Bones: An Automatic Skeleton-Based C-to-CUDA Compiler for GPUs

CEDRIC NUGTEREN and HENK CORPORAAL, Eindhoven University of Technology

The shift toward parallel processor architectures has made programming and code generation increasingly challenging. To address this *programmability* challenge, this article presents a technique to fully automatically generate efficient and readable code for parallel processors (with a focus on GPUs). This is made possible by combining algorithmic skeletons, traditional compilation, and "*algorithmic species*," a classification of program code. Compilation starts by automatically annotating C code with class information (the algorithmic species). This code is then fed into the skeleton-based source-to-source compiler BONES to generate CUDA code. To generate efficient code, BONES also performs optimizations including host-accelerator transfer optimization and kernel fusion. This results in a unique approach, integrating a skeleton-based compiler for the first time into an automated flow. The benefits are demonstrated experimentally for PolyBench GPU kernels, showing geometric mean speed-ups of 1.4× and 2.4× compared to PPCG and PAR4ALL, and for five Rodinia GPU benchmarks, showing a gap of only 1.2× compared to hand-optimized code.

Categories and Subject Descriptors: C.1.4 [**Processor Architectures**]: Parallel Architectures; D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms: Performance, Languages

Additional Key Words and Phrases: GPU, algorithmic skeletons, parallel programming, compiler, CUDA

ACM Reference Format:
Cedric Nugteren and Henk Corporaal. 2014. Bones: An automatic skeleton-based C-to-CUDA compiler for GPUs. ACM Trans. Architec. Code Optim. 11, 4, Article 35 (December 2014), 25 pages.
DOI: http://dx.doi.org/10.1145/2665079

1. INTRODUCTION

The past decades of processor design have led to an increasingly heterogeneous computing environment, in which multicores are used in conjunction with accelerators such as Graphics Processing Units (GPUs). Both parallelism and heterogeneity have made programming a challenging task: Programmers are faced with a variety of new languages and are required to have detailed architectural knowledge to optimize their applications. This has made programmability and the related performance, portability, and productivity issues of major importance. Despite a significant amount of work on compilation, autoparallelization, and autotuning, programmers are still struggling with these issues, having to deal with low-level languages such as OpenCL and CUDA.

This work introduces a new source-to-source compiler to address these programmability issues. The compiler (named BONES) addresses the shortcomings of existing compilers, which fall short in at least one of the following areas: (1) They require code

Authors' address: C. Nugteren (corresponding author) and H. Corporaal, Department of Electrical Engineering, Eindhoven University of Technology, Postbus 513, 5600MB Eindhoven, The Netherlands; emails: {c.nugteren, h.corporaal}@tue.nl.

Permission to make digital or hard copies of part or all 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 show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2014 ACM 1544-3566/2014/12-ART35 $15.00
DOI: http://dx.doi.org/10.1145/2665079

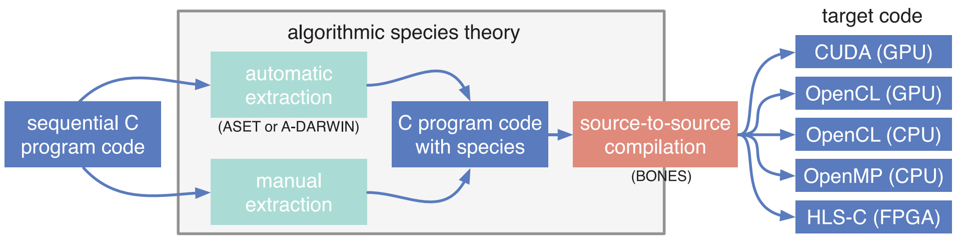


Fig. 1. A graphical overview of the relation between the algorithmic species theory (earlier work) and the BONES source-to-source compiler with five of its targets (this work focuses on the CUDA target).

restructuring or annotations, (2) they directly produce binaries or generate human unreadable code, or (3) they do not generate efficient code. The first shortcoming mostly affects application programmers who are unfamiliar with parallel programming, whereas the second shortcoming mostly affects programmers who are using compilers to perform the initial steps and are willing to further optimize the result. The third shortcoming affects all types of users. This work addresses these shortcomings by taking a unique approach: It uses an algorithm classification to drive a source-to-source compiler based on *algorithmic skeletons*.

The idea of algorithmic skeletons [Cole 1991] is motivated by the observation of similarities across efficient code for particular classes of program code. The technique is based on the use of parametrizable program code, known as *skeletons* or *skeleton implementations*. An individual skeleton can be seen as template code for a specific class of computations on a specific microprocessor. Users of skeleton-based compilers are required to identify a skeleton suitable for their program code and their target microprocessor, and they can subsequently invoke the skeleton to obtain program code for the chosen target. If no skeleton implementation is suitable for the specific class or microprocessor, it can be added manually. Future program code of the same class can then benefit from reuse of the skeleton code. Benefits of skeleton-based compilation are, among others, the flexibility of extension to other targets and the performance potential: Optimizations can be performed in the native language within the skeletons. Examples of recent skeleton-based source-to-source compilers are Aldinucci et al. [2013], Enmyren and Kessler [2010], and Steuwer et al. [2011].

