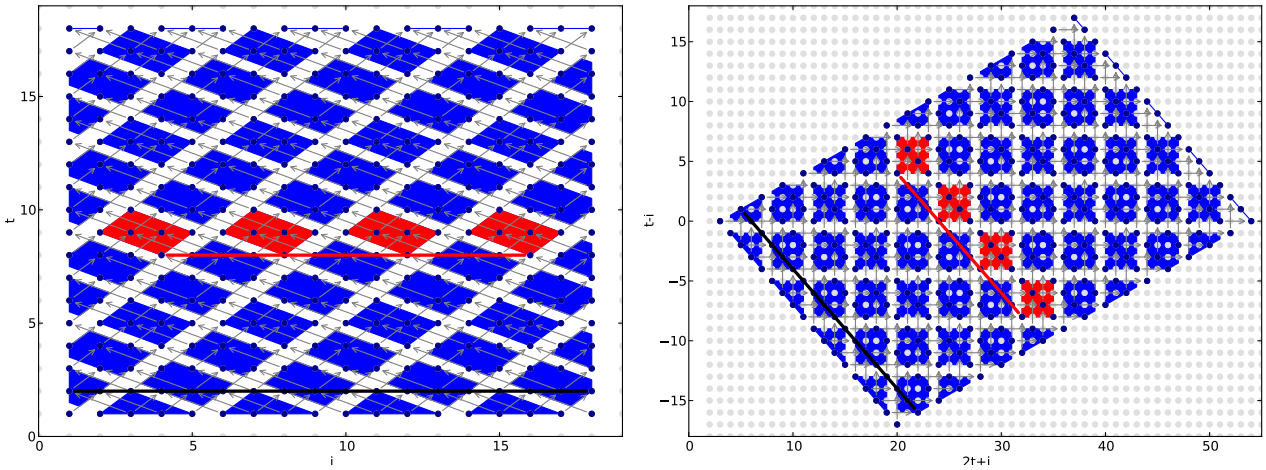
(a) original space (b) transformed space

Figure 1: Symmetric dependences & square tiling



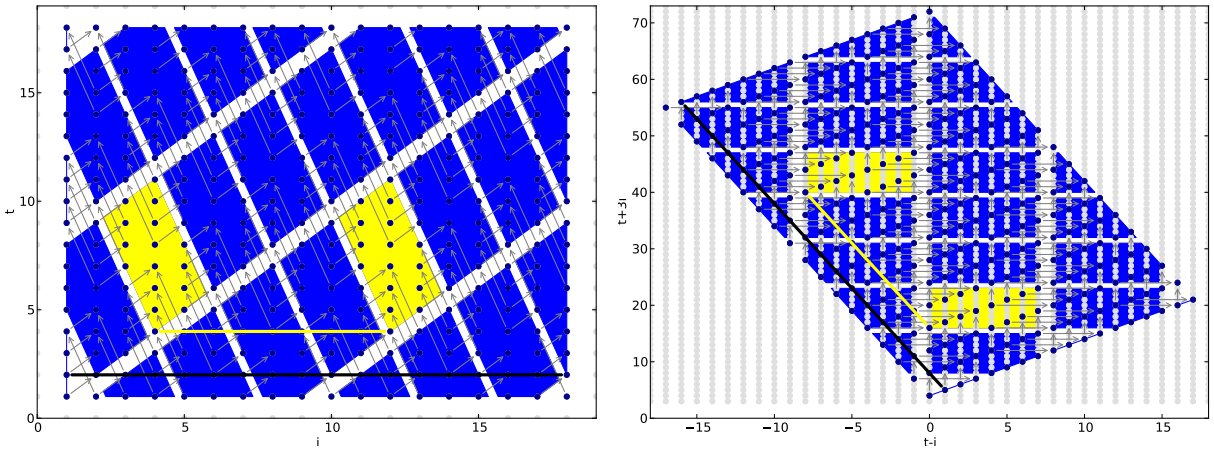
(a) original space (b) transformed space

Figure 2: Symmetric dependences & non-square tiling



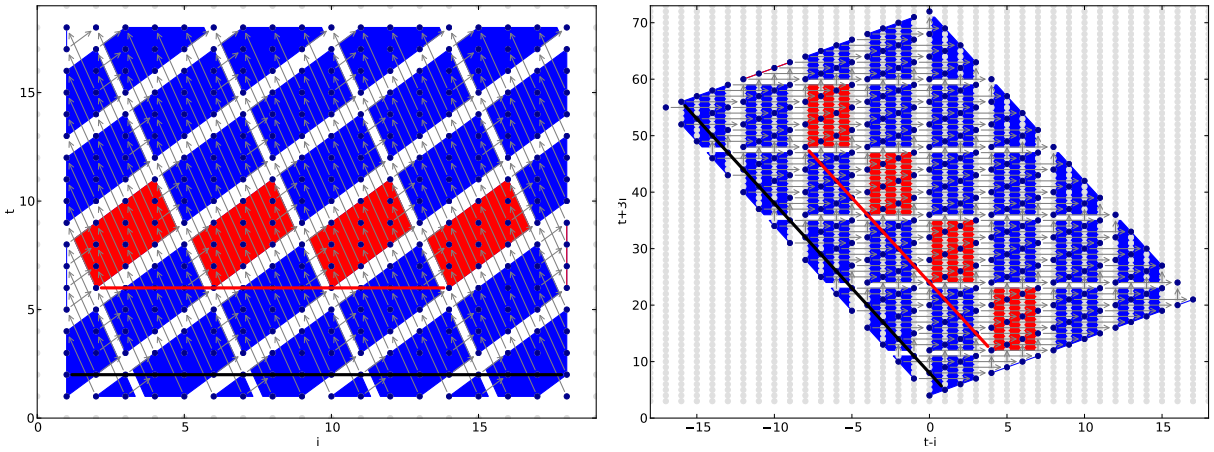
(a) original space (b) transformed space

Figure 3: Asymmetric dependences & square tiling



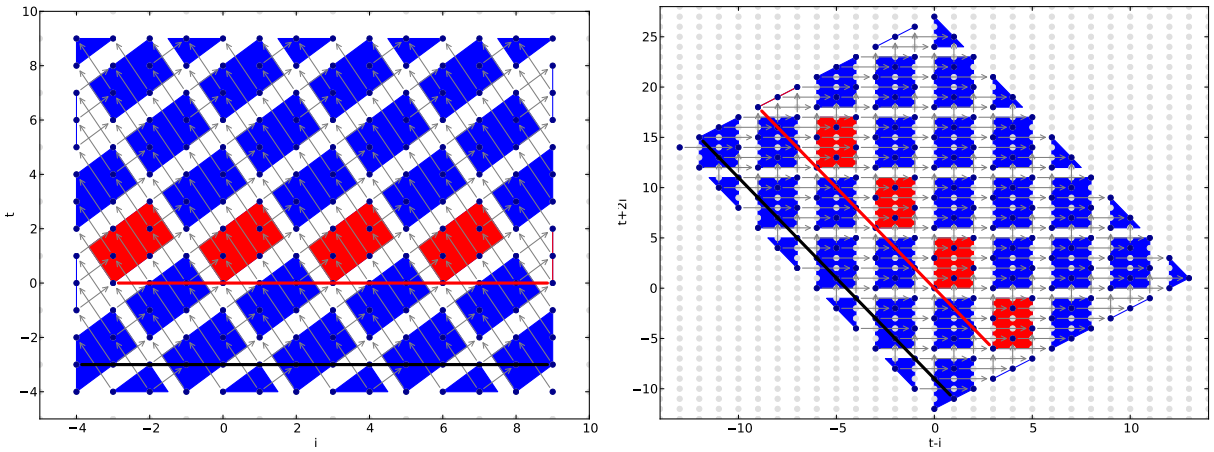
(a) original space (b) transformed space

Figure 4: More than one time step - Tiling read off from dependence cone and used by hexagonal tiling. Square tiles cause loss of tile-level parallelism.



(a) original space (b) transformed space

Figure 5: More than one time step - Tiling read off from dependence cone and used by hexagonal tiling. Non-square tiles ensure good efficiency and maximal tile-level parallelism.



(a) original space (b) transformed space

Figure 6: Diamond tiling

wavefront coefficients, $\lambda_0 = 1, \lambda_1 = 3$ are chosen. As shown earlier, this severely reduces tile-level parallelism. On the other hand, for the same memory footprint as before, this tiling executes 16 computations.

We can restore concurrent start with the default wavefront by using non-square tile sizes. Figure 5 shows a non-square tiling ($s_0 = 12, s_1 = 4$) which enables concurrent start, which has maximal tile-level parallelism and which reaches 12 computations for a memory footprint of three. Consequently, we would prefer this tiling over the previous two.

2.4 Optimal tiles with default wavefront

As seen in the previous section, the use of the default wavefront coefficients is necessary to ensure high tile-density. However, by itself it gives no guarantee neither for concurrent start nor does it ensure that all tiles share the same integer point placement. As those properties are important, we present the conditions under which they can be reached.

First, we explore the integer point placement. Assuming tiling hyperplanes $\mathbf{h}_i$ are combined into a matrix:

$$H = \begin{pmatrix} \mathbf{h}_0 \\ \vdots \\ \mathbf{h}_k \end{pmatrix}$$

then tile sizes that are multiples of the determinant of H will ensure that all tiles have the same configuration of integer points since $\det(H) \cdot H^{-1}$ is an integer matrix. The hyperplanes used, e.g., in Figure 3 yield

$$H = \begin{pmatrix} 1 & -1 \\ 2 & 1 \end{pmatrix}$$

and consequently $\det(H) = 1+2 = 3$. As $s_0 = s_1 = 4$ are not multiples of 3, the tiles differ in the integer point placement. For the same figure, tile sizes such as, e.g., $s_0 = s_1 = 3$ would ensure a uniform integer placement across all tiles. The above condition is sufficient independently of the chosen wavefront schedule.

Next, we investigate the conditions on tile sizes to ensure concurrent start with the default default wavefront coefficients. Let $h_{x,0}$ be the first component of $\mathbf{h}_x$ and $h_{x,1}$ the second. The default wavefront then is $\lfloor (h_{0,0}t + h_{0,1}i)/s_0 \rfloor + \lfloor (h_{1,0}t + h_{1,1}i)/s_1 \rfloor$. Now, to achieve concurrent start, we need to ensure that the default wavefront schedule only depends on the time dimension t and that all space dimensions (i.e., i) are eliminated. This is true under the condition $s_0/|h_{0,1}| = s_1/|h_{1,1}|$. Note that the wavefront may still depend on the fractional part of the space dimension, but this only results in a variation within a fixed range, independently of the size of the domain. We can see that in Figure 1, where we reach concurrent start for the default wavefront, this conditions holds with $4/1 = 4/1$. On the other hand, when changing the tile sizes to $s_0 = 4$ and $s_1 = 6$ as in Figure 2, the previous condition turns into $4/1 = 6/1$ and concurrent start is not possible with the default wavefront. The above shows that to obtain concurrent start the two tile sizes cannot be chosen independently, but need to be scaled together. To make this more clear we introduce a new variable s which can be chosen freely and which is then used to define $s_0 = s|h_{0,1}|$ and $s_1 = s|h_{1,1}|$ such that concurrent start is obtained.

3. UNIFIED DIAMOND AND HEXAGONAL TILING

In this section we present a extended formulation of diamond tiling which allows the creation of hexagonal tiles. The hexagonal tiles calculated are similar to the ones presented in [6], but are not identical in shape.

3.1 The schedule for hexagonal tiling (2D case)

Let us first consider a two-dimensional iteration space. To obtain such a schedule we start from the diamond tiling approach, which means we first calculate a set of tiling hyperplanes, transform the index space with these hyperplanes and then apply rectangular tiling in the transformed space. We then (optionally) transform the rectangular tiling by "stretching" the rectangular tiles along the concurrent start hyperplane. The stretched rectangular tiles in the transformed space form hexagonal tiles in the original space. As a result we have a single schedule that describes diamond tiling, if tiles are stretched by a vector of length zero, and hexagonal tiling, if they are stretched by a non-zero-length vector.

In the following description, we assume that the tiling hyperplanes h_0, h_1 are computed by the diamond tiling algorithm as described in [2]. We focus on the description of the (possibly) stretched tiling scheme in the transformed space. As input for the stretched tiling scheme, we take the tile sizes s_0, s_1 as well as a vector $\mathbf{v} = (v_0, v_1)$, which is parallel to the concurrent start hyperplane (in the transformed space). We also require that the direction vector of the concurrent start hyperplane $\mathbf{n} = (n_0, n_1)$ is strictly positive in all components, as guaranteed by the algorithm of [2].

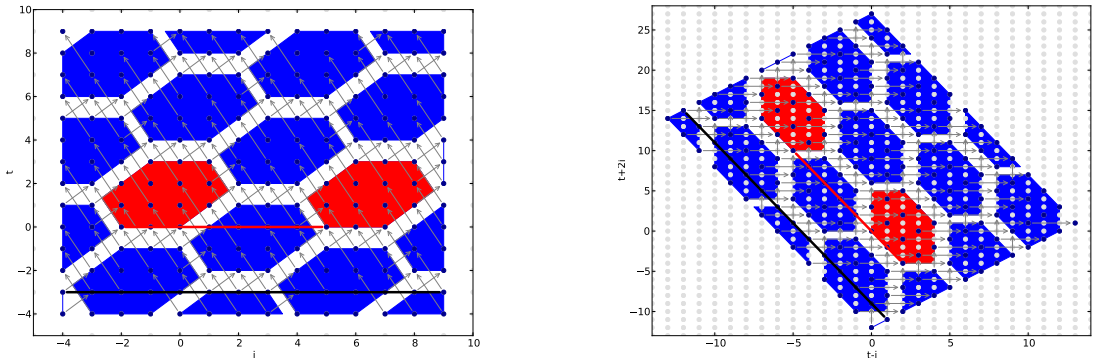
We first model diamond tiling using a standard 2D rectangular tiling in the transformed space. In this tiling the symbols s_0, s_1 define the tile sizes along the dimensions d_0, d_1 while T_0, T_1 are the resulting tile schedule dimensions (we ignore the point schedule dimensions, as this mapping is not interesting for this discussion). The following map describes such a rectangular tiling.

$$(d_0, d_1) \to (T_0, T_1) : s_0 T_0 \leq d_0 < s_0(T_0 + 1) \wedge \\ s_1 T_1 \leq d_1 < s_1(T_1 + 1)$$

Our goal is to achieve and maintain concurrent start using the default wavefront. Consequently s_0 and s_1 cannot be chosen freely (see Section 2.4). We require the user to choose tile sizes that ensure concurrent start. Figure 6 illustrates the above rectangular tiling using the transformation $(t, i) \to (t + 2i, t - i)$, as well as the tile sizes $s_0 = 6, s_1 = 3$. The red tiles show the concurrent start wavefront.

Starting from this rectangular tiling we want to stretch the contained tiles by a vector $\mathbf{v}$ with components v_0, v_1, where $\mathbf{v}$ is parallel to the concurrent start hyperplane. In principle, $\mathbf{v}$ can have either of two possible directions, but to simplify the schedule formulation we choose $\mathbf{v}$ such that $v_0 < 0 \wedge v_1 > 0$. Figure 7 shows a stretching as we obtain it for $\mathbf{v} = (-4, 2)$ and $\mathbf{n} = (1, 2)$.