This work discusses BONES, a source-to-source compiler based both on algorithmic skeletons and traditional compiler optimizations. In contrast to other skeleton-based compilers, BONES uses a formally defined algorithm classification and does not require programmers to manually identify classes nor select a suitable skeleton [Nugteren et al. 2013c]. This is achieved by using "*algorithmic species*," a classification of program code based on memory access patterns [Nugteren et al. 2013a, 2013b]. The relation of BONES with algorithmic species is shown in Figure 1. The main contributions are:

—**Section 2:** A survey of existing source-to-source compilers targeting GPUs.
—**Section 3:** A brief background on the algorithmic species classification.
—**Section 4:** The skeleton-based source-to-source compiler BONES, automatically generating efficient and readable code for parallel architectures (with a focus on GPUs).
—**Section 5:** Host-accelerator transfer optimizations (e.g., CPU-GPU) within BONES.
—**Section 6:** Kernel fusion with respect to algorithmic species.
—**Section 7:** A comprehensive experimental evaluation comparing the effects of the optimizations made in BONES, the performance of different targets, and a comparison against PPCG and PAR4ALL, two automatic C-to-CUDA compilers.
—**Section 8:** A discussion and evaluation of the taken approach.

```
int N = 512*512;
for(i=1; i<N-1; i++) {
  B[i] = 0.3*A[i-1] + 0.4*A[i] + 0.3*A[i+1];
}
```

Fig. 2. A 1D stencil example used to illustrate existing source-to-source compilers.

2. RELATED WORK

There are a large number of source-to-source compilers targeted at (partially) automating GPU programming. This section briefly introduces the most prominent compilers and discusses several of them in more detail. The focus lies on source-to-source compilers with a CUDA or OpenCL target. The following classes are distinguished:

—**Directives:** Several C-to-CUDA source-to-source compilers use directives in the form of pragmas. These user-supplied directives translate directly to compiler transformations, addressing the more labor-intensive tasks such as host-accelerator memory transforms. Examples are *hi*CUDA [Han and Abdelrahman 2011] (discussed in Section 2.1) and compilers using OpenMP directives [Noaje et al. 2011].

—**Algorithmic skeletons:** Existing work also uses the algorithmic skeletons approach to target GPUs. The most prominent examples are SkePU [Enmyren and Kessler 2010] (discussed in Section 2.2) and SkelCL [Steuwer et al. 2011]. These are not implemented as source-to-source compilers but instead use C++ templates to invoke skeletons. Other skeleton approaches targeting CUDA include Muesli [Ernsting and Kuchen 2011] and the work by Sato and Iwasaki [2009].

—**Semiautomatic:** OpenACC is a semiautomatic approach in that it performs code generation based on a combination of user-supplied directives and static analysis. Three C-to-CUDA compilers are based on OpenACC directives: PGI Accelerator [Wolfe 2010] (discussed in Section 2.3), HMPP Workbench, and the Cray compiler.

—**Fully automatic:** We identify two source-to-source compilers based on static analysis techniques: PAR4ALL [Amini et al. 2012] and PPCG [Verdoolaege et al. 2013]. Both are discussed in more detail in Section 2.4. Other automatic compilers are C-to-CUDA [Baghdadi et al. 2010] and a CUDA version of Pluto [Baskaran et al. 2010], but they are either not publicly available or not fully functional.

To illustrate the programming style and programming effort for different source-to-source compilers, we take the 1D stencil computation of Figure 2 as an example.

2.1. Directives Using *hi*CUDA

Source-to-source compilers based on directives rely on the help of programmers to generate code. A common case is the use of annotations (e.g., pragmas) in the source code that guide (or direct) the compiler toward generating efficient target code. A good example is the C-to-CUDA compiler *hi*CUDA [Han and Abdelrahman 2011].

Figure 3 applies *hi*CUDA directives to the example stencil code of Figure 2. The original code is still present (lines 1, 7, and 10), but a number of directives have been added. First of all, array sizes have to be set in case of dynamically allocated memory (line 2). Second, memory has to be allocated on the GPU (lines 3–4), copied between CPU and GPU (lines 3 and 13), and freed (line 14). The kernel is defined in lines 5–12. Most of these directives translate directly into CUDA statements or library calls.

Users of *hi*CUDA are still required to have GPU programming expertise. For example, the programmer has to specify the number of threads and threadblocks and has to specify which loops should be parallelised. Additionally, to use the on-chip scratchpad memory, the programmer has to supply directives defining which memory section to store locally and when to synchronize between threads (lines 8–9 in Figure 3). An

```
int N = 512*512;
#pragma hicuda shape A[N] B[N]
#pragma hicuda global alloc A[*] copyin
#pragma hicuda global alloc B[*]
#pragma hicuda kernel conv tblock(N/256) thread(256)
#pragma hicuda loop_partition over_tblock over_thread
for(i=1; i<N-1; i++) {
  #pragma hicuda shared alloc A[i-1:i+1] copyin
  #pragma hicuda barrier
  B[i] = 0.3*A[i-1] + 0.4*A[i] + 0.3*A[i+1];
}
#pragma hicuda kernel_end
#pragma hicuda global copyout B[*]
#pragma hicuda global free A B
```

Fig. 3. The 1D stencil example annotated with *hi*CUDA directives.

```
 1 // Global function definition
 2 OVERLAP_FUNC(stencil, float, 1, in,
 3   return 0.3*in[-1] + 0.4*in[0] + 0.3*in[1];
 4 )
 5
 6 // Main function
 7 int N = 512*512;
 8 skepu::Vector<float> A_v(N);
 9 skepu::Vector<float> B_v(N);
10 for (i=0; i<N; i++) {
11   A_v[i] = A[i];
12 }
13 skepu::MapOverlap<stencil> stencilKernel(new stencil);
14 stencilKernel(A_v,B_v);
15 for (i=0; i<N; i++) {
16   B[i] = B_v[i];
17 }
```

Fig. 4. The 1D stencil example using SkePU's algorithmic skeletons.

advantage of *hi*CUDA is its interprocedural support. For example, "kernel" directives can be placed around function calls, whereas other directives can be placed within these functions. Furthermore, the generated code can be inspected and modified. However, the code is not easily readable and is thus not suitable for further fine tuning.

2.2. Algorithmic Skeletons through SkePU

Applying algorithmic skeletons to many-core architectures such as GPUs has been accomplished recently in several works, of which SkePU [Enmyren and Kessler 2010] is an example. SkePU is publicly available and supports CUDA and OpenCL as targets.

Figure 4 shows the SkePU implementation of the stencil example of Figure 2. Seven skeletons are supported (*map*, *mapArray*, *mapOverlap*, *mapReduce*, *reduce*, *scan,* and *generate*), from which we select *mapOverlap* to match the stencil code (line 13). The figure also shows the copying and conversion of arrays into the `skepu::Vector` containers (lines 8–12 and 15–17). Furthermore, the functionality is defined (lines 2–4), and the kernel is invoked (line 14).

SkePU implements skeletons as C++ libraries and therefore does not require separate compilation: Including the supplied header files and compiling with *nvcc* is sufficient. This approach also has drawbacks: SkePU requires the original code to be rewritten, thus increasing the programming effort and limiting the portability. As shown in the example, SkePU also uses special data containers. This will not pose a problem for cases where the whole program can be designed using SkePU containers.

```
1 int N = 512*512;
2 #pragma acc region
3 {
4   #pragma acc for independent
5   for(i=1; i<N-1; i++) {
6     B[i] = 0.3*A[i-1] + 0.4*A[i] + 0.3*A[i+1];
7   }
8 }
```

Fig. 5. The 1D stencil example using OpenACC directives and PGI Accelerator.

However, for cases where kernels are considered as small parts of a larger application, a copy-in and a copy-out is required. SkePU furthermore supports lazy memory copying, can generate code for systems with multiple GPUs, and supports both CUDA and OpenCL.

2.3. OpenACC Directives with PGI Accelerator

PGI Accelerator [Wolfe 2010] is a commercial C and Fortran to CUDA source-to-source compiler. It performs extensive code analysis, but also relies on programmer input in the form of OpenACC directives. By combining compiler analysis with a varying degree of directives, the user remains in control of the effort versus performance tradeoff.

An OpenACC version of the example of Figure 2 is given in Figure 5. In the ideal case, PGI Accelerator requires only a single directive to delimit the scope of acceleration (`acc region`, line 2) and will extract the remaining information using static analysis: which arrays to copy-in or copy-out, which loops to parallelize, which temporary results to store in on-chip memory, and the like. If the compiler cannot find such information statically, the user will be asked to supply additional directives. Such a directive is provided for in the stencil example (line 4) to inform the compiler that the iterations of the loop are independent of each other.

The PGI Accelerator source-to-source compiler provides information to the user as to how the code is generated. The programmer can then supply additional directives to guide the compiler in a specific direction. The compiler furthermore analyzes aspects such as thread occupancy, memory accesses, and register usage.

2.4. Automatic Compilation with Par4All and PPCG

Several C-to-CUDA compilers are fully automatic: They are able to perform dependence analysis and loop transformations without user annotations. The most prominent examples are Par4All [Amini et al. 2012] and ppcg [Verdoolaege et al. 2013].

The code transformation and parallelization framework PIPS is the main component of the Par4All source-to-source compiler, which generates CUDA code. The compiler takes unmodified C-code as input, such as the stencil code example of Figure 2. However, because PIPS is based on convex array region analysis [Creusillet and Irigoin 1997], restrictions apply to the input code, requiring loops to have static control and affine bounds and references [Amini et al. 2012; Creusillet and Irigoin 1997]. Par4All uses macros to hide CUDA statements from the generated code to improve readability.

The compiler ppcg is based on the polyhedral model and generates CUDA code. It has similar properties as Par4All: The compiler is fully automatic but imposes restrictions on input code; only static affine loop nests are transformed. Both ppcg and Par4All are able to perform host-accelerator (e.g., CPU-GPU) transfer optimizations.