Before we implement the actual stretching, we first add two additional constraints to each tile. The first one bounds each tile at its lexicographic minimal point with the concurrent start hyperplane, the second one bounds each tile at its lexicographic maximal point with the same (but translated) hyperplane. We implement the lower boundary by placing the hyperplane at the origin and by offsetting it for each tile



(a) original space

(b) transformed space

Figure 7: Hexagonal-tiling

according to the tile sizes. To offset the tile along d_0 we adjust the right hand side of the lower bound by $n_0 s_0 T_0$ and $n_1 s_1 T_1$. The upper boundary is implemented by reversing the lower hyperplane. The location of the upper hyperplanes for tile (T_0, T_1) is the origin of tile $(T_0 + 1, T_1 + 1)$.

$$
\begin{aligned}
(d_0, d_1) \rightarrow (T_0, T_1) : \\
& s_0 T_0 \leq d_0 < s_0(T_0 + 1) \wedge \\
& s_1 T_1 \leq d_1 < s_1(T_1 + 1) \wedge \\
& n_0 s_0 T_0 + n_1 s_1 T_1 \leq n_0 d_0 + n_1 d_1 \wedge \\
& n_0 d_0 + n_1 d_1 < n_0 s_0 (T_0 + 1) + n_1 s_1 (T_1 + 1)
\end{aligned}
$$

As a last step, we now stretch the tiles along $\mathbf{v}$. This requires us to increase the size of the rectangular tiles by v_0 in the d_0 dimension and v_1 in the d_1 dimension. We also account for the shifted positions of the rectangular tiles by adding some offsets o_0, o_1 to the upper and lower tile boundaries that will be derived later in this section. Finally we adjust the locations of the concurrent start planes by using $c_0 = n_1(s_0 + v_0) + n_0 v_1$ and $c_1 = n_1(s_1 + v_1) + n_0 v_0$.

$$
\begin{aligned}
(d_0, d_1) \rightarrow (T_0, T_1) : \exists o_0, o_1 : \\
& o_0 = -v_0 T_0 + v_0 T_1 \wedge o_1 = -v_1 T_0 + v_1 T_1 \wedge \\
& s_0 T_0 + o_0 + v_0 \leq d_0 < s_0(T_0 + 1) + o_0 \wedge \\
& s_1 T_1 + o_1 \leq d_1 < s_1(T_1 + 1) + v_1 + o_1 \wedge \\
& c_0 T_0 + c_1 T_1 \leq n_0 d_0 + n_1 d_1 \wedge \\
& n_0 d_0 + n_1 d_1 < c_0(T_0 + 1) + c_1(T_1 + 1)
\end{aligned}
$$

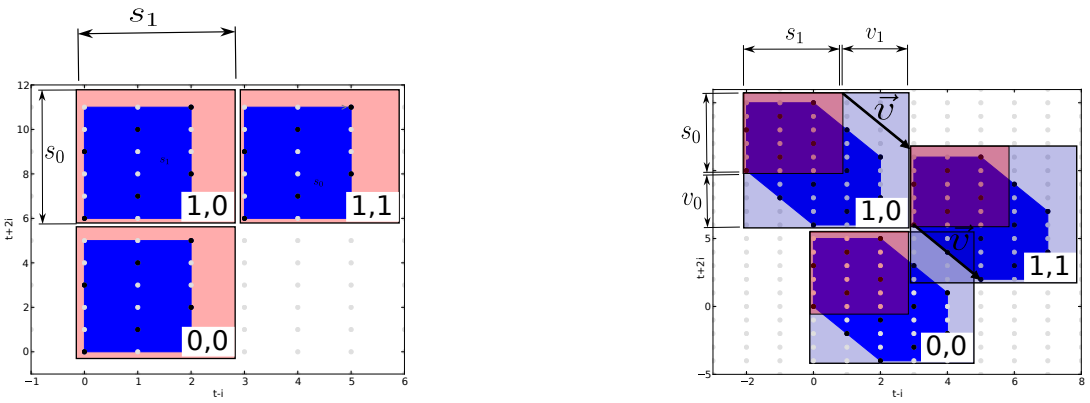
Figure 8 illustrates the last step in detail. On the left side we see in red the original square tiles (0,0), (1,0) and (1,1) each of size 6×4. On the right side, we see the tiles with the same tile numbers, but stretched along $\mathbf{v}$. We can see that the rectangular tile shapes have been extended by 4 along d_0 and by 2 along d_1 resulting in the light blue tile shapes (the dark blue tile shapes illustrate the contained integer points). We can also see that the position of the red tile shape of tile (0,0) has not moved. However, when going one step up to tile (1,0) which means increasing the tile number T_0 by one, we offset the tile by $-v_0$ along d_1 as well as $-v_1$ along d_1. Similarly, when going from tile (1,0) to tile (1,1) which means increasing the tile number T_1 by one, we offset the tile by v_0 along d_0 and v_1 along d_1. Combined this yields the offset $o_0 = -v_0 T_0 + v_0 T_1$ for d_0 and $o_1 = -v_1 T_0 + v_1 T_1$ for d_1. The new values c_0 and c_1 do now also take into account the offset of the plane. When varying T_0 we now do not only need to take the vertical tile size s_0 into account, but in addition we include the additional vertical offset v_0 as well as the changed horizontal offset v_1. To support concurrent start hyperplanes of different orientations such offsets are scaled by the relevant components of $\mathbf{n}$. The corresponding changes have been added when adjusting c_1.

A very important observation to make is that the tiles (T_0, T_1) as well as $(T_0 + 1, T_1 + 1)$ have overlapping rectangular tiles. However, the concurrent start hyperplanes that have been added right at the position of $\mathbf{v}$ ensure that the tiles are non-overlapping and still tile the full space. Also, as our stretching and translation was only along the concurrent start hyperplane, no dependences have been violated. Finally, if the previous tiling had concurrent start, stretching along the concurrent start hyperplane preserves this property.

3.2 Hexagonal tiling for higher dimensions

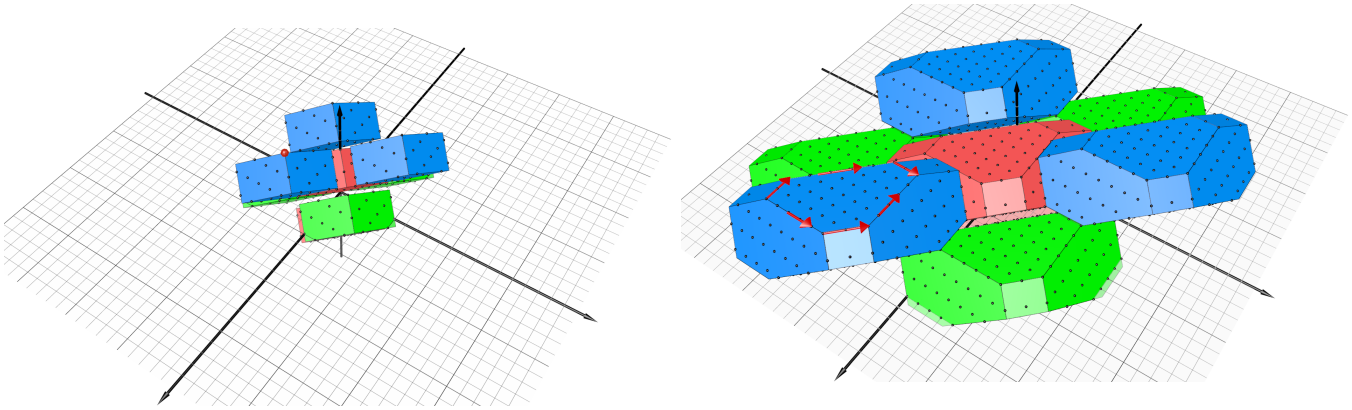
To extend our unified hexagonal tiling to higher dimensional kernels we use a shape derived from a truncated octahedra [4] to create a tiling for one time and two space dimensions, that not only provides two dimensions of parallelism, but that also gives the freedom to adjust the size of the tile shape independently for the different dimensions. Figure 9b illustrates such a tiling. In the illustration the time dimension goes upwards whereas the space dimensions go to the lower left and the lower right corner of the rendering. The hyperplane orthogonal to the time dimension is the concurrent start hyperplane. Tiles of the same color are executed at the same time step. As visible in the figure, the tiles of a single color are within a hyperplane parallel to the concurrent start hyperplane. The individual tiles of a single color are independent and can be executed in parallel. There is parallelism along both space dimensions. All tiles share a single tile shape.

The hexagonal tiling is derived from the diamond tiling illustrated in Figure 9a (the same example as used by Bandishti et al. [2]). At the beginning the peak of all tiles is formed by a single point (illustrated by the red dot on the lower left blue tile of Figure 9a). Similarly to the construction of hexagonal tiling for one space dimension, we then bound each tile at the top and at the bottom by the concurrent start hyperplane and stretch the peak to form a plane. However, for the case of two space dimension we "stretch" along three different vectors all chosen to be parallel to the concurrent start hyperplane and, in addition, to be inside one of the tiling hyperplanes. We illustrate in the blue



(a) unstretched (b) stretched

Figure 8: The stretching in the transformed space



(a) Plain Diamond tiling - unstretched

(b) Hexagonal tiling - Derived from the tiling in Figure 9a, but stretched along the concurrent start hyperplane

Figure 9: Hexagonal tiling - two space dimensions (3D rendering)

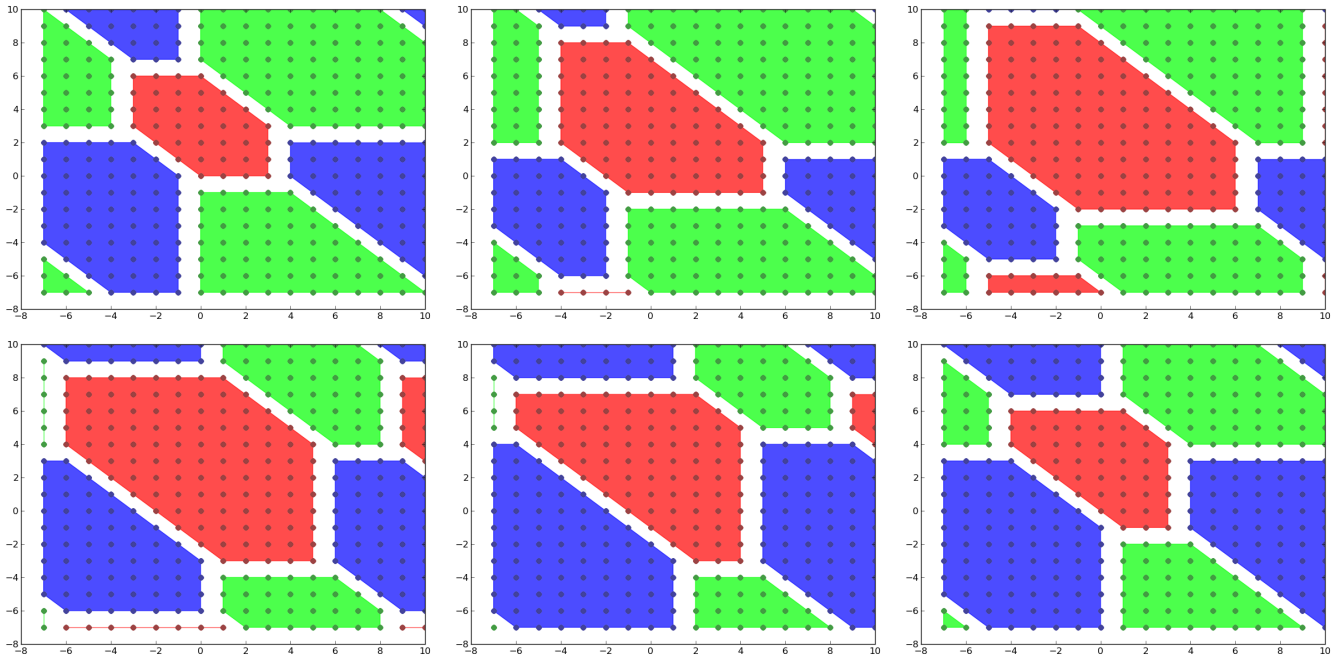


Figure 10: Hexagonal tiling - two space dimensions (Time steps 0-5)

tile at the lower left of Figure 9b these stretching vectors in red. Figure 10 illustrates that the tiling is space filling. By stretching only within the concurrent start hyperplane no dependences have been violated and concurrent start is preserved. The graphical illustration and the above claims only give an intuition of this tiling scheme. Additional work is required to understand the construction of such a tiling, its properties and its effectiveness. However, the promise we see is that we can translate the advantages of hexagonal tiling into higher dimensional cases — enabling flexible tile sizes, concurrent start as well as thread-level parallelism along multiple dimensions for higher-dimensional kernels.

4. RELATED WORK

Aside from the already discussed diamond and hybrid-hexagonal tiling [2, 6], there has been a lot of successful research in generating code to efficiently perform stencil computations. There is Pochoir [14], a domain-specific C++ framework as well as Henretty et al. [7] with a DSL-based approach. Strzodka [12] uses an in-tile wavefront traversal technique to achieve efficient cache use even with tile sizes larger than the available cache memory. All approaches generate efficient CPU code. Then, there are a set of general optimizers. PPCG [16] generates parallel CPU and GPU code using classical (time) tiling. It relies on affine transformations to extract parallelism and improve locality, using a variant of the Pluto algorithm [3]. Reservoir Labs' R-Stream is also a reference polyhedral compiler targeting GPUs [10, 15]. Par4All [1] is an open source parallelizing compiler developed by Silkan targeting multiple architectures. The compiler is not based on the polyhedral model, but uses abstract interpretation for array regions, performing powerful interprocedural analysis on the input code. Finally, there are tools that generate efficient GPU code. Here Holewinski's Overtile [8] and Grosser's split tiling [5] compilers represent, besides [6], the state-of-the-art for the automatic generation of efficient GPU code relying on overlapped and split tiling, respectively.

5. CONCLUSION

We presented a formulation of hexagonal tiling that combines the benefits of diamond tiling and hybrid-hexagonal tiling. Starting from the published diamond-tiling algorithm, we formulated conditions on tile sizes and wavefront coefficients to ensure concurrent start. We also formulated the condition that ensures the same integer point placement across all tiles. And most importantly, we extended the original diamond tiling algorithm to hexagonal tiles. The added flexibility of hexagonal tiles does not only make the choice of tile sizes more flexible but also enables the creation of tiles with a flat summit. Both these features have been shown useful for GPU code generation. Finally, we gave an outlook on our plans to extend this tiling scheme to higher dimensional stencils, an extension that will bring together flexible tile sizes and multiple dimensions of parallelism.

Acknowledgments. This work greatly benefited from regular discussions with Uday Bondhugula. It was partly funded by a Google European Fellowship in Efficient Computing, by the European FP7 project CARP id. 287767, the COPCAMS ARTEMIS project, and award 0926688 from the U.S. NSF.

6. REFERENCES

[1] Mehdi Amini, Béatrice Creusillet, Stéphanie Even, Ronan Keryell, Onig Goubier, Serge Guelton, Janice Onanian McMahon, François-Xavier Pasquier, Grégoire Péan, Pierre Villalon, et al. Par4All: From convex array regions to heterogeneous computing. In *IMPACT*, 2012.

[2] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling stencil computations to maximize parallelism. In *ACM Supercomputing Conf.*, 2012.

[3] Uday Bondhugula, J. Ramanujam, and et al. PLuTo: A practical and fully automatic polyhedral program optimization system. In *PLDI*, 2008.

[4] H.S.M. Coxeter. Regular and semi-regular polytopes. i. *Mathematische Zeitschrift*, 46(1):380–407, 1940.

[5] Tobias Grosser, Albert Cohen, Paul HJ Kelly, J Ramanujam, P Sadayappan, and Sven Verdoolaege. Split tiling for gpus: automatic parallelization using trapezoidal tiles. In *GPGPU-6*. ACM, 2013.

[6] Tobias Grosser, Albert Cohen, Sven Verdoolaege, P. Sadayappan, and Justin Holewinski. Hybrid hexagonal/classical tiling for GPUs. In *International Symposium on Code Generation and Optimization*, 2014.