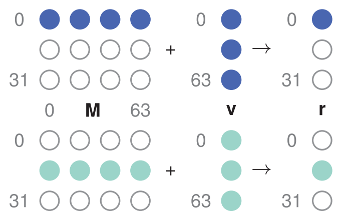
2.5. Evaluation and Discussion

This section discussed several different source-to-source compilers, including compilers based on directives (*hi*CUDA and PGI Accelerator), compilers using algorithmic

Table I. Overview of Properties of the Discussed Approaches

	*hi*CUDA	SkePU	PGI Acc.	Par4All	PPCG	BONES
Technique	Directives	Skeletons	Directives +analysis	Static analysis	Static analysis	Skeletons +analysis
Freely available	✓	✓	×	✓	✓	✓
CUDA target	✓	✓	✓	✓	✓	✓
OpenCL target	×	✓	×	✓	✓	✓
Compilation (source-to-source)	✓	×	✓	✓	✓	✓
Fully automatic	×	×	×	✓	✓	✓
Uses regular C data-types	✓	×	✓	✓	✓	✓
Data-sizes needed at run-time	✓	✓	×	✓	✓	✓
Generates readable code	×	N/A	×	✓	✓	✓
Generates multi-GPU code	×	✓	×	×	×	×
Performs kernel fusion	×	×	×	✓	✓	✓

```
for (i=0; i<32; i++) {
  r[i] = 0;
  for (j=0; j<64; j++) {
    r[i] += M[i][j]*v[j];
  }
}
```



`M[0:31,0:63]|chunk(-,0:63) ∧ v[0:63]|full → r[0:31]|element`

Fig. 6. Matrix-vector multiplication (left) classified as species (bottom) and an illustration of accesses of the *i*-loop's first 2 iterations using different colors (right).

skeletons (SkePU), and compilers based on static analysis (PGI Accelerator, Par4All, and PPCG). Table I gives a summary of their properties.

Par4All and PPCG are winners from a programmability, productivity, and portability perspective because they require no modifications to the input code (performance is evaluated in Section 7). Not considering performance, the other three discussed compilers fall short in that (1) they are not fully automatic and require code restructuring (SkePU) or annotations (*hi*CUDA and PGI Accelerator), and (2) they directly produce binaries (SkePU) or generate human unreadable code (*hi*CUDA and PGI Accelerator). This work addresses these shortcomings by using the algorithmic species classification to drive a source-to-source compiler based on algorithmic skeletons.

3. BACKGROUND: ALGORITHMIC SPECIES

Algorithmic species is an algorithm classification based on memory access patterns of arrays in loop nests. There are two formal theories of algorithmic species: the original theory based on the polyhedral model [Nugteren et al. 2013b] and a newer theory based on array reference characterizations [Nugteren et al. 2013a].

Consider the matrix-vector multiplication of Figure 6. The corresponding species (valid for both theories; shown at the bottom of Figure 6) with respect to the outer loop can be interpreted as follows: The production of a single *element* out of the total 32 elements in `r` requires a *chunk* of data in the second dimension of `M` (a row) and the *full* array `v` of size 64. This species captures the memory access patterns and can be used for classification purposes; for example, to predict performance or to guide compiler optimizations. The species for the stencil example of Figure 2 is given as "*A[1:N-2]|neighbourhood(-1:1) → B[1:N-2]|element*," capturing the data-reuse as a 1D *neighbourhood* pattern.

Internally, the newer array reference theory of algorithmic species [Nugteren et al. 2013a] classifies individual accesses as the 5-tuple $(\mathcal{N}, \mathcal{A}, \mathcal{D}, \mathcal{E}, \mathcal{S})$. This description includes the reference's name $(\mathcal{N})$, whether it is a read or write access $(\mathcal{A})$, its domain $(\mathcal{D})$, the number of consecutive elements accessed with respect to the parallel loop(s) $(\mathcal{E})$ and its step $(\mathcal{S})$. For example, accesses to `r` (seen from the i-loop) in Figure 6 are classified as (r, *write*, [0 : 31], 1, 1) and are translated later into "`r[0:31]|element`."

The example of matrix-vector multiplication covers the access patterns *element*, *chunk,* and *full*, forming a single species when combined. In total, five patterns (*element*, *neighbourhood*, *chunk*, *full*, and *shared*) can be combined into different species. Species can be extracted manually or using A-DARWIN [Nugteren et al. 2013a], a tool that follows the formal species theory. It classifies code that fits the polyhedral model fully automatically and partially automatically otherwise. Note that this tool classifies parallel code rather than performing transformations to extract (more) parallelism.

4. A SKELETON-BASED SOURCE-TO-SOURCE COMPILER

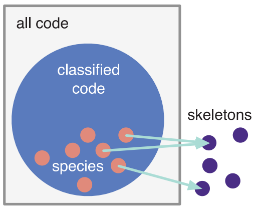
This section discusses BONES, a source-to-source compiler based on algorithmic skeletons. The compiler takes C program code annotated with class information as an input (the algorithmic species from Section 3). This "species information" is used to determine which skeleton to use and is additionally used to enable or disable additional transformations and optimizations that are not possible within the skeletons themselves. BONES[1] is written in Ruby and uses the C-to-AST module CAST to be able to work on Abstract Syntax Trees (ASTs). The compiler ships with a total of 15 pre-written skeletons for five different targets: CUDA for NVIDIA GPUs, OpenCL for AMD GPUs, OpenCL for CPUs (both for the AMD and the Intel SDK), and OpenMP for CPUs, of which five are for the GPU-CUDA target.

In contrast to existing skeleton-based compilers, BONES uses the algorithmic species information to select a suitable skeleton. This not only enables automation of the tool chain (by combining with A-DARWIN), but also overcomes common criticisms of skeleton-based compilers such as (1) the problem of selecting a suitable skeleton or (2) what to do when a user selects an incompatible skeleton.