[7] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. In *ICS*. ACM, 2013.

[8] Justin Holewinski, Louis-Noël Pouchet, and P Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *ICS*, pages 311–320. ACM, 2012.

[9] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *PLDI*, pages 235–244, 2007.

[10] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Baskaran, David Wohlford, Cédric Bastoul, and Richard Lethin. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *GPGPU-3*, 2010.

[11] G. Smith. *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Oxford University Press, 2004.

[12] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and H Seidel. Cache accurate time skewing in iterative stencil computations. In *Parallel Processing (ICPP), 2011 International Conference on*. IEEE, 2011.

[13] A. Taflove. *Computational electrodynamics: The Finite-difference time-domain method*. Artech House, 1995.

[14] Yuan Tang, Rezaul Alam Chowdhury, Bradley C Kuszmaul, Chi-Keung Luk, and Charles E Leiserson. The pochoir stencil compiler. In *SPAA*. ACM, 2011.

[15] Nicolas Vasilache, Benoit Meister, Muthu Baskaran, and Richard Lethin. Joint scheduling and layout optimization to enable multi-level vectorization. In *IMPACT*, Paris, France, January 2012.

[16] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM TACO*, 9(4):54, 2013.

Optimization of two Jacobi Smoother Kernels by Domain-Specific Program Transformation

Stefan Kronawitter
University of Passau
Innstraße 33
94032 Passau, Germany
stefan.kronawitter@uni-passau.de

Christian Lengauer
University of Passau
Innstraße 33
94032 Passau, Germany
christian.lengauer@uni-passau.de

ABSTRACT

Our aim is to apply program transformations to stencil codes, in order to yield highest possible performance. We observe memory bandwidth as a major limitation in stencil code performance. We conducted a small study in which we applied optimizing transformations to two Jacobi smoother kernels: one 3D 1st-grade 7-point stencil and one 3D 3rd-grade 19-point stencil. To obtain highest performance, the optimizations have to be customized for the execution platform at hand. We illustrate this by experiments on two x86 architectures and one BlueGene/Q architecture. A compiler with specific knowledge about stencil codes and execution platforms should be able to apply our transformations automatically. We are working towards such a compiler in the DFG-funded project ExaStencils.

1. INTRODUCTION

Multigrid methods [6] are widely used in scientific applications, especially in physics or chemistry simulations. Much of the time consumed by a multigrid algorithm is due to the smoother used by it. We study two concrete Jacobi smoothers [2]: one for a 3D 1st-grade 7-point smoother and one for a 3D 3rd-grade 19-point smoother. Both smoothers refer to a single output and two input grids. The three grids are located in distinct memory areas.

In the 3D 1st-grade smoother kernel, the update of a single element requires the old input value along with the nearest neighbours in each direction, and a single corresponding point of the second input array. Figure 1 depicts the 2D footprint of this stencil. The source code of the corresponding kernel appears in Figure 3.

In the 3D 3rd-grade 19-point smoother kernel, the update of a single element requires not only the nearest neighbours but all elements in a range of three points in each direction. Figure 2 depicts the 2D footprint of this stencil. The source code of the corresponding 3D kernel appears in Figure 4.

The direct implementations of the codes in Figures 3–4 have very poor performance. Suitable optimizations can improve the performance considerably, yet, they are not applied automatically by present-day compilers. To tackle this problem we present transformations, some of which exploit domain knowledge. Our intention is to let them be applied automatically by a domain-specific optimizing compiler. We are working towards such a compiler in project ExaStencils[1], which is part of the DFG-funded priority research initiative SPPEXA[2].

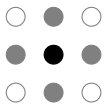


Figure 1: Footprint of a 2D 1st-grade stencil.

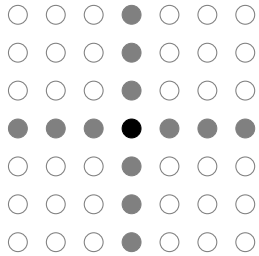


Figure 2: Footprint of a 2D 3rd-grade stencil.

A serious limiting factor turns out to be the memory bandwidth, since the computation of a single result value requires loading almost as many values from memory as floating-point operations are performed. We employ different blocking techniques to reduce the bandwidth requirements of our codes. The evaluation of our optimizations show that architectures with small, but very fast caches, like modern x86 consumer CPUs, require a more complex blocking technique than architectures with a large, but comparatively slow highest-level cache, like, e.g., IBM's BlueGene/Q.

In summary, we make the following contributions:

- a set of six optimizing transformations that can be applied automatically to stencil codes,
- an experimental evaluation of the optimizations applied to two codes on three architectures,
- insights into the suitability of the various optimizations for the individual architectures,
- insights into the basic limitations of stencil code performance.

HiStencils 2014
First International Workshop on High-Performance Stencil Computations
January 21, 2014, Vienna, Austria
In conjunction with HiPEAC 2014.

http://www.exastencils.org/histencils/2014/

[1] http://www.exastencils.org
[2] http://www.sppexa.de

```
for (int x = 1; x < dimX-1; ++x)
  for (int y = 1; y < dimY-1; ++y)
    for (int z = 1; z < dimZ-1; ++z)
      out[x][y][z] = a * in[x][y][z]
              + b1 * (in[x][y][z-1] + in[x][y][z+1] + in[x][y-1][z] + in[x][y+1][z] + in[x-1][y][z] + in[x+1][y][z])
              -  c * rhs[x][y][z];
```

Figure 3: Kernel code of a 3D 1st-grade Jacobi smoother.

```
for (int x = 3; x < dimX-3; ++x)
  for (int y = 3; y < dimY-3; ++y)
    for (int z = 3; z < dimZ-3; ++z)
      out[x][y][z] = a * in[x][y][z]
              + b1 * (in[x][y][z-1] + in[x][y][z+1] + in[x][y-1][z] + in[x][y+1][z] + in[x-1][y][z] + in[x+1][y][z])
              + b2 * (in[x][y][z-2] + in[x][y][z+2] + in[x][y-2][z] + in[x][y+2][z] + in[x-2][y][z] + in[x+2][y][z])
              + b3 * (in[x][y][z-3] + in[x][y][z+3] + in[x][y-3][z] + in[x][y+3][z] + in[x-3][y][z] + in[x+3][y][z])
              -  c * rhs[x][y][z];
```

Figure 4: Kernel code of a 3D 3rd-grade Jacobi smoother.

2. OPTIMIZATIONS

We choose three basic optimizations, which can be applied safely to improve the performance of the generated target code. These optimizations are not domain-specific. They improve the performance on any architecture and do not exploit knowledge about stencil codes. In particular, they do not alter the order of the kernel iterations. They are described in Subsection 2.1. Subsection 2.2 describes three further, domain-specific optimizations. They do affect the order of the kernel iterations and which may reduce the pressure on the bus from processor to main memory. The effect is that different architectures favour different optimizations.

2.1 Basic transformations

One of the simplest optimizations is to remove complex address computations from the innermost loop of the kernel. Consider the source code in Figure 3. If each array is stored linearly in memory, a 3D array access `in[x][y][z]` results in the polynomial index expression

```
*(in+(x*dimY+y)*dimZ+z)
```

For each iteration of the two loops on `x` and `y`, the values of `x`, `y`, `dimY` and `dimZ` remain constant. Thus,

```
in_p = in+(x*dimY+y)*dimZ
```

can be precomputed before entering the innermost loop. Then, the 3D access `in[x][y][z]` can be replaced by `in_p[z]`, which is computed much more easily. This technique of precomputing the constant values in the index polynomial for access of elements of a multidimensional array has been standard fare in compiler classes for decades [1]. However, it is not always applied by a compiler, e.g., by gcc, since different other, previous transformations can stand in the way of this optimization. We observed that gcc is not able to perform this transformation after the code was vectorized manually via vector intrinsics. Consequently, we applied it explicitly to the innermost loop, which is in every language laid out contiguously in memory – even in Java. Figure 5 illustrates this optimization for a single 3D array, regardless of whether it is linearized in memory or accessed using a 3D pointer `double ***in`.

```
for (int x = 0; x < dimX; ++x)
  for (int y = 0; y < dimY; ++y) {
    double *in_p = &in[x][y][0];
    for (int z = 0; z < dimZ; ++z)
      use(in_p[z]);
  }
```

Figure 5: Simplified address calculation for the innermost loop.

Another optimization beneficial for higher-grade stencils would be to reorder the computations such that an entire cache line is read at the same time. This prevents a repeated load of the same data into L1 cache [3]. Consider again the stencil of Figure 2. Let us assume that the data is stored in row-major order in memory (as, e.g., in C) and that the required neighbours in the same row can be loaded from contiguous addresses, while the elements from the same column have a higher stride. Consequently, when loading elements from the same column but from a different row, each access requires the corresponding cache line to be loaded into L1 cache, even if only a single double precision value is copied to a register. Therefore, the cache line could be evicted before subsequent elements are needed in the next iteration; unnecessary cache misses result. To prevent them, contiguous input data for multiple output elements can be stored in CPU registers at the same time, while the computations are reordered to process the loaded data early.

This register blocking can also be viewed as a preparatory step for vectorizing the kernel, our third basic optimization. Almost all modern processors have SIMD units, which can be used to perform the same computation in parallel on multiple data elements. Current x86 processors support the Advanced Vector Intrinsics (AVX) or, at least, the Streaming SIMD Extensions (SSE). AVX provides 256-bit registers, which can be used to store and process eight single-precision or four double-precision floating-point values in parallel. In contrast, SSE provides only 128-bit registers, which can also be used for both single-precision and double-precision values. Another common architecture in high-performance computing is the BlueGene/Q. Each of the 16 cores of a BlueGene/Q chip has a 256-bit vector unit but, in contrast to Intel's AVX, IBM's Quad Processing Extension (QPX) performs all operations

with double precision, which restricts the possible vector length to 4.

Today, most compilers provide automatic vectorizers but, in almost every situation, an explicit vectorization can yield higher performance. The corresponding compiler intrinsics can be used to declare and work with appropriate vector types. Not only must one select the suitable intrinsics carefully, one must also make sure that all input data is aligned correctly if the architecture does not support unaligned load and store operations for vector types, as is the case with BlueGene/Q. For multidimensional arrays, the programmer must also choose the width of the innermost dimension carefully: if it is not a multiple of the vector length, it must be padded to ensure that all lines are aligned properly.

2.2 Domain-specific transformations

Domain-specific optimizations of stencil codes modify the iteration domain of the kernel. Since every data element is accessed repeatedly during the computation, one simple optimization is to reorder the iterations such that each grid point is processed completely during one stay in the processor's cache. An approach commonly used is to divide the output grid into axis-aligned blocks and load all input data required to compute a single block in the highest-level cache. Then, only data points near the border of the blocks need to be loaded repeatedly.

Let us call a 2D subset of a 3D grid block a plane. The simple 3D blocking scheme just described can be improved by exploiting the fact that, for the 1st-grade kernel, only three and, for the 3rd-grade kernel, seven neighbouring planes must reside in cache at the same time. Thus, the outermost dimension must not be blocked explicitly. Nguyen et al. call this technique 2.5D blocking [4].

Another possibility to increase the performance is to combine multiple applications of the kernel code. Since the results of all kernel invocations except the last one are temporary and are not needed after the final result is computed, it is sufficient to store only few planes in a temporary buffer. E.g., if two applications of a 1st-grade Jacobi smoother are being combined, the buffer for the temporary results must be large enough to hold three planes. In a warm-up phase, the first two planes of the initial smoother application have to be computed and stored in the buffer. In the main phase, a third plane is computed, which is then stored in the next buffer slot, overwriting the oldest entry. Using all three temporary planes, one result plane can be computed and stored in the output grid. This is repeated until the last plane of the input has been processed.

If the cache is large enough to hold the complete buffer, the amount of data that must be transferred over the bus from the processor to main memory can be reduced. For large grids, even three planes may require too much memory, since the cache is typically only a few MB wide. But the amount of data that must reside in cache can be reduced by combining both blocking techniques described previously. For this so-called 3.5D blocking [4], the output grid is tiled, using 2.5D blocking, and the resulting spatial blocks are further blocked temporally. On the one hand, this reduces the amount of data to be transferred over the bus significantly, if the buffer is not evicted accidentally from cache. On the other hand, a combined temporal and spatial blocking requires a small part of the temporary grid to be recomputed repeatedly. This leads to an increase in overhead if the block size is too small or the number of smoother applications which are combined is too large. A suitable number N of smoother applications to be combined for a maximum throughput is given by the inequality

$$N \geq \left\lceil \frac{\gamma}{\Gamma} \right\rceil \tag{1}$$

introduced by Nguyen et al. [4]. γ denotes the bandwidth-to-compute ratio of the stencil kernel. Analogously, Γ describes the peak-bandwidth-to-peak-compute ratio of the execution platform.

Let us consider the 3D 1st-grade smoother code of Figure 3. An update of a single grid point requires three multiplications, six additions, one subtraction, eight reads from both input arrays and one store to the output, resulting in nineteen operations in total. Since input elements that have been loaded previously can be reused, only two elements must be fetched from main memory, one from `in` and one `rhs`, which results in transferring sixteen bytes for double precision. Analogously, a single value is written back to main memory, which results in another eight bytes to be transferred. In general the hardware has to load a cache line from main memory before it can be modified, so the old value of the output array has to be fetched from main memory, too. This so-called write-allocate increases the pressure to the bus and in total, this makes a bandwidth-to-compute ratio of $\gamma = 32$ bytes / 19 op.

Another, similar technique, which does not require the recomputation of values, is called time skewing [7]. But, the combination temporal and spatial blocking without a need to recompute values requires either additional memory or more complex address calcuations. Time skewing comes with different blocking techniques, which may result in differently sized trapezoidal blocks [5]. In contrast, the 3.5D blocking scheme described previously generates always equally sized blocks.

3. PERFORMANCE ANALYSIS

We have tested our optimizations on three different systems. Two are consumer x86 machines, while the third is a BlueGene/Q architecture. The first test environment has an Intel Ivy Bridge quadcore processor, which is clocked at 3.1 GHz when all four cores are running, a 6 MB L3 cache and 16 GB of DDR3-1600 main memory. The second system has an AMD Thuban hexacore processor, clocked at 2.7 GHz, also a 6 MB L3 cache, but only 8 GB of DDR2-800 memory. In contrast to these two consumer chips, our third test system consists of several 16-core BlueGene/Q processors, of which we use only one since we are interested in node-level performance. The deepest cache of this architecture is the L2 cache, which is 32 MB wide. Each processor also contains two built-in memory controllers with 8 GB of DDR3-1333 main memory.

Both Intel Ivy Bridge and AMD Thuban benefit from quite efficient out-of-order execution units. That is, the processor is able to reorder the instruction stream decoded from the binary for best use of the available execution units. In contrast, BlueGene/Q executes all threads in-order but, compensating for this limitation, a single physical processor core can execute up to four threads in parallel in hardware. Thus, if a single thread is not able to load the processor core to capacity, parallel threads can take up the slack. For this reason, we also measured the performance of 32 threads running on a

Platform	Peak BW	Peak Gop/s	Byte/op
Ivy Bridge	22	49	0.45
Thuban	13	31	0.42
BlueGene/Q	38	102	0.37

Table 1: Peak bandwidth (GB/s), peak double-precision compute performance (Gop/s) and bandwidth-to-compute ratio (B/op) of the three platforms.

single 16 core chip. Some of Intel's Ivy Bridge processors are also able to execute two threads concurrently in hardware. But, as all threads execute the same, small kernel code and there are no other code blocks, which might make use of different parts of the processor, the aforementioned out-of-order execution unit is sufficient to achive best performance. Because of the additional overhead, a management of multiple threads would actually decrease performance. Consequently, we disabled this so-called hyper-threading technology of Ivy Bridge.

Both x86 processors can perform a floating-point addition and an independent multiplication in parallel; BlueGene/Q has instead a fused multiply-add instruction. But most stencil codes contain far fewer multiplications than additions and cannot exploit these special instructions. Thus, the peak compute performance of stencil codes can reach at most one half of the theoretical peak compute performance of these architectures. We measured both peak bandwidth and peak stencil compute performance using micro benchmarks and their results are shown in Table 1. The peak bandwidth is measured in GB/s and the peak compute performance in Gop/s. A single operation op is either a floating point operation or a load/store instruction.

On the x86 architectures, we used gcc 4.7 to generate the executables. We ascertained that the Intel compiler icc 13 did not generate faster binaries for our test codes. On the BlueGene/Q, we used bgxlC, IBM's XL C++ compiler version 12, which is optimized for this architecture. Because of a compiler bug in the compiler's C frontend, we had to use the C++ compiler, even though our test code is written in standard C99.

3.1 3D 1st-grade smoother

The first kernel we have examined is a 3D 1st-grade Jacobi smoother. Figure 6 illustrates the benefits of our transformations an all three test platforms. As a baseline, we measured a *naive* implementation of this kernel, which is basically the same code as shown in Figure 3, except that it was parallelized using the following OpenMP pragma:

```
#pragma omp parallel for collapse(3) schedule(static)
```

which directs the compiler to distribute the iterations of the three-fold loop nest equally across all threads.

Experiment *basic opt.* includes all optimizations described in Section 2.1, whereas *temp. blocking* measured a purely temporal blocking and *3.5D blocking* refers to the combined temporal and spatial blocking described in Section 2.2. An purely spatial blocking experiment is not included, since it yields no improvement over the *basic opt.* experiment.

Our basic transformations enable the compiler to generate quite efficient code which saturates the available memory bandwidth of all test systems while using at most half of the physical processor cores. In contrast, the *naive* implementation is compute-bound on all architectures. This means that it does not use the full memory bandwidth; thus, it benefits from using additional available cores. In our experiments, every core added increased the performance.

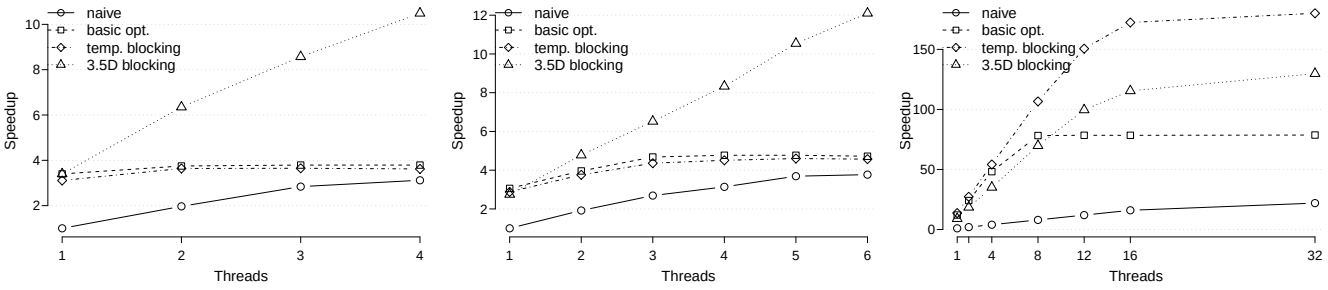
For the *3.5D blocking* technique, we determined the value of N in Inequality 1 as 5 for AMD Thuban and IBM BlueGene/Q and 4 for Intel Ivy Bridge. Consequently, we chose to implement a version that merges five Jacobi iterations. Figures 6(a) and 6(b) show a linear speedup for both x86 architectures and a resulting performance much higher than the memory bandwidth-bound *basic opt.* version. This technique transforms the bandwidth-bound to a compute-bound kernel. The same holds for the IBM BlueGene/Q processor when using all 16 physical cores, as shown in Figure 6(c), but the difference to the *basic opt.* version is not as high as for the consumer processors.

The purely temporal blocking experiment showed the most surprising behaviour. On the two x86 architectures, there is no improvement at all, even though we combined only two smoother applications, rather than five, in order to save on temporary buffer space needed to hold the data handed from one smoother call to the next. The reason is that the amount of memory required for three planes of the 512^3 elements large input does not fit into the cache of either x86 processor. Therefore, a simple temporal blocking shows no improvement. On the other hand, BlueGene/Q's cache is large enough to hold all temporary buffers as well as larger parts of the input: even for three smoother applications combined, this code performs very well as shown in Figure 6(c). Combining five smoother applications, as in case of the *3.5D blocking* version, requires temporary buffers to be large enough to store four intermediate results but, in order to reduce the danger that parts of the buffer are evicted from cache, an additional spatial blocking is required. This leads to smaller blocks of data processed consecutively, which results in an increase of the number of blocks – or, rather, the number of times that computation starts with a cold L1 cache. Due to the comparatively high latency of BlueGene/Q's L2 cache, the purely temporal blocking scheme outperforms *3.5D blocking*. In contrast to the *temp. blocking* scheme, the latter also requires the recomputation of temporary results at the border of each block, which causes additional overhead.

3.2 3D 3rd-grade smoother

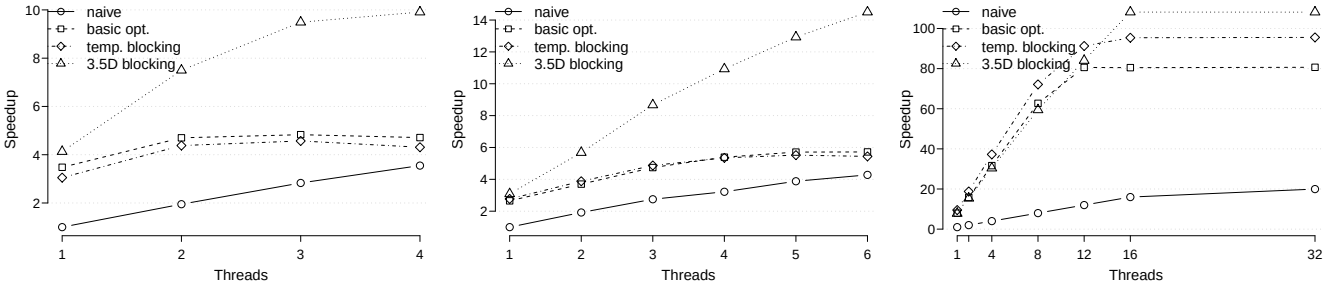
The benefits of our optimizations of the 3rd-grade smoother are depicted in Figure 7. Again, we started with a *naive* baseline implementation as shown in Figure 4, parallelized the same way as described in the previous subsection. The implementation is just as inefficient, which renders this code compute bound on the tested platforms. After applying all basic transformations, the code becomes memory-bound but, in contrast to the 1st-grade smoother, the 3rd-grade *basic opt.* version is more compute-intensive: it requires more processor cores to load the memory bandwidth to capacity.

Combining temporal and spatial blocking to the *3.5D blocking* version first requires to calculate the bandwidth-to-compute ratio γ of the 3rd-grade smoother in order to determine the value of N in Inequality 1. Consider again the code in Figure 4. Updating a single grid element requires 24 floating-point operations and 21 memory instructions; thus, a total of 45 operations are needed whereas, in the best case, only 32 bytes have to be transferred over the bus. With



(a) Intel Ivy Bridge (b) AMD Thuban (c) IBM BlueGene/Q

Figure 6: Performance results for the 1st-grade smoother on three platforms.



(a) Intel Ivy Bridge (b) AMD Thuban (c) IBM BlueGene/Q

Figure 7: Performance results for the 3rd-grade smoother on three platforms.

a bandwidth-to-compute ratio of $\gamma = 32$ bytes / 45 op $=$ 0.71 bytes/op and

$$N \geq \left\lceil \frac{\gamma}{\Gamma} \right\rceil = 2 \tag{2}$$

for all platforms, we combined two smoother applications in the *3.5D blocking* technique.

As in case of the 1st-grade smoother, the combination of temporal and spatial blocking is essential for x86 processors to maximize the throughput of the code. Figures 7(a) and 7(b) show a speedup of more than factor two compared to the *basic opt.* version.

A purely temporal blocking approach requires even larger temporary buffers than for the 1st-grade kernel, since more neighbouring planes have to be stored for the computation of a single output plane. Consequently, on both the Intel Ivy Bridge and the AMD Thuban, the *temp. blocking* scheme shows no benefit over the *basic opt.* scheme: both are memory bound. On the IBM BlueGene/Q, a simple temporal blocking leads to a faster code because the cache is large enough to store the entire buffer. But, when using all 16 cores, the *3.5D blocking* version performs better, even though both versions combine two smoother applications. That is, theoretically, the cache is large enough but, practically, cache lines corresponding to the buffer are accidentally evicted from the L2 cache and have to be reloaded for the next access.

4. CONCLUSIONS

We have conducted a small study of domain-specific program transformations to bring Jacobi smoother stencil codes to highest performance. In summary, we have learned the following.

Memory bandwidth can be a serious performance brake. In order not to let it take hold, the optimization of the codes has to be customized for the execution platform. We have studied platforms of two types:

- x86 architectures have comparatively small highest-level caches with a low latency. This requires the fragmentation of the data into comparatively small blocks. The best partitioning scheme is *3.5D blocking*. The AVX/SSE instructions for unaligned loads and stores are an additional help. However, for highest performance, vectorization has to be specified explicitly.
- BlueGene/Q is a comparatively simple architecture aimed at high processor numbers. This is reflected in the fact that performance on a single processor is poor and additional processors result in a comparatively high speedup. The highest-level cache is larger and has higher latency. This makes a purely temporal blocking scheme most suitable.

In our experiments, execution speed on the three architectures differed by at most a factor of 2, with Ivy Bridge being the fastest. Also, it must be said that theoretical peak performance is not within reach because of the low arithmetic intensity of stencil codes.

5. ACKNOWLEDGEMENTS

This work is part of project ExaStencils in the DFG priority programme SPPEXA, grant no. LE 912/15-1.

6. REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques and Tools*. Addison-Wesley, 2nd edition, 2007. Section 6.4.3.

[2] M. Bolten. *Multigrid Methods for Structured Grids and their Application in Particle Simulation*. PhD thesis, Bergische Universität Wuppertal, 2008.

[3] H. Dursun, K. Nomura, W. Wang, M. Kunaseth, L. Peng, R. Seymour, R. K. Kalia, A. Nakano, and P. Vashishta. In-core optimization of high-order stencil computations. In *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 533–538. CSREA Press, 2009.

[4] A. D. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In *Conf. on High Performance Computing Networking, Storage and Analysis (SC)*, pages 1–13. IEEE, 2010.

[5] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel. Cache oblivious parallelograms in iterative stencil computations. In *Proc. 24th Int. Conf. on Supercomputing (ICS)*, pages 49–59. ACM, 2010.

[6] U. Trottenberg, C. W. Osterlee, and A. Schuller. *Multigrid*. Academic Press, 2000.

[7] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Proc. 14th Int. Parallel & Distributed Processing Symp. (IPDPS)*, pages 171–180. IEEE Computer Society, 2000.

Evaluating the Fission/fusion Transformation of an Iterative Multiple 3D-stencil on GPUs

Siham Tabik*
Dept of Computer Architecture, University of Malaga
29071 Malaga
Spain
siham@uma.es

Alin Murarasu
Dept of Informatics, Technische Universitat Munchen
Munchen,Germany
Murarasu@in.tum.de

Luis F. Romero
Dept of Computer Architecture, University of Malaga
29071 Malaga
Spain
felipe@uma.es

ABSTRACT

This paper focuses on problems that can be expressed as iterative multiple 3D-stencils and find-out ways to optimize them for GPUs. We selected for this study a representative example, the Anisotropic Nonlinear Diffusion (AND) algorithm, which is one of the most powerful methods to denoise multidimensional data. When optimizing such algorithms for cache-based architectures, emphasis is placed on fusing all the involved stencils to increase data reutilization. However, on local-store based architectures (GPUs) the challenge is finding the optimal fission-fusion level, which implies exploring a combinatorial number of the stencils, thus making the process non-trivial. We found experimentally that the fission of 3D-stencils with high registers and shared memory pressure and fusion of 3D-stencils with high and similar concurrencies provide the best compromise reutilization, concurrency and overheads. The optimal fission-fusion combination is 1.5× faster than the case in which we fully decompose our stencil on the NVIDIA GPUs.

Keywords

3D-stencils, fission, fusion, GPUs

1. INTRODUCTION

The most powerful methods to model and manipulate images, surfaces and volumetric data sets in image processing, computer graphics and visualization are those based on solving PDE equations [15]. In general, these algorithms are often expressed as an iterative "big" multiple 3D-stencil, where each stencil has a different arithmetic intensity. As a concrete example, we study the case of the Anisotropic Non-linear Diffusion (AND) algorithm which is widely used in signal processing [15, 4]. Our goal is to develop an efficient implementation of AND for high-throughput processors such as Nvidia GPUs.

A typical optimization strategy for this kind of stencils aims at improving the locality and is based on partitioning the volume data into blocks that fit the fast on-chip memories, e.g. cache. Then, the idea is to keep each block in fast memory until it is completely denoised. Although this might seem at first a reasonable solution, in practice its performance is strongly influenced by the different computational stages (smaller stencils) of the denoising process. As the stages typically have different processing characteristics, they require updated optimizations when making the transition from one stage to another. We refer to the decomposition into stages as *kernel fission*. This is in contrast with the initial method which is more rigid, i.e. the fitting to the fast memory does not adapt to the requirements of different stages.

Nvidia GPUs are massively parallel processors that provide an advantageous ratio between GFlop/s rate and power consumption. They are characterized by a large number of cores (e.g. 16) called Streaming Multi-Processors (SM), wide SIMD units (e.g. 32 lanes), and a complex memory hierarchy including scratchpad memory (called shared memory) explicitly controllable by the programmer, a 2-level cache hierarchy, constant and texture memory, and global memory (RAM). Since they are in-order processors, GPUs hide pipeline latencies by interleaving the execution of thousands of GPU threads per core. To support a fast context switch for such a large number of concurrent threads, each SM is equipped with 32K 32-bit registers. GPUs are programmed using CUDA, a thread based programming model which offers only a subset of the synchronization options available in other thread based models for CPUs, more precisely barriers within a group of threads scheduled on one SM and atomic operations. For complex stencils, two GPU aspects are of major importance: (1) improving data reuse at register level, shared memory and L1-cache, and (2) ensuring high concurrency. The question that we try to answer is how much should we decompose a complex stencil on GPUs.

This paper brings the following two contributions to the existing work on stencil computations on GPUs:

- We express the optimization of a widely used denoising algorithm called AND as a multi-fission problem, i.e. we split the computation into four computational stages (four stencils) and we explore different combi-

*The corresponding author.

HiStencils 2014
First International Workshop on High-Performance Stencil Computations
January 21, 2014, Vienna, Austria
In conjunction with HiPEAC 2014.

http://www.exastencils.org/histencils/2014/

nations. Our approach could be applied to other applications based on complex stencils.

- We present performance results obtained on Nvidia GPUs that demonstrate the difficulty of addressing the fission problem. More precisely, the best multi-fission scheme is not obvious: The computation should be divided in 2 stages (2 stencils). This version is 1.52× faster than the four stencils version and 7× faster than the worst 2 stencils version.

The structure of the paper is the following: We present first the related works where we emphasize on the fact that although fission / fusion is an old technique applied especially to for-loops, it has not been fully studied in the scope of complex stencils on GPUs to the best of our knowledge. In Sec. 3, the mathematics behind the complex stencil of the AND algorithm is described. Our approach to fission is explained in Sec. 4. Sec. 5 describes all the GPU-implementations evaluated in this work. Sec. 6 shows our experiments and indicates the best combination of smaller stencils. Sec. 6 provides conclusions and future works.