Algorithmic species map in principle one-to-one to skeletons: The compiler will need to supply a skeleton for every species it wants to support. These skeletons must be constructed in such a way that they are correct (and preferably optimized) for all possible types of program code that belong to a particular species. However, for practical reasons (to save work and code duplication and to be able to accommodate an infinite amount of species), BONES provides a species-to-skeleton mapping such that multiple species can map to the same skeleton. For example, for the GPU-CUDA target, algorithmic species of the form "*neighbourhood* $\rightarrow$ *element*" and "*neighbourhood* $\wedge$ *element* $\rightarrow$ *element*" both map to a single skeleton that performs explicit caching of the *neighbourhood* in the GPU's on-chip scratchpad memory. Figure 7 illustrates this many-to-one species-to-skeleton mapping: A limited number of skeletons cover a larger amount of species. BONES provides optimized skeletons for the species encountered so far (see Section 8 for a discussion), but others can be added as needed. As shown in the figure, algorithmic species classify a subset of all code, considering only parallel loop nests.

The workings of BONES are illustrated by Figure 8. This figure shows an example input with two code segments identified as two different species (species X and species Y). The compiler first loads and invokes the corresponding skeletons for a given target. The target is specified by the user but could be selected dynamically based on the invoked skeleton and data sizes, as is done by SkePU [Dastgeer et al. 2013]. Next,

[1]Source code is available at http://github.com/cnugteren/bones/.



species (∗ is a wildcard)	skeleton
`[*:*]|neighbourhood` → `[*:*]|element`	1D-caching
`[0:N,0:N]|chunk(-,0:N)` ∗ → ∗	shuffling x1
`[0:N,0:N]|chunk(-,0:N)` ∧ `[0:N,0:N]|chunk(-,0:N)` ∗ → ∗	shuffling x2
∗`|element` → ∗`|shared`	reduction
all other species	default

Fig. 7. Illustration of the relation between species and skeletons (left). Species are classes of code (small circles) within the region of interest (large circle) that map many-to-one to skeletons. The table shows a practical case of a species-to-skeleton mapping for the GPU-CUDA target, discussed further in Section 4.1.

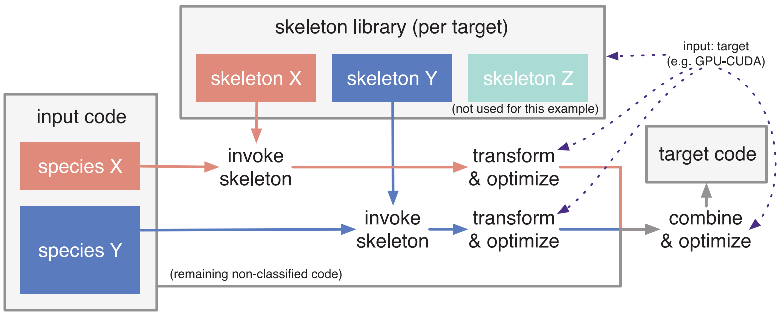


Fig. 8. Illustration of the structure of the BONES compiler for example input code with two different species (left). The example shows the skeleton library (top) and the transformation and optimization passes.

the skeleton-specific transformations and optimizations are performed. Finally, BONES combines the results to obtain target program code.

Apart from using skeletons for the performance-critical parts of the code (the kernels), BONES also uses templates (or skeletons) to generate the remainder of the code. These skeletons are target-specific, rather than target- and species-specific, but still have parameters (e.g., array name or array size). Examples of such skeletons include OpenCL/CUDA memory allocation, host-to-accelerator data transfer, platform initialization, and header inclusion. Using skeletons for such tasks keeps the compiler lightweight and makes modification or extension to other targets straightforward.

4.1. Example Skeletons

The use of skeletons within BONES is illustrated through an OpenMP skeleton. Figure 9 shows the skeleton itself and its instantiation for the matrix-vector multiplication example ($\vec{r} = \mathbf{M} \cdot \vec{v}$, Figure 6). The skeleton (left) highlights keywords: `parallelism` represents the amount of potential parallelism found within the species and `code` fills in the transformed code. For brevity, the example is heavily simplified, excluding comments, boundary and initialization code, function calls, and definitions, and it makes assumptions such as the divisibility of the amount of parallelism by the thread count.

Additionally, Figure 10 shows an example of a skeleton for the GPU-CUDA target. This skeleton is specific to species of the form *"0:N,0:N|chunk(-,0:N) → 0:N,0:N|element,"* similar to the matrix-vector multiplication example. A naive mapping to CUDA results in *uncoalesced* accesses to the *chunk* array: Subsequent accesses will be made by the same thread. To re-enable coalescing in these cases (important for performance), a special skeleton with a preshuffling kernel is introduced. This skeleton (Figure 10) has a kernel for the work (lines 1–8) and a kernel to reorder the input by rearranging data in the on-chip memory (lines 10–21). The use of this skeleton implies

```
int count = omp_get_num_procs();
omp_set_num_threads(count);
#pragma omp parallel
{
  int work, start, end;
  int tid = omp_get_thread_num();
  work = ⟨parallelism⟩/count;
  start = tid*work;
  end = (tid+1)*work;

  // Start the parallel work
  for(gid=start; gid<end; gid++) {
    ⟨code⟩


  }
}
```

```
int count = omp_get_num_procs();
omp_set_num_threads(count);
#pragma omp parallel
{
  int work, start, end;
  int tid = omp_get_thread_num();
  work = 32/count;
  start = tid*work;
  end = (tid+1)*work;

  // Start the parallel work
  for(gid=start; gid<end; gid++) {
    r[gid] = 0;
    for (j=0; j<64; j++)
      r[gid] += M[gid][j]*v[j];
  }
}
```

Fig. 9. An example simplified skeleton for OpenMP (left) and the same skeleton invoked for matrix-vector multiplication (right). Details and optimizations are omitted for brevity and clarity.

```
 1 // CUDA kernel for the actual work (simplified)
 2 __global__ void kernel0(...) {
 3   int gid = blockIdx.x*blockDim.x+threadIdx.x;
 4   if (gid < ⟨parallelism⟩) { // Incomplete blocks
 5     ⟨ids⟩
 6     ⟨code⟩
 7   }
 8 }
 9
10 // CUDA kernel for pre-shuffling (simplified)
11 __global__ void kernel1(...) {
12   int tx = threadIdx.x; int ty = threadIdx.y;
13   __shared__ ⟨type⟩ b[16][16];
14   int gid0 = blockIdx.x*blockDim.x + tx;
15   int gid1 = blockIdx.y*blockDim.y + ty;
16   int nid0 = blockIdx.y*blockDim.y + tx;
17   int nid1 = blockIdx.x*blockDim.x + ty;
18   b[ty][tx] = in[gid0 + gid1*⟨dims⟩/⟨params⟩];
19   __syncthreads();
20   out[nid0 + nid1*⟨params⟩] = b[tx][ty];
21 }
```

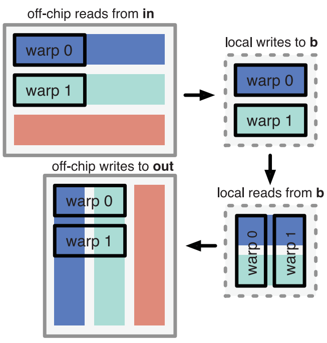


Fig. 10. An example simplified skeleton (left) for the GPU-CUDA target with a preshuffling kernel (lines 10–21) to enable coalesced accesses. The shuffling is illustrated schematically (right).

a transformation in the original code (e.g., from `M[i][j]` into `M[j][i]`), which is handled by the compiler. The skeleton highlights additional keywords (compared to Figure 9): `ids` computes the identifier corresponding to the current thread, `type` gives the data type, `dims` represents the array sizes, and `params` represents the chunk sizes. Again, this skeleton is simplified (e.g., not showing boundary checks or host code).

Currently, BONES provides five CUDA skeletons (see Figure 7). This includes a "default" skeleton for which the parallel loops are mapped to threads and the loop body is mapped to a CUDA kernel. A variation of this is a skeleton for *neighbourhood* type computations, caching input data into the GPU's on-chip scratchpad memory. Furthermore, there are two skeletons with a preshuffling kernel (see Figure 10): one for a single input and one for two inputs. Finally, there is a special associative reduction skeleton for species in the form of *"element → shared."* In this case, an alternative is to use the default skeleton and ask the compiler to replace all operations involving *shared* variables by atomic counterparts to ensure correctness.

4.2. Compiler Optimizations

This section discusses basic optimizations: Two more advanced optimizations are discussed separately in Sections 5 (host-accelerator data transfers) and 6 (kernel fusion). BONES performs various basic transformations and optimizations, most of which are conditionally applied based on the algorithmic species. An example of a basic transformation is the replacement of array indices and names of input arrays in *neighbourhood*-based skeletons for GPUs by local indices and names. This transformation is a matter of name-changing; the actual definition of the local indices and the prefetching into local memories is performed within the corresponding skeletons.

Furthermore, BONES flattens data structures to a single dimension when generating GPU code (to satisfy CUDA/OpenCL requirements). Nested parallel loops are also flattened, decoupling the number of loops from the threads or work items provided by CUDA and OpenCL. In contrast, many existing approaches (e.g., Amini et al. [2012], Enmyren and Kessler [2010], and Verdoolaege et al. [2013]) map multidimensional loops to multidimensional threads or work items, thus limiting applicability to two- or three-dimensional loops and data structures. In contrast, BONES is able to handle arrays of any dimension and any degree of loop nesting.

Additionally, several performance-oriented transformations are made within BONES. This includes *register file caching* and *thread coarsening* [Magni et al. 2013]. Register file caching can replace array accesses (mapped to off-chip memories) with scalar accesses (mapped to registers) under certain conditions. For example, in matrix-vector multiplication (Figure 6), accesses to vector $\vec{r}$ can be replaced by scalar accesses under the condition that a final store to $\vec{r}$ is added at the end of the loop body. In case there are insufficient registers available, spilling will ensure a neutral worst-case performance impact. Thread coarsening or thread merging is a technique to increase the workload per thread. This comes at the cost of parallelism, but could increase data reuse through locality or factor out common instructions [Lee et al. 2013]. In BONES, coarsening is implemented by replicating code line by line (and renaming variables where required) and is only considered if the species has data reuse. For example, in the case of "*0:M,0:N|chunk(-,0:N)* $\rightarrow$ *0:M,0:N|element*", a total of $N \cdot M$ elements are produced, although only M chunks are available as input, resulting in the reuse of the input by a factor N. Coarsening is only enabled for kernels with sufficient parallelism (e.g., at least 2^{15} threads on a GPU) but without divergent control flow. The performance effects of register file caching and thread coarsening are discussed in Section 7.