2. BACKGROUND AND MOTIVATION

2.1 Anisotropic nonlinear diffusion (AND)

The Anisotropic nonlinear diffusion PDE-based technique is a filtering method that reduces noise and enhances local structure. It has been widely used in many fields especially in computational vision, biomedicine and differential geometry because it shows a superior performance to other filtering methods [15]. It was first introduced in visualizing electron tomograms in [5, 6]. In this work we used AND for filtering 3D-grids [4].

2.2 AND on multicore versus GPUs

Multicore systems are characterized by larger caches and a small number of cores. One thread per core can use up to 64Kb L1, 4Mb L2 and 18 Mb L3. Whereas, the GPU has a larger number of cores/SM that share a small area of shared memory (shmem), 48 Kb. Thus, a high shmem utilization per thread may penalize concurrency.

When optimizing AND for multicore architectures all emphasis is placed in splitting (or fissioning) data into blocks that fits the caches [14]. Halos are needed only in the frontiers between neighbor threads (or cores). The communication of halos between cores is performed via the shared cache levels, typically shared L2 or shared L3. Re-computation of halos, when needed, is considered only between cores from different sockets or nodes.

On the GPU, one has to consider not only a much finer splitting of data but also fission of the calculation, known as kernel fission. For the PDE-based applications that can be expressed as iterative multiple 3D-stencil, increasing the number of kernels increases accesses to global memory and decreases stress on shmem and registers. Decreasing the number of kernels increases pressure on registers and shmem since it implies a larger overhead due to copying a larger volume of halos and also re-computation in some cases. This means that the performance of this kind of problem is a trade-off between 1) on-chip memory utilization, 2)concurrency and 3)global memory accesses. Evaluating kernel fission in AND on the GPU needs exploring a large number of combinations, thus making the process non-trivial. To reduce the space of the combinations we consider only the ones that involve the dominant stencil.

2.3 Stencils

Stencils are characterized by a fixed data access pattern and number of arithmetical operations. Simple stencils, typically N-points stencils (N=7,15, ..) that perform single Jacobi iteration have been widely analyzed and tuned for multicore processors and GPUs [9, 10, 17, 12]. However, most of these studies consider only out-of-place sweeps (one grid for read and one grid for write) and do not extend their analysis to the boundary problem. When optimizing complex stencils that involve multiple simpler stencils, the boundary problem becomes an issue from a programming point of view since it requires a large volume of halos and sometimes re-computation.

Our own work is complementary to this line since it analyzes realistic applications that involve more than two 3D-stencils, more than two grids and includes boundary conditions for each stencil. Our objective is to find out the most efficient ways to optimize this kind of stencils for GPUs.

2.4 Fission / fusion optimization

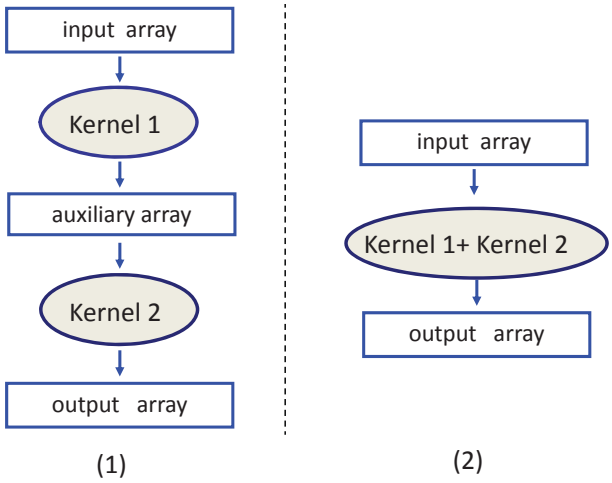


Figure 1: The type of kernel fission/fusion analyzed in this work for multiple 3D-stencils. The arrays in rectangle, array 1, intermediate array and array 2 are read and write from/to global memory in the GPU.

Kernel fission which consists of splitting one kernel into two or more kernels and its orthogonal optimization, kernel fusion, are both inspired from the well known loop fission/fusion technique. Fissioning one multiple 3D-stencil kernel implies more accesses to global memory. Figure 1 illustrates this optimization.

A large number of works show that fusion is good for performance. On multicore processors, [7, 11] showed that fusing multiple dependent kernels improves cache reuse, reduces the overheads of runtime systems and contention in work-queues. On GPUs, there is in addition several attempts to automate this optimization for the kernels of some domain specific applications. For example, the authors of [16] state

that using fusion and fission together in warehousing applications reaches a significant improvement of performance especially on large volumes of data. Our work is similar only in philosophy with the existing fission / fussion optimizations. To the best of our knowledge, the technique fission/fusion has not been covered for real-world iterative multiple 3D-stencils on GPUs.

Implementing this technique is not straightforward as we are required to first identify the stencils implied in the initial AND-kernel, analyze dependencies between them and subsequently decide how to split. The tradeoffs that characterize GPUs, e.g. between locality and concurrency, complicate the fission decision. Therefore, multiple fission variants are implemented and later compared based on their respective performance numbers measured on GPUs. We made decisions based on the weight of the smaller stencils to limit the evaluation space to the most interesting variants.

3. THE MATHEMATICAL BASIS OF AND

The most sophisticated and powerful denoising algorithms are those that solve Partial Differential Equations (PDE) [1, 2, 8, 4]. In this work we focus on a representative one of these methods, the Anisotropic Nonlinear Diffusion. AND accomplishes a sophisticated edge-preserving denoising that takes into account the structures at local scales. AND tunes the strength of the smoothing along different directions based on the local structure estimated at every point of the multidimensional image. This section presents local structure determination via structure tensors, the concept of diffusion, a diffusion approach commonly used in image processing and, finally, details of the numerical implementation.

3.1 Estimation of structure tensor

The *structure tensor* is the mathematical tool that allows us to estimate the local structure in a multidimensional image. Let $I(\mathbf{x})$ denote a 3D image, where $\mathbf{x} = (x, y, z)$ is the coordinate vector. The structure tensor of I is a symmetric positive semi-definite matrix given by:

$$\mathbf{J}(\nabla I) = \nabla I \cdot \nabla I^T = \begin{bmatrix} I_x^2 & I_x I_y & I_x I_z \\ I_x I_y & I_y^2 & I_y I_z \\ I_x I_z & I_y I_z & I_z^2 \end{bmatrix} \quad (1)$$

where $I_x = \frac{\partial I}{\partial x}$, $I_y = \frac{\partial I}{\partial y}$, $I_z = \frac{\partial I}{\partial z}$ are the derivatives of the image with respect to x, y and z, respectively.

The eigen-analysis of the structure tensor allows determination of the local structural features in the image [15]:

$$\mathbf{J}(\nabla I) = [\mathbf{v_1}\ \mathbf{v_2}\ \mathbf{v_3}] \cdot \begin{bmatrix} \mu_1 & 0 & 0 \\ 0 & \mu_2 & 0 \\ 0 & 0 & \mu_3 \end{bmatrix} \cdot [\mathbf{v_1}\ \mathbf{v_2}\ \mathbf{v_3}]^T \quad (2)$$

The orthogonal eigenvectors $\mathbf{v_1}$, $\mathbf{v_2}$, $\mathbf{v_3}$ provide the preferred local orientations, and the corresponding eigenvalues μ_1, μ_2, μ_3 provide the average contrast along these directions.

3.2 Diffusion in image processing

Diffusion is a physical process that equilibrates concentration differences as a function of time, without creating or destroying mass. In image processing, density values play the role of concentration. This observation is expressed by the *diffusion equation* [15]:

$$I_t = \mathrm{div}(\mathbf{D} \cdot \nabla I) \quad (3)$$

where $I_t = \frac{\partial I}{\partial t}$ denotes the derivative of the image I with respect to the time t, ∇I is the gradient vector, $\mathbf{D}$ is a square matrix called *diffusion tensor* and div is the *divergence* operator:

$$\mathrm{div}(\mathbf{f}) = \frac{\partial f_x}{\partial x} + \frac{\partial f_y}{\partial y} + \frac{\partial f_z}{\partial z}$$

In AND the smoothing depends on both the strength of the gradient and its direction measured at a local scale. The diffusion tensor $\mathbf{D}$ is therefore defined as a function of the structure tensor J:

$$\mathbf{D} = [\mathbf{v_1}\ \mathbf{v_2}\ \mathbf{v_3}] \cdot \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{bmatrix} \cdot [\mathbf{v_1}\ \mathbf{v_2}\ \mathbf{v_3}]^T \quad (4)$$

where $\mathbf{v_i}$ denotes the eigenvectors of the structure tensor. The values of the eigenvalues λ_i define the strength of the smoothing along the direction of the corresponding eigenvector $\mathbf{v_i}$. The values of λ_i rank from 0 (no smoothing) to 1 (strong smoothing). Therefore, this approach allows smoothing to take place anisotropically according to the eigenvectors determined from the local structure of the image. Consequently, AND allows smoothing on the edges: Smoothing runs along the edges so that they are not only preserved but smoothed. AND has turned out, by far, the most effective denoising method by its capabilities for structure preservation and feature enhancement [15, 5, 3, 4].

3.3 Gaussian smoothing

One of the most common ways of setting up the diffusion tensor $\mathbf{D}$ gives rise to the so-called Edge Enhancing Diffusion (EED) approach [15]. The primary effects of EED are edge preservation and enhancement. Here strong smoothing is applied along the preferred directions of the local structure, (the second and third eigenvectors, $\mathbf{v_2}$ and $\mathbf{v_3}$). The strength of the smoothing along the normal of the structure, i.e. the eigenvector $\mathbf{v_1}$, depends on the gradient: the higher the value is, the lower the smoothing strength is. Consequently, λ_i are then set up as:

$$\begin{cases} \lambda_1 = g(|\nabla I|) \\ \lambda_2 = 1 \\ \lambda_3 = 1 \end{cases} \quad (5)$$

with g being a monotonically decreasing function, such as $g(x) = 1/\sqrt{(1 + x^2/K^2)}$, where $K > 0$ acts as a contrast parameter [15]; Structures with $|\nabla I| > K$ are regarded as edges, otherwise they are considered to belong to the interior of a region. Therefore, smoothing along edges is preferred over smoothing across them, hence edges are preserved and enhanced.

3.4 Solving PDE

The diffusion equation, Eq. (3), can be numerically solved using finite differences. The term $I_t = \frac{\partial I}{\partial t}$ can be replaced by an Euler forward difference approximation. The resulting explicit scheme allows calculation of subsequent versions of the image iteratively:

$$\begin{aligned} &I^{s+1} = I^s + \\ &\tau \cdot (\tfrac{\partial}{\partial x}(D_{11} I_x) + \tfrac{\partial}{\partial x}(D_{12} I_y) + \tfrac{\partial}{\partial x}(D_{13} I_z) + \\ &\tfrac{\partial}{\partial y}(D_{21} I_x) + \tfrac{\partial}{\partial y}(D_{22} I_y) + \tfrac{\partial}{\partial y}(D_{23} I_z) + \\ &\tfrac{\partial}{\partial z}(D_{31} I_x) + \tfrac{\partial}{\partial z}(D_{32} I_y) + \tfrac{\partial}{\partial z}(D_{33} I_z)) \end{aligned} \quad (6)$$

where s is the iteration index, τ denotes the time step size, I^s denotes the image at time $t_s = s\tau$, the terms I_x, I_y,

I_z are the derivatives of the image I^s with respect to x, y and z, respectively. Finally, the D_{mn} terms represent the components of the diffusion tensor $\mathbf{D}^s$. The standard scheme to approximate the spatial derivatives ($\frac{\partial}{\partial x}$, $\frac{\partial}{\partial y}$ and $\frac{\partial}{\partial z}$) is based on central differences.

In this traditional explicit scheme for solving the partial differential equation Eq. (3), the stability is an issue [15]. The maximum time step used in the experiments carried out in this work is $\tau = 0.1$. Typically the total number of iterations ranges from 60 to 100 in 3D problems [15, 3, 4] with that value of τ.

4. AND:AN ITERATIVE MULTIPLE 3D STENCIL

The algorithm for solving the PDE in Eq. (6) using the discretization of the temporal and spatial derivatives described above can be expressed as an iterative multiple four 3D-stencils as simplified in Figure 2. The first stencil is a 7-point gauss computation, the second is a 7-point stencil that computes the structure tensor, a one-point stencil that solves an 3×3 eigenvalue system and the last stencil is a multiple stencil that solves the PDE equation.

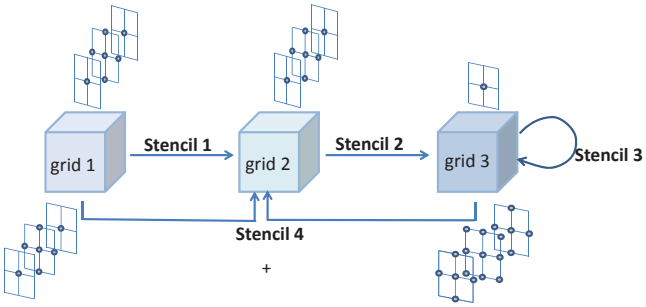


Figure 2: The four 3D-stencils involved in AND and their respective data dependence.

An efficient parallel implementation of AND for single and multicore processors consists of dividing the volume into subvolumes that fit entirely in the cache hierarchy and assign for denoising each subvolume to one core. This implementation has shown to have a high cache reutilization and low memory footprint as demonstrated in [14, 13].

AND can be seen as an iterative complex stencil where each iteration can be expressed using four 3D-stencils that operate on three 3D-arrays as plotted in Figure 2. The 3D-grids, grid 1 and grid 2, of dimension $N_x * N_y * N_z$ represent the initial noisy volume and the final denoised volume after one iteration. For the next iteration, the content of grid 2 becomes the initial volume to be denoised. grid 3 of dimension $6 \times N_x * N_y * N_z$ is used to store the structure tensor of grid 2. Table 1 sumarizes the characteristics of each one of the four stencils.

From here on stencil 1, 2, 3 and 4 will be labeled as $GAUSS$, ST, DT and PDE respectively.

- Stencil 1 ($GAUSS$), is a 7-points gauss smoothing stencil, it reads the input noisy 3D-grid of size $N_x * N_y * N_z$ and generates a different 3D-grid of the same size.

Table 1: A simplified summary of the stencils involved in AND.

Stencils	Flops/ stencil	Reads/ stencil	writes/ stencil	# of in and output grids
1	8	7	1	1 in 1 out
2	12	6	1	1 in 1 out
3	27	6	1	1 for in and out
4	69	51	1	2 in 1 out

- Stencil 2 (ST), is a 6-points stencil that calculates the structure tensor of size $6*N_x*N_y*N_z$ from the 3D-grid obtained in stencil 1.
- Stencil 3 (DT), is a in-place 1-point stencil, where each point is in fact a 3×3 symmetric matrix thus only 6 elements are stored. This stencil uses the jacobi iterative method to solve an eigen-value eigen-vector problem of the 3×3-matrix at each point of the 3D-grid.
- Stencil 4 (PDE), is a multiple, 7-points and 16-points stencils. It reads both the input 3D-grid and the diffusion tensor obtained from stencil 3 to solve the Partial Differential Equation of the discritized problem.

Fusing all these stencils together decreases the accesses to main memory and also the used space in the main memory. Theoretically, this implies an increase of the theoretical peak performance of AND but it omits factors such as recomputation of the borders which may penalize concurrency and also the synchronization needed to update the halos and borders of each stencil.