5. OPTIMIZING HOST-ACCELERATOR DATA TRANSFERS

Several of today's parallel microprocessors are designed as *accelerators*: They require a *host* processor to dispatch tasks. Furthermore, they might have a separate memory, requiring host-accelerator transfers of data. Executing multiple kernels subsequently gives opportunities to optimize these transfers in several ways [Guelton et al. 2012; Jablin et al. 2012]: (1) transfers can be omitted (e.g., subsequent kernels use the same data), (2) transfers can be done in parallel with host code (e.g., start the copy as soon as the data is ready), and (3) transfers related to a previous or upcoming kernel can run in parallel with kernel execution. In case the host and accelerator share a memory (e.g., CPU, fused CPU/GPU), transfers can be completely removed. This section discusses the optimizations for such cases first, followed by optimizations for cases when the host and accelerator have their own memory (e.g., GPU, Intel MIC).

To improve performance for the case in which the host and accelerator share the same memory, BONES enables *zero-copy* for the OpenCL targets using an aligned memory

```
copy−in(A,1)
sync(1)
B ← A
copy−out(B,2)
sync(2)
copy−in(B,3)
copy−in(D,3)
sync(3)
C ← B + D
copy−out(C,4)
sync(4)
```

```
copy−in(A,1)
sync(1)
B ← A
copy−out(B,2)
sync(2)
copy−in(D,3)
sync(3)
C ← B + D
copy−out(C,4)
sync(4)
```

```
copy−in(A,1)
copy−in(D,3)
sync(1)
B ← A
copy−out(B,2)
sync(2)
sync(3)
C ← B + D
copy−out(C,4)
sync(4)
```

```
copyin(A,1)
copyin(D,3)
sync(1)
B ← A
copyout(B,4)
sync(3)
C ← B + D
copyout(C,4)
sync(4)
```

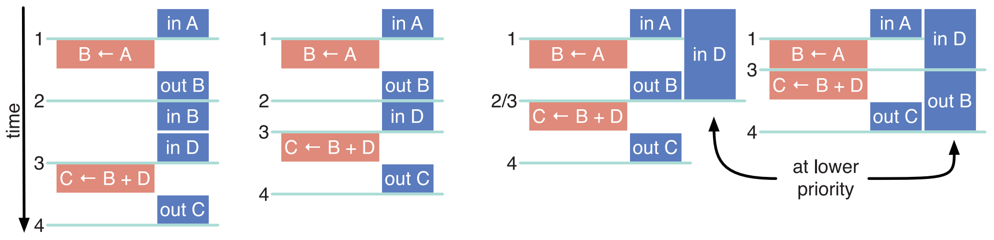


Fig. 11. Pseudocode illustrating several optimization steps: original (left), partly optimized (mid-left and mid-right), fully optimized (right). The second argument to the copy-in and copy-out functions give the deadlines, corresponding to the numbers of the synchronization barriers. The figures below the pseudocode show time progressing from top to bottom (not to scale).

allocation scheme. In this case, a memory copy can be saved by performing a pointer-only copy (a zero-copy). BONES enables zero-copy in OpenCL for CPUs by fulfilling Intel's two requirements: (1) using specific OpenCL memory map and memory un-map functions and (2) aligning all allocations to 128-byte boundaries. To ensure aligned memory allocations in the original code, BONES provides an aligned dynamic memory allocation implementation and ensures that stack arrays are aligned.

Next, consider the case where the host and accelerator have their own memory. After marking the identified algorithmic species with pragmas in A-DARWIN, the tool is instructed to (1) mark inputs and outputs as copy-ins and copy-outs for the current kernel and (2) add synchronization pragmas after the transfers. BONES then generates a second asynchronous host thread, receiving transfer requests and performing synchronizations. In its most basic form, BONES performs a copy-in of all required data before starting a kernel, and it performs a copy-out immediately afterward. This is illustrated through an example with two kernels, shown on the far left in Figure 11.

After producing the initial nonoptimized form, A-DARWIN is extended to perform different optimization steps recursively: (1) copy-ins directly after copy-outs are removed (e.g., from left to mid-left in Figure 11), (2) copy-ins are moved to the front if the data is not written by the previous species (e.g., from mid-left to mid-right in Figure 11), (3) copy-outs can be delayed if the data is not written by the next species (e.g., from mid-right to right in Figure 11), (4) unused synchronization barriers are omitted or merged, and (5) transfers are moved out of a loop body if possible (not shown in this example). The performance impact is discussed in Section 7.