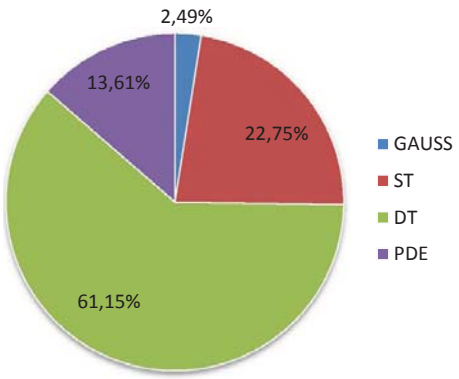


Figure 3: The weight of each stencil of AND in the total runtime.

To guide our analysis we first measured the weight of each stencil in the total runtime as plotted in Figure 3. The runtime of each stencil includes its corresponding border conditions problem. As it can be seen from this Figure, DT is the heaviest stencil, therefore, we implemented and later evaluated all the fission/fusion combinations around DT, namely, $G|ST+DT+PDE$, $G+ST+DT|PDE$, $G+ST|DT+PDE$, $G|ST+DT|PDE$, $G+ST|DT|PDE$ and $G|ST|DT|PDE$. The symbols | and + refers to fission and fusion respectively.

5. GPU IMPLEMENTATIONS

The kernels, or stencils, of all the evaluated combinations follow the same scheme. The 3D-grids are processed by tiles of dimension $DIMX \times DIMY$ along the z-axis. First, data and halos are read from global memory and copied into a tile in shared memory, then the corresponding computation is carried out and finally the output 3D-grids are updated. The main differences between each implementation are the number of accesses to main memory, the used space in main memory, the used number of registers and space of shmem.

Table 2: Global memory accesses for read and update in/out 3D-grids

implementation	global memory accesses per implementation per kernel
$G\|ST+DT+PDE$	g1 g2 | g1 g2
$G+ST+DT\|PDE$	g1 g3 | g1 g3 g2
$G+ST\|DT+PDE$	g1 g3 | g1 g3 g2
$G\|ST+DT\|PDE$	g1 g2 | g2 g3 | g1 g3 g2
$G+ST\|DT\|PDE$	g1 g3 | g3 g3| g1 g3 g2
$G\|ST\|DT\|PDE$	g1 g2 | g2 g3 | g3 g3 | g1 g3 g2

g1, g2, g3 refers to the input 3D-grid (labeled as grid 1 in Figure 2), output 3D-grid (labeled as grid 2 in Figure 2) and the structure tensor(labeled as grid 3 in Figure 2) used in the implementations. tile1, tile2 and tile3 are the auxiliary arrays used in shmem to store tiles from grid 1, grid 2 and grid 3 respectively.

Table 3: Shared memory usage per implementation per kernel

implementation	shmem usage per implementation per kernel
$G\|ST+DT+PDE$	tile1 | 4*tile1 12*tile3
$G+ST+DT\|PDE$	4*tile1 3*tile2 | 3*tile1 6*tile3
$G+ST\|DT+PDE$	4*tile1 3*tile2 | 3*tile1 6*tile3
$G\|ST+DT\|PDE$	tile1 | tile2 | 3tile1 6*tile3
$G+ST\|DT\|PDE$	4*tile1 | | 3*tile1 6*tile3
$G\|ST\|DT\|PDE$	tile1 | tile2 | | 3*tile1 6*tile3

6. EXPERIMENTAL ANALYSIS OF STENCILS FISSION/FUSION IN AND

6.1 Additional optimizations

A performance tuning was used before measuring the impact of fission/fusion transformation on the performance of AND. In particular, register blocking with an appropriate loop unrolling was used in the kernels when it is feasible and when it improves the performance. No padding is required since all the used volume sizes are multiple of 128. Experimentally, we found that synchronizing the threads at one block level before and after solving the eigen-value eigen-vector problem in stencil 3 improves drastically its performance since it conducts to coalescing accesses to memory. We also used the mathematical functions implemented by CUDA, (e.g., fabsf and sqrtf) by using the compilation option -use_fast_math

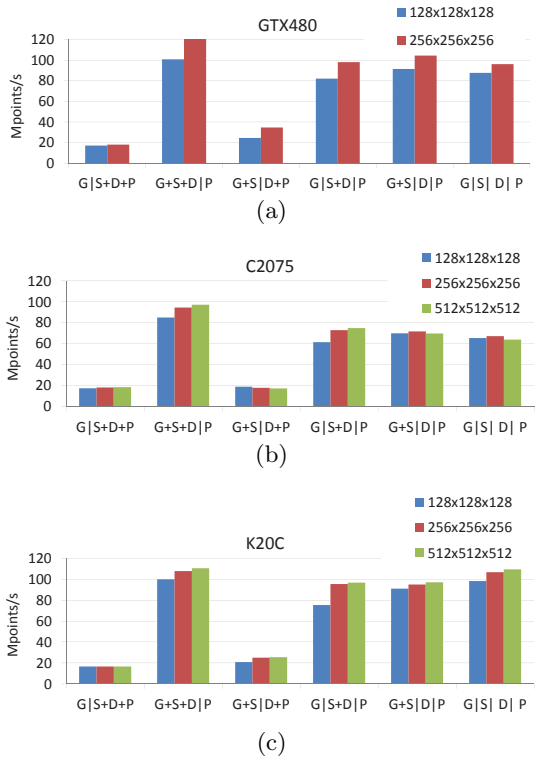


Figure 4: The throughput in Mpoints/s of all the fussion/fusion combinations, $G|ST+DT+PDE$, $G+ST+DT|PDE$, $G+ST|DT+PDE$, $G|ST+DT|PDE$ and $G|ST|DT|PDE$ on GTX489 , C2075 and K20C.

6.2 Fission/fusion transformation impact

This section provides and analyzes the performance evaluations of the 6 combinations, $G|ST+DT+PDE$, $G+ST+DT|PDE$, $G+ST|DT+PDE$, $G|ST+DT|PDE$, $G+ST|DT|PDE$ and $G|ST|DT|PDE$. Recall that we consider all the combinations except the full fused version $G+ST+DT+PDE$. In fact, the experimental results shown in this section confirm that fusing the three heaviest stencils together leads to a very poor performance and thus fusing GAUSS with this version will leads to similar performance. The experimental setup is summarized in Table 4.

Table 4: Brief summary of the GPUs, configuration and compilers used in the experiments. *The bandwidth Nvidia benchmark was used.

	C2070	K20C	GTX 480
Peak GFlops (single precision)	1288	3520	1816
Bandwidth* (GB/s)	84.8	143.9	117.7
# SMs	14	13	8
# Cores/SM	32	192	32
Clock Speed (GHz)	1.15	0.71	1.50
shmem/block (Kb)	48	48	48
# Reg/block	32768	65536	32768
Global memory (Gb)	5.24	4.68	1.45
ECC enabled	Yes	Yes	Yes
Used CUDA Compiler	4.0	5.0	4.0

All the CUDA-implementations are written in CUDA C and deal with single precision float datatype.

Figure 4(a), (b) and (c) show the throughput expressed in Mpoints/s, calculated as $N_x * N_y * N_z/time(s)$, of the combinations $G|ST + DT + PDE$, $G + ST + DT|PDE$, $G + ST|DT + PDE$, $G|ST + DT|PDE$, $G + ST|DT|PDE$ and $G|ST|DT|PDE$ on the three GPUs described in Table 1, and using three 3D-arrays of sizes $128 \times 128 \times 128$, $256 \times 256 \times 256$ and, $512 \times 512 \times 512$. The used thread blocks are of size 16×16. From the figures, one can observe that the combination $G + ST + DT|PDE$ provides the best performance over all the combinations and on all the GPUs especially on GTX and C2075. $G + ST + DT|PDE$ is up to $6\times$ better than $G|ST + DT + PDE$, and up to $1.52\times$ better than the fully fission version $G|ST|DT|PDE$.

In particular, $G + ST + DT|PDE$ and $G|ST|DT|PDE$ are both versions that provide near and sometimes competitive performance. $G + ST + DT|PDE$ is $1.3\times$, $1.4\times$ and $1.52\times$ faster than $G|ST|DT|PDE$ for the 3D-grids of sizes $128 \times 128 \times 128$, $256 \times 256 \times 256$, and $512 \times 512 \times 512$ respectively on C2070. $G + ST + DT|PDE$ is $1.14\times$ and $1.28\times$ faster than $G|ST|DT|PDE$ for the 3D-grids of sizes $128\times128\times128$, $256\times256\times256$ respectively on GTX 480. The implementation $G|ST|DT|PDE$ could not be evaluated on GTX 480 for the volume size equals $512\times512\times512$ because of memory space limitations. On Kepler, the implementations $G + ST + DT|PDE$ and $G|ST|DT|PDE$ offer very competitive throughput for the evaluated volumes. This fact can be explained by the large bandwidth and high number of registers in K20C.

To further understand these differences we measured the overhead due to updating halos and borders and also to the necessary re-computation of halos in versions $G|ST + DT + PDE$ and $G + ST + DT|PDE$. Figure 5 shows this overheads and their weights versus the weight of real useful work. Although, the combination $G|ST + DT + PDE$ has a large data reutilization and few accesses to global memory (only to read and write input and output volume), its performance is strongly penalized by the large overhead which means a lower occupancy. Whereas, version $G + ST + DT|PDE$ presents a good balance between useful work and overhead. Figure 5 represent the overhead and useful work for the volume of size $256 \times 256 \times 256$ on C2070. The plots of the rest of the volumes on all the GPUs are very similar.

In addition, we carried out a profiling of DT and PDE stencils as the combination of this two stage is the reason behind the poor performance of the implementations $G|ST + DT + PDE$ and $G + ST|DT + PDE$. We found that PDE is register bound and shows a very low active warps per active cycle, of the order of 15, while DT shows a much better concurrency level, around 25 active warps per active cycle for volume size equals $256 \times 256 \times 256$. As result, combining a stencil with poor concurrency with a stencil with higher concurrency penalizes the performance of the last one.

In summary, applying the multi-fission transformation by implementing 3D-stencils with a high registers and shemm pressure into one kernel and fusing kernels with similar concurrency provides the best compromise reutilization, concurrency and overheads.

7. CONCLUSIONS

This paper evaluates the impact of a multi-fission transformation on the performance of an iterative multiple 3D-stencil on Nvidia GPUs. In particular, we analyzed the fission / fusion transformation of AND, a representative method of PDE-based algorithms used in signal processing. Our results validate that fission improves the performance of our application on the GPU. A remarkable result is that the fission of 3D-stencils with lower concurrency and fusion of 3D-stencils with high and similar concurrencies provide the best trade-off between locality and concurrency on GPUs, especially on Fermi. For the case of AND, the two-kernels combination that fuses the three single 3D-stencils into one kernel and the multiple stencil into one kernel is $1.52\times$ faster than full fission (four kernels).

We have identified two main directions for future work: (1) indicating the best decomposition of complex stencils at design time, and (2) including our fission approach in a more general-purpose framework for tuning stencil codes on GPUs.

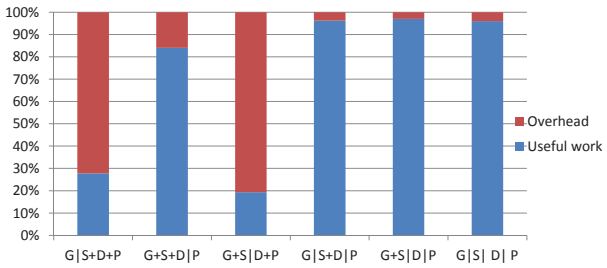


Figure 5: The overhead due to synchronization and memory accesses necessary to copy halos and borders for the six combinations $G|ST + DT + PDE$, $G+ST+DT|PDE$, $G+ST|DT+PDE$, $G|ST+DT|PDE$, $G + ST|DT|PDE$ and $G|ST|DT|PDE$ on GTX480, C2075 and K20C. The overheads of combinations $G|ST+DT+PDE$ and $G+ST+DT|PDE$ include also re-computation of some halos.

8. REFERENCES

[1] D. Barash. Fundamental relationship between bilateral filtering, adaptive smoothing, and the nonlinear diffusion equation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 2002.

[2] J.-J. Fernandez. Tomobflow: feature-preserving noise filtering for electron tomography. *BMC bioinformatics*, 10(1):178, 2009.

[3] J.-J. Fernández and S. Li. An improved algorithm for anisotropic nonlinear diffusion for denoising cryo-tomograms. *Journal of structural biology*, 144(1):152–161, 2003.

[4] J.-J. Fernandez and L. Sam. Anisotropic nonlinear filtering of cellular structures in cryoelectron tomography. *Computing in science & engineering*, 7(5):54–61, 2005.

[5] A. S. Frangakis and R. Hegerl. Noise reduction in electron tomographic reconstructions using nonlinear anisotropic diffusion. *Journal of structural biology*, 135(3):239–250, 2001.

[6] A. S. Frangakis, A. Stoschek, and R. Hegerl. Wavelet transform filtering and nonlinear anisotropic diffusion assessed for signal reconstruction performance on

multidimensional biomedical data. *Biomedical Engineering, IEEE Transactions on*, 48(2):213–222, 2001.

[7] A. Haidar, H. Ltaief, P. Luszczek, and J. Dongarra. A comprehensive study of task coalescing for selecting parallelism granularity in a two-stage bidiagonal reduction. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 25–35, May.

[8] J. M. J. Fehrenbach. Small non-negative stencils for anisotropic diffusion. *arXiv:1301.3925, Numerical Analysis*, 2013.

[9] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[10] P. Micikevicius. 3d finite difference computation on gpus using cuda. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84. ACM, 2009.

[11] A. G. Navarro, R. Asenjo, S. Tabik, and C. Cascaval. Analytical modeling of pipeline parallelism. In *PACT*, pages 281–290, 2009.

[12] M. Rumpf and R. Strzodka. *Nonlinear diffusion in graphics hardware*. Springer, 2001.

[13] S. Tabik, E. M. Garzón, I. García, and J.-J. Fernández. Implementation of anisotropic nonlinear diffusion for filtering 3d images in structural biology on smp clusters. In *Proc. Int. Conf. Parallel Computing: Current & Future Issues of High-End Computing, ParCo*, volume 33, pages 727–734, 2005.

[14] S. Tabik, E. M. Garzón, I. García, and J.-J. Fernández. High performance noise reduction for biomedical multidimensional data. *Digital Signal Processing*, 17(4):724–736, July 2007.

[15] J. Weickert. *Anisotropic diffusion in image processing*. Teubner Stuttgart, 1998.

[16] H. Wu, G. Diamos, J. Wang, S. Cadambi, S. Yalamanchili, and S. Chakradhar. Optimizing data warehousing applications for gpus using kernel fusion/fission. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2433–2442. IEEE, 2012.

[17] Y. Zhao. Lattice boltzmann based pde solver on the gpu. *The Visual Computer*, 24(5):323–333, 2008.

Optimizing Stencil Computations for NVIDIA Kepler GPUs

Naoya Maruyama
RIKEN Advanced Institute for Computational Science
Kobe, Japan
nmaruyama@riken.jp

Takayuki Aoki
Tokyo Institute of Technology
Tokyo, Japan
taoki@gsic.titech.ac.jp

ABSTRACT

We present a series of optimization techniques for stencil computations on NVIDIA Kepler GPUs. Stencil computations with regular grids had been ported to the older generations of NVIDIA GPUs with significant performance improvements thanks to the higher memory bandwidth than conventional CPU-only systems. However, because of the architectural changes introduced with the latest generation of the GPU architecture, Kepler, we show that existing implementation strategies used for such older GPUs are not as effective on Kepler as before. To fully exploit the potential performance of the latest generation of the GPU architecture, our implementation method uses shared memory for better data locality combined with warp specialization for higher instruction throughput. Our method achieves approximately 80% of the estimated peak performance by the roofline model, and even higher performance with temporal blocking.

1. INTRODUCTION

GPU accelerators are increasingly used in various fields of scientific simulations because of superior performance and power efficiency. The peak theoretical memory bandwidth of a single NVIDIA Kepler K20X GPU reaches up to 250 GB/s. Such high memory system performances are particularly important for many of stencil kernels, which are typically memory intensive. In fact, past studies reported that stencil computations with regular Cartesian grids can be ported to GPU with significant performance acceleration compared to conventional CPU-only systems [10, 11].

Effective optimization techniques for stencils include locality optimization by loop blocking. On GPUs, there are several different on-chip memories that could be used for blocking. Shared memory is a programmable memory that can be accessed with very low latency, and has been used for blocking in many past studies [1, 6]. It was particularly important to optimize GPU programs with shared memory blocking when using older generations of NVIDIA GPUs; however, we have also observed that such manual optimizations are not always necessary with recent GPU architecture called Fermi, where DRAM loads are cached at L1 cache. Although GPU cache size is typically very small, it is often very effective for regular data access patterns in stencils with Cartesian grids, allowing good performance even without manual shared memory blocking [11].

One of the major architectural changes in Kepler, the most recently released GPU architecture by NVIDIA, is that L1 cache is not used for DRAM load caching, but only used for register spilling [9]. Therefore, stencil kernels written for Fermi GPUs with an assumption that redundant localized data accesses are automatically cached at L1 cache may not perform as efficiently on Kepler as on Fermi. Since the new architecture also introduced a number of major and minor changes in hardware configurations, it is important to examine stencil kernel performance on Kepler GPUs with particular focus on memory access optimizations.

This paper evaluates the performance of a 7-point 3-D stencil on Fermi and Kepler with a series of memory access optimizations.[1] We first begin with a baseline program that is written in a straightforward way, and apply loop blocking at registers and shared memory. We also extend the stencil with temporal blocking for further saving DRAM accesses [4, 7, 12, 14, 15]. While these manual blocking optimizations with shared memory should in theory improve performance in memory-bound stencil computations, its effectiveness may by limited by the constraints imposed by the GPU compute architecture and limited on-chip memory capacity. Although stencils with Cartesian grids mostly consist of regular computation patterns, irregular processing of halo regions can have large performance impacts due to the wide-vector execution model. In fact, several past studies reported that temporal blocking is not effective for 3-D stencil problems [4, 15]. We study several implementation techniques for shard memory blocking, including those using texture memory and warp specialization, and attempt to answer the following questions:

- How does stencil programs optimized for Fermi perform on Kepler?
- How much performance gain can be obtained with shared memory blocking?
- How close to peak performance can be obtained in stencil computations on GPUs?

[1] All program code is available for download at `http://github.com/naoyam`.

HiStencils 2014
First International Workshop on High-Performance Stencil Computations
January 21, 2014, Vienna, Austria
In conjunction with HiPEAC 2014.

`http://www.exastencils.org/histencils/2014/`

```
1   // trip count of the time loop
2   int count;
3   // 3-D arrays of size nx*ny*nz
4   float f1[nx,ny,nz];
5   float f2[nx,ny,nz];
6   // coefficient variables
7   float cc, cw, ce, cs, cn, cb, ct;
8
9   // Time loop executing for count iterations
10  repeat count times
11    // update all grid points from f1 to f2
12    for (x,y,z) in nx*ny*nz
13       // 7-point stencil where accessing exterior
14       // points is replaced with nearby boundary points
15       int w = (x == 0)    ? 0 : -1;
16       int e = (x == nx-1) ? 0 : 1;
17       int n = (y == 0)    ? 0 : -1;
18       int s = (y == ny-1) ? 0 : 1;
19       int b = (z == 0)    ? 0 : -1;
20       int t = (z == nz-1) ? 0 : 1;
21       f2[x,y,z] = cc*f1[x,y,z] + cw*f1[x+w,y,z]
22           + ce*f1[x+e,y,z] + cs*f1[x,y+s,z]
23           + cn*f1[x,y,z+n] + cb*f1[x,y,z+b]
24           + ct*f1[x,y,z+t];
25    swap f1, f2;
26    end for
27  end repeat
```

Figure 1: Pseudo code for the 7-point diffusion stencil.

Our performance studies using both Fermi and Kepler GPUs reveal that the shared memory blocking with warp specialization is highly effective in achieving optimal performance on Kepler, but not on Fermi. Common implementation techniques to use the shared memory, such as [6] and [10], however, is shown to be less effective. We also show that temporal blocking with warp specialization can further improve the performance. Overall, our optimized implementations achieve approximately 80% of the estimated peak performance by the roofline model, and even higher performance with temporal blocking on Kepler. While the studies in this paper are limited to a 7-point stencil code, we believe that our findings would give a useful guideline for optimizing other stencils for NVIDIA GPUs.

2. TARGET STENCIL

As an example of stencil, we use a simple 7-point stencil that computes a diffusion equation on 3-D domains of single-precision floating-point data. Figure 1 shows its pseudo code. Notice that each point is updated by accessing six nearest neighbor points as well as its own previous value (i.e., `f1[x,y,z]`) with an exception for boundary points. For each boundary point, neighbor access that falls outside the defined grid area is replaced by its own value (lines 15-20). In this paper, we evaluate and optimize the performance of this stencil on a single GPU device.

The performance of the stencil for a given architecture can be estimated based on the roofline model [13]. As shown in the pseudo code, an update of a single point requires 13 single-precision FP operations, so each iteration performs $13 \times \texttt{nx} \times \texttt{ny} \times \texttt{nz}$ operations. To simplify modeling of DRAM accesses, we assume that for each iteration once array points are loaded from memory, they remain on cache for that iteration. This is an optimistic assumption for problem sizes larger than the last-level cache size, which is typically the case with reasonable problem sizes for realistic computational fluid simulations. Based on the assumption, the size of data loaded and stored per iteration is modeled as $\texttt{nx} \times \texttt{ny} \times \texttt{nz} \times \texttt{sizeof(float)} * 2$ bytes. Thus, the compute intensity, defined as flop per DRAM byte, is calculated as $\frac{13 \times \texttt{nx} \times \texttt{ny} \times \texttt{nz}}{\texttt{nx} \times \texttt{ny} \times \texttt{nz} \times \texttt{sizeof(float)} * 2} = 1.625$.

The intensity clearly indicates that the performance of the stencil on a GPU is bounded by its DRAM bandwidth. For example, the performance of the latest NVIDIA GPU Kepler is as high as 4 TFLOPS (single precision) [9], whereas the theoretical peak memory bandwidth is 250 GB/s. Optimizing memory accesses, therefore, is the most important strategy in improving the performance of the stencil. In particular, although the above modeling assumes the perfect data locality that all grid points in array `f1` are loaded from DRAM at most once and that subsequent accesses are from the cache, the constraints imposed by the GPU architecture require non-trivial tuning as presented in this paper.

3. OVERVIEW OF KEPLER GPU ARCHITECTURE

This section gives a brief overview of the NVIDIA Tesla Kepler GPU [9], which is the main target of our study in this paper. We particularly focus on the differences between Kepler and the previous generation of NVIDIA GPU architecture called Fermi. Since the specific Kepler GPU used in this paper is Tesla K20X, we first present a brief overview of K20X, and explain how memory accesses in CUDA are mapped to actual data movements in the architecture.

The K20X GPU consists of a GK110 processor equipped with 6 GB of GDDR5 memory. The GK110 processor in K20X consists of 14 streaming multiprocessors (SM), each of which consists of 192 CUDA cores clocked at 732 MHz, achieving 3.95 TFLOPS in single-precision peak performance. Each SM has an on-chip memory area of 64 KB that can be accessed both explicitly as shared memory and implicitly as L1 cache. It also has a register file of 256 KB, which is doubled from Fermi. In addition to these memories, the GK110 processor is equipped with a read-only cache of 48 KB per SM, which is an extension of the texture cache available in the older generations of GPUs, but becomes more easily accessible in Kepler. Furthermore, 1.5 MB of L2 cache is shared by all 14 SMs.

Each memory access to CUDA global memory is first serviced by the off-chip GDDR memory. The same coalescing rule as enforced in Fermi is still applicable in Kepler [8]. However, a significant change in memory accesses is that global memory loads and stores are not cached in the L1 cache on Kepler, whereas on Fermi load accesses are cached by default. We have observed that the Fermi L1 cache is often very effective for reducing the DRAM bandwidth pressure in stencil computations with regular grids, making explicit blocking with the shared memory almost unnecessary in such computations. In fact, our optimized CUDA implementation of a phase-field simulation application [11], which won a 2011 Gordon Bell Award, did not use shared memory since we saw no performance benefit on Fermi GPUs. The same implementation, however, would not perform as efficiently on Kepler as on Fermi since global memory loads are no longer cached.

Instead of relying on automatic caching with the L1 cache, the shared memory and read-only cache can be used on Kepler, both of which require code modifications. In CUDA,

```
1  __global__ void baseline(float *f1, float *f2,
2                           int nx, int ny, int nz,
3                           float ce, float cw, float cn,
4                           float cs, float ct, float cb,
5                           float cc) {
6    int xy = nx * ny;
7    int i = blockDim.x * blockIdx.x + threadIdx.x;
8    int j = blockDim.y * blockIdx.y + threadIdx.y;
9    const int block_z = nz / gridDim.z;
10   int k = block_z * blockIdx.z;
11   const int k_end = k + block_z;
12   int c = i + j * nx + k * xy;
13 #pragma unroll
14   for (; k < k_end; ++k) {
15     int w = (i == 0)        ? c : c - 1;
16     int e = (i == nx-1)     ? c : c + 1;
17     int n = (j == 0)        ? c : c - nx;
18     int s = (j == ny-1)     ? c : c + nx;
19     int b = (k == 0)        ? c : c - xy;
20     int t = (k == nz-1)     ? c : c + xy;
21     f2[c] = cc * f1[c] + cw * f1[w] + ce * f1[e] + cs * f1[s]
22         + cn * f1[n] + cb * f1[b] + ct * f1[t];
23     c += xy;
24   }
25   return;
26 }
```

Figure 2: Baseline stencil kernel.

the shared memory can be explicitly used as a scratchpad memory by annotating arrays with the `__shared__` attribute. The read-only cache can be explicitly used by the `__ldg` intrinsic or the CUDA compiler automatically uses the read-only cache when arrays are declared with the `const` and `__restrict__` annotations. We present our optimization techniques using those memories in Section 5.

4. BASELINE IMPLEMENTATION AND PERFORMANCE

A straightforward CUDA implementation of the stencil is shown in Figure 2. We use this code as the baseline for subsequent performance evaluations and optimizations. It partitions the 3-D domains of size `nx` × `ny` × `nz` into sub domains of `blockDim.x` × `blockDim.y` × `nz`/`gridDim.z`, each of which is computed by a CUDA thread block. Note that we assume that the z dimension is the slowest varying dimension. As shown in the code, the grid points in the x-y planes are computed in parallel by the threads in a thread block, whereas the computation over the z-direction is swept sequentially.

The performance of the code on Fermi M2075 and Kepler K20X GPUs are shown in Figure 3. The size of the grid is 256^3. The CUDA version used in our experiments is version 5.0 on Linux kernel 2.6.32 (CentOS 6.4). We only evaluate the kernel execution time excluding the PCI data transfer cost. We configure the L1 cache and shared memory partitioning to 48 KB and 16 KB, respectively. The measurement is done five times and the fastest performance is reported. In addition to the baseline performance, the figure shows the upper bound performance that is estimated with the roofline performance model. The measured bandwidth of the Fermi and Kepler GPUs are 103 GB/s and 170 GB/s, respectively. As discussed in Section 2, the compute intensity of the kernel is 1.625, indicating the peak attainable performances are 167 GLFOPS and 276 GFLOPS, respectively.

As shown in the graph, the actual performances of the baseline version has relatively large gap compared to the

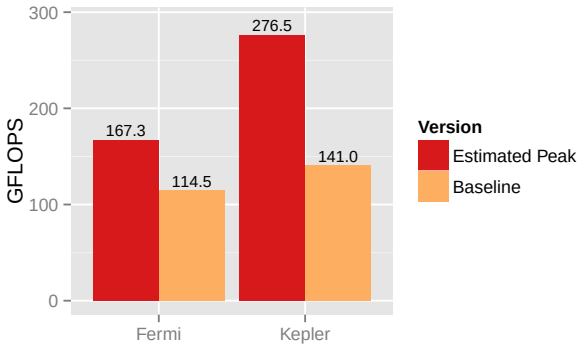


Figure 3: Performance of the baseline implementation on Fermi M2075 and Kepler K20X.

model-estimated peak performances especially on Kepler. The ratios against the peak for Fermi and Kepler are 66.7% and 49.7%, respectively.

5. OPTIMIZATIONS

The large gap with the peak performance indicates that the baseline kernel still has a large room for improvements. This section presents a series of optimizations focused on minimizing off-chip memory accesses. Specifically, we first explain a set of basic optimizations, including loop peeling and register blocking. Next, we apply spatial and temporal blocking to exploit the on-chip memories available on the GPU.

5.1 Basic Optimizations

We optimize the computation by moving the conditional operations out of the loop. Notice that the conditional evaluations on variables `i` and `j` are loop independent, so they can be safely moved before the loop. The conditional operations on variable `k` are loop dependent, but peeling the first and last iterations allows us to remove the conditional operations from the loop inside. It also uses registers to cache the points along the Z dimension since they are reused locally by each thread [2].

5.2 Spatial Blocking with Read-Only Cache

To use the read-only cache on Kepler, we add `const` and `__restrict__` annotations to the input grid parameter declaration and `__restrict__` to the output parameter. This change should not affect actual native code for Fermi, but when compiled for Kepler the compiler should generate code that uses the read-only cache for all loads from the input grid.

5.3 Spatial Blocking with Shared Memory

Another well-known opportunity for reducing the DRAM pressure is explicit blocking using the shared memory. As previously presented by, e.g., [6, 10], each thread block allocates a 2-D chunk of shared memory for its corresponding sub domain and its halo data. Every thread stores its point value to the shared memory, and the boundary threads that are located on the boundary of the sub domain are also responsible for loading the halo data from the global memory to the shared memory. Since the data reuse between

threads in a thread block only exists for the single X-Y plane with the z offset being zero, we only need one 2D plane of $(\texttt{bx}+2) \times (\texttt{by}+2)$ in the shared memory, where `bx` and `by` represent the dimension of a sub domain. We also change the configuration of shared memory and L1 cache sizes as 48 KB and 16 KB, respectively.

A potential problem with this implementation is that loading halo data needs conditional operations, which are executed every iteration. In particular, the accesses to the Y-direction halo points cause branch divergence since only the two boundary threads among their warps are involved. Therefore, although it appears that only those boundary threads issue the global memory load and shared memory store instructions, the execution overhead can be significant because every thread in the same warp also issue the same instructions. In fact, as shown in Section 6, this version turned out to be less efficient than the above versions.

Phillips and Fatica presented an optimization method that does not use conditional operations by exploiting the texture memory [10]. All threads in a thread block redundantly reads the global memory four times, each of which is diagonally shifted to account for the halo points. The cost of increased number of read transactions and mis-aligned accesses is minimized by using the texture memory. We also evaluate this method with our stencil on Fermi and Kepler.

5.4 Spatial Blocking with Warp Specialization

We further extend the shared memory optimization by using a programming technique called warp specialization [1]. Since warp divergence does not happen if all threads in a warp take the same control flow, the overhead by the conditional operations and branch divergence can be minimized by a careful assignment of warps based on control flows as described below.

We first divide the execution of a thread block to three control flows: one for the interior points, another for the Y-direction boundaries, and another for the X-direction boundaries. This is realized in our CUDA kernel by creating three disjoint code paths that correspond to the three flows. The conditional branch to select each path is done only once at the beginning of the kernel, so the cost of branches is eliminated. Furthermore, since the warps for the interior points are not responsible for loading the X-direction halo points, the branch divergence caused in the previous version is completely eliminated.

A drawback of the warp specialization is the increased number of threads per thread block, which has two performance implications. First, it requires a larger number of registers per thread block, which can adversely affect the number of active threads running on each SM. Since latency hiding by a large number of active threads is one of the most important optimizations for achieving high memory throughput, increase of the register usage can result in lower overall performance. Second, it could also increase the cost of thread synchronization within a thread block. Since the blocking with the shared memory needs two times of thread synchronization per iteration, the increased cost of thread synchronization can also reduce the overall performance.

Another drawback of this approach is the increased cost of code development. Even though some part of the separate warp groups perform common operations,

5.5 Temporal Blocking with Shared Memory

Finally, we study the effectiveness of temporal blocking using the shared memory [3, 4, 7, 12, 14, 15]. It has been shown to be particularly effective for bandwidth-bound computations such as stencils.

Similar to the shared memory version, we use warp specialization to minimize the cost of halo processing. In this present work, we only focus on two-way blocking; further aggressive blocking remains to be studied. Also for simplicity we use overlapped tiling [4] that incurs redundant overlapped grid points to update for saving DRAM memory accesses. We speculate that the increased compute cost due to overlapped tiling can be justified on GPUs, which has very high flops and bandwidth ratios.

Figure 4 illustrates a flow of each thread block performing two time steps with only one global memory read and one write. The colors represent the groups of warps. Filled regions are areas that are updated by the stencil, while the dotted regions are only loaded from the global memory for updating neighbor points. As indicated in the figure, each thread block uses a 2-D plane of shared memory region of size $(\texttt{bx}+4) \times (\texttt{by}+2)$, which is updated in place once, and then stored back to global memory after another update.

As shown in the figure, we use four groups of warps: one for the interior points (blue region), one for each of the horizontal and vertical halo regions (red and yellow regions), and another for the four diagonal points (green region). The number of actual warps for each group depends on the actual size of the sub domain, although the green region always uses just one warp. Note that since only four points are accessed in the green region, only four threads are used among the 32 threads.

The overhead due to this blocking is two-fold. First, it increases the computation cost since additional $2(\texttt{bx} \times \texttt{by} + \texttt{bx} \times \texttt{bz} + \texttt{by} \times \texttt{bz})$ points need to be updated when performing two time steps for each of $\texttt{bx} \times \texttt{by} \times \texttt{bz}$ sub domains. We expect that this overhead would be negligible given the low compute intensity of the stencil. Second, compared to the non temporal blocking version, it needs to load additional grid points in the dotted region. Assume that the blue region is perfectly aligned to a line-size boundary and that the horizontal dimension is the stride-one dimension. In this case, the accesses to the yellow dotted regions in fact has no additional cost compared to the non temporal-blocked version, since the dotted regions should also be covered the same cache lines as the filled region. Overall, the size of the data that are additionally loaded consists of the red region ($\texttt{bx} \times \texttt{bz} \times 2 \times \texttt{sizeof(float)}$ bytes), the bottom and top blue regions ($\texttt{bx} \times \texttt{by} \times 2 \times \texttt{sizeof(float)}$ bytes), and the green regions ($\texttt{bz} \times 4 \times 128$ bytes). Note that the cache line size in NVIDIA GPUs is 128 bytes. Therefore, the reduction of DRAM accesses can be estimated as: $0.5 + \frac{\texttt{bx} \times \texttt{bz} + \texttt{bx} \times \texttt{by} + \texttt{bz} \times 64}{\texttt{bx} \times \texttt{by} \times \texttt{bz} \times 2}$.

6. EXPERIMENTAL EVALUATIONS

To evaluate the effectiveness of the stencil optimizations, we measure the performance of the stencil on Fermi M2075 and Kepler K20X GPUs. We use CUDA version 5.0 on Linux 2.6.32 (CentOS 6.4). Since this paper only focuses on intra-GPU performance optimizations, we only measure the GPU kernel performance, excluding the cost of PCI data transfers. The reported performance numbers are based on the measured timing that and the cost analysis presented in

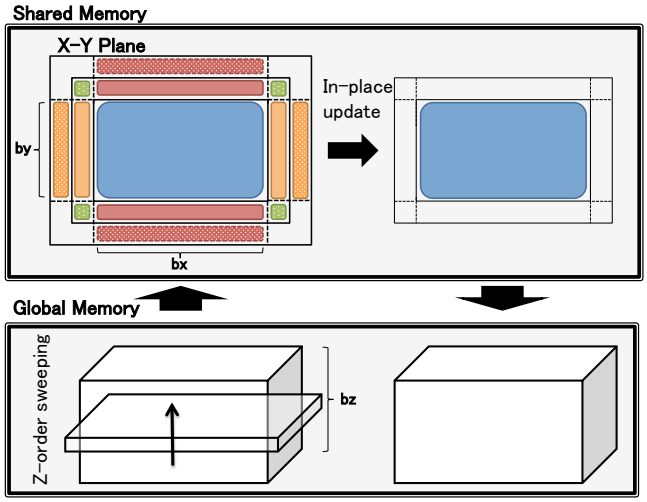


Figure 4: Temporal blocking execution flow. The filled regions are updated by the stencil, while the dotted regions are only read from global memory for updating neighbor points. Each color represent a group of warps for the same execution flow.

Table 1: Summary of employed blocking methods.

	Register	RO cache	Shared memory	Temporal
Baseline				
OPT	✓			
ROC	✓	✓		
SHM	✓		✓	
SHM-TEX	✓		✓	
SHM-WARP	✓		✓	
TEMP	✓		✓	✓

Section 2. We measure execution time of 100 time steps of each kernel for five times and use the fastest one to minimize system noise.

We evaluate five variations of optimized kernels. Specific blocking methods used in each kernel is summarized in Table 1. Note that the baseline version is the same as presented in Section 4. The Opt kernel uses the optimizations described in Section 5.1; the ROC kernel uses the read-only cache as described in Section 5.2; the SHM kernel uses the shared memory blocking described in Section 5.3; the SHM-TEX and SHM-WARP also use the shared memory blocking but uses the texture-based DRAM loads and warp specialization; the TEMP kernel is the one with the temporal blocking. The ROC kernel result is only reported for Kepler since the read-only cache is not available in Fermi GPUs. The sizes of CUDA thread block and grid of each kernel as well as its register and shared memory size are shown in Table 2. The block sizes are selected among several valid configurations by an empirical search.

Figures 5 and 6 show the performance of each version on Fermi and Kepler, respectively. The dashed line shows the possible peak performance estimated based on the roofline model discussed in Section 2. Note that it does not account for the DRAM transaction reduction by temporal blocking, it can be surpassed by the optimized version with temporal blocking. The best performances on the Fermi and Kepler GPUs are 131 GFLOPS and 287 GFLOPS, which are obtained with the OPT and TEMP kernels, respectively.

The results with the Fermi GPU shows that explicit blocking with shared memory does not yield performance improvements. Without temporal blocking, all explicit shared memory blocking kernels exhibited degraded performances by approximately 40%. The DRAM transaction saving by temporal blocking allowed the TEMP kernel to achieve much better performance than the SHM kernels; however, it was still slightly behind the kernel with no shared memory blocking. In fact, the limited effectiveness of shared memory blocking for stencils on the GPU match with our past experiences.

In contrast to the Fermi results, the shared memory optimizations were able to achieve performance improvements with varying degrees. The performance of the SHM-WARP and TEMP kernels were 232 GFLOPS and 287 GFLOPS, achieving the speedup of 1.65 and 2.05 times compared to the baseline. The reason of the better effectiveness of the SHM-WARP on Kepler still remains to be studied; however, we speculate that other architectural changes such as the increased number of registers and twice the size of L2 cache would allow for minimizing the cost of warp specialization. Further detailed studies will be reported in the future.

Although the temporal blocking version did achieve the better performance than the non temporal blocking versions, its improvement turned out not to be as large as expected. More specifically, the speedup of the TEMP over SHM-WARP kernels is only 1.23 times. Our preliminary analysis with CUDA Performance Profiler indicates that the cost of thread synchronization is the largest performance bottleneck. More detailed analysis is a subject of our ongoing study.

Another interesting finding is that the performance gap between the baseline and most optimized versions is much larger on Kepler than on Fermi, which implies that the importance of code optimizations for stencils is more profound. While achieving comparable performance as the TEMP kernel would be fairly challenging for more complex kernels than the simple stencil studied in this paper, the optimization techniques used in the ROC kernel are relatively straightforward to apply. In fact, our stencil code generation framework is already able to apply most of the ROC optimizations automatically [5].

7. RELATED WORK

Stencil computations with regular grids have been one of the extensively studied types of computations on GPUs because many of such computations can be improved by taking advantage of the GPU DRAM bandwidth. Micikevicius reported an implementation method of finite difference stencils in CUDA with shared memory blocking [6]. Phillips and Fatica reported a CUDA implementation of Jacobi kernel for multiple GPUs with MPI. They also presented blocking methods with the shared memory and texture memory. Datta et al. [2] studied stencil performances and optimizations on various multi-core and accelerator architectures. Shimokawabe et al. presented a highly scalable phase-field simulation based on stencil computations [11].

The most common optimizations used in these past studies are blocking at registers and shared memory. Shared memory blocking was particularly important in pre-Fermi generations of NVIDIA GPUs since they were not equipped with hardware cache memories. However, since the Fermi

Table 2: Details of kernel configurations and resource usage. The values of the TB and Z block columns show the empirically-found optimal 2-D thread-block size and the Z dimension of CUDA grid for each kernel. The number of registers and shared memory are per-thread and per-block resource usage, respectively.

Kernel	Fermi (SM 2.0)				Kepler (SM 3.5)			
	TB size	Z block	# registers	Shared memory (bytes)	TB size	Z block	# registers	Shared memory (bytes)
Baseline	128x1	4	23	0	128x1	16	28	0
OPT	128x1	8	28	0	128x1	8	30	0
ROC					128x1	8	30	0
SHM	128x2	32	27	2080	128x2	16	30	2080
SHM-TEX	64x2	256	32	1056	64x2	1	33	1056
SHM-WARP	64x4	128	27	1056	32x4	1	32	544
TEMP	32x8	16	24	1440	32x8	8	29	1440

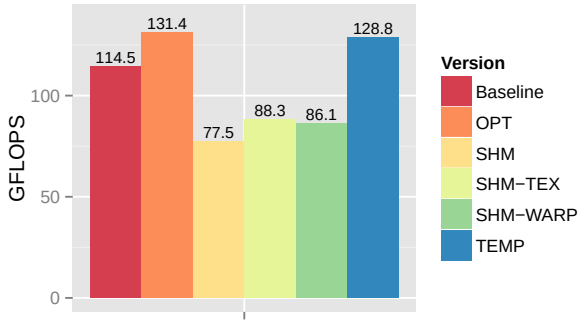


Figure 5: Performance of the baseline and optimized versions on Fermi M2075. The dashed line shows the peak attainable performance estimated based on the roofline model with no temporal blocking.

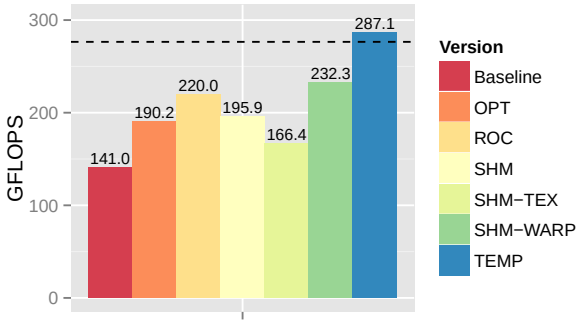


Figure 6: Performance of the baseline and optimized versions on Kepler K20X. The dashed line shows the peak attainable performance estimated based on the roofline model with no temporal blocking.

GPU can automatically cache global memory loads at L1 cache, we have observed that explicit blocking with shared memory is not always beneficial. Our experimental results reported in this paper also exhibit similar performance behavior. The latest generation of NVIDIA GPUs, Kepler, again changed the memory architecture and now global memory loads are not cached at L1, so different blocking methods need to be employed to efficiently run on the new architecture. This paper looked at several optimization methods and showed that simple implicit blocking with the read-only cache is indeed very effective, while explicit blocking with shared memory can further improve the performance.

Temporal blocking has also been extensively studied with various scientific computations such as stencils [12, 14]. It has also recently been evaluated on GPUs [3, 4, 7, 15], although mixed performance benefits have been reported. For example, for 3-D stencils, Zumbusch reported that it did not improve the performance on both Fermi and Kepler when compared to the version only with register blocking and no temporal blocking [15]. Holewinski et al. also reported that temporal blocking was not beneficial for 3-D Jacobi stencil [4]. In contrast to those previous results, our temporal blocking achieved an improvement by 20% with the 3-D stencil problem on the Kepler GPU.

Efficient halo accesses with warp specialization was first reported by Bauer et al. [1]. Our experiments in this paper confirm similar performance results on Kepler and show that further speedups can be attainable by combining warp specialization with temporal blocking.

8. CONCLUSION

This paper presented performance studies of a 7-point 3-D stencil on the recent NVIDIA GPU architectures. Our experimental evaluations show that the blocking with shared memory is essential for the stencil to achieve optimal performance on Kepler. This used to be the case for older pre-Fermi generations of NVIDIA GPUs, but had been considered unnecessary for stencils with regular grids since the hardware L1 cache of the Fermi GPU often works quite effectively. Overall, we achieved approximately 80% of the estimated peak performance by the roofline model, and even higher performance with temporal blocking on Kepler. While our current experiments are limited to the 7-point stencil, we expect that the results reported here will be applicable to other 3-D stencils as well.

The shared memory blocking optimizations, however, incurred non-trivial cost of code transformation. For example, while the original stencil can be written with less then 20 lines of code in non optimized CUDA, the fully optimized kernel takes more than 300 lines of code, resulting in 10-times increase of code size. Since manual applications to full-scale stencil applications that consist of, e.g., tens of much larger stencil kernels would be prohibitively costly, automated transformation techniques for these optimizations need to be developed. We plan to extend our high-level

stencil framework, Physis, to automatically generate CUDA codes with the optimizations shown to be effective in this study [5].

Acknowledgments

This project was partially supported by a JST, CREST program: "Highly Productive, High Performance Application Frameworks for Post Petascale Computing."

References

[1] M. Bauer, H. Cook, and B. Khailany. CudaDMA: optimizing GPU memory bandwidth via warp specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11. ACM, 2011.

[2] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and autotuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, Piscataway, NJ, USA, 2008. IEEE Press.

[3] T. Grosser, A. Cohen, P. H. J. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege. Split tiling for GPUs: automatic parallelization using trapezoidal tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 24–31. ACM, 2013.

[4] J. Holewinski, L. N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on GPU architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 311–320. ACM, 2012.

[5] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11. ACM, 2011.

[6] P. Micikevicius. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 79–84. ACM, 2009.

[7] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–13. IEEE Computer Society, 2010.

[8] NVIDIA. CUDA C Programming Guide version 5.0, 2013.

[9] NVIDIA. NVIDIA Kepler GK110 Architecture Whitepaper. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2013.

[10] E. Phillips and M. Fatica. Implementing the Himeno Benchmark with CUDA on GPU Clusters. In *IEEE International Parallel & Distributed Processing Symposium*, pages 1–10, Apr. 2010.

[11] T. Shimokawabe, T. Aoki, T. Takaki, T. Endo, A. Yamanaka, N. Maruyama, A. Nukada, and S. Matsuoka. Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11. ACM, 2011.

[12] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C. K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 117–128. ACM, 2011.

[13] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009.

[14] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 171–180. IEEE, 2000.

[15] G. Zumbusch. Vectorized Higher Order Finite Difference Kernels. In *State-of-the-Art in Scientific and Parallel Computing (PARA)*, 2012.