6. KERNEL FUSION

Loop fusion is a performance-oriented transformation that combines two loop nests into a single new loop nest [Pouchet et al. 2010]. Its dual is loop distribution, creating new loops by splitting a loop body. Because fusion has been extensively discussed in the literature [Darte 2000; Kennedy and McKinley 1994; Pouchet et al. 2010], we focus on a special (case of loop fusion: kernel fusion in the context of algorithmic species. Kernel

```
for(i=0; i<8; i++) {
  B[i] = 0;
  for(j=0; j<4; j++)
    B[i] += A[i][j];
}

for(i=0; i<8; i++) {
  C[i] = 4*B[i];
}
```

⇓

```
for(i=0; i<8; i++) {
  B[i] = 0;
  for(j=0; j<4; j++)
    B[i] += A[i][j];
  C[i] = 4*B[i];
}
```

✓

```
for(i=0; i<8; i++) {
  val = A[i];
  for(j=0; j<4; j++)
    B[i][j] = val;
}
for(i=0; i<8; i++) {
  B[i][0] *= C[0];
  B[i][1] *= C[1];
}
```

⇓

```
for(i=0; i<8; i++) {
  val = A[i];
  for(j=0; j<4; j++)
    B[i][j] = val;
  B[i][0] *= C[0];
  B[i][1] *= C[1];
}
```

✓

```
for(i=0; i<8; i++) {
  A[i] = 0;
}

for(i=0; i<8; i++) {
  B[i] = A[i+1];
}
```

⇓

```
for(i=0; i<8; i++) {
  A[i] = 0;
  B[i] = A[i+1];
}
```

×

Fig. 12. Three examples with two loop nests each (top) and the corresponding fused code (bottom). For the left and middle examples, fusion is legal; for the right example, it is not.

fusion is limited in the sense that (1) fusion must not introduce loop-carried dependences because they prevent parallel execution, and (2) the loops are not considered interchangeable or transformable—they are already classified as a particular species.

Kernel fusion (and loop fusion in general) can be beneficial in terms of performance and energy efficiency for several reasons: (1) kernel startup times can be reduced (e.g., thread launch overheads), (2) optimization opportunities might arise by merging the two loop bodies (e.g., common expression elimination), and (3) data locality can improve. On the other hand, kernel fusion can also degrade performance, for example, by introducing additional control flow or cache contention.

Loop fusion can be decomposed into three main aspects that are strongly connected and often evaluated together [Kennedy and McKinley 1994]. First, loop transformations such as interchange and shifting can be applied to make fusion possible and advantageous by creating compatible *loop headers* (step, bounds) [Darte 2000]. Second, the safety of fusion needs to be evaluated: Dependencies need to be preserved when fusing two loops. Finally, the performance aspects need to be considered: Fusion should only be applied if it is beneficial for performance or energy efficiency. Because loop headers can be made compatible before feeding C code into BONES and A-DARWIN (using tools such as Pluto or PoCC), this section discusses only the latter two aspects. Specifically, we reformulate the problem of kernel fusion in the context of algorithmic species, discussing whether fusion is legal and when it is beneficial to be applied.

6.1. Legality of Fusion

Consider two fusion candidate kernels X and Y, with Y following X directly. Fusion of these kernels is legal in any of the following cases:

—**Independent case:** Kernel Y does not read from or write to any output of kernel X and does not write to any input of kernel X. Locality can arise by sharing an input.

—**Equal access pattern:** Kernel Y reads from or writes to output(s) of kernel X or writes to input(s) of X as long as the access patterns are equal. An example is shown in the left of Figure 12, in which X writes to `B` and Y reads from `B` with the same pattern: *0:7|element*.

—**Compatible access pattern:** Kernel Y reads from or writes to output(s) of kernel X and writes to input(s) of X for a combination of access patterns that preserves loop-carried dependencies. The specific combinations are patterns with nonintersecting domains and patterns for which the intersecting (but unequal) domains are part of *chunk* accesses (formal definition in equation (1)). An example is given in the middle of Figure 12, showing compatible access patterns for array B.

An example of a case where loop fusion is illegal is shown on the right-hand side of Figure 12. The access patterns to array A are incompatible in this case (*0:7|element* and *1:8|element*). Tools such as Pluto and PoCC can resolve this by shifting the second loop by one iteration, thus creating matching access patterns.

Legality of kernel fusion can be translated into the 5-tuple notation as used for the array reference-based theory of algorithmic species [Nugteren et al. 2013a]. For kernels X and Y, this results in Equation (1) (assuming a 1D array for simplicity), to be applied to all array reference combinations $(\mathcal{R}_x, \mathcal{R}_y)$ for which the names match $(\mathcal{N}_x = \mathcal{N}_y)$ and at least one of the references is a write. This reduces to a domain intersection test for two *element*-type accesses because, in that case, the step and the number of elements are of unit size: $\mathcal{S}_x = \mathcal{S}_y = \mathcal{E}_x = \mathcal{E}_y = 1$.

$$\begin{aligned} &\nexists(i_x \cdot \mathcal{S}_x + e_x = i_y \cdot \mathcal{S}_y + e_y) \\ \textbf{with} \quad & i_x \neq i_y \quad \textbf{and} \quad 0 \leq e_x < \mathcal{E}_x \quad \textbf{and} \quad 0 \leq e_y < \mathcal{E}_y \\ \textbf{and} \quad & \mathcal{D}_x^{lower} \leq i_x \leq \mathcal{D}_x^{upper} \quad \textbf{and} \quad \mathcal{D}_y^{lower} \leq i_y \leq \mathcal{D}_y^{upper}. \end{aligned} \tag{1}$$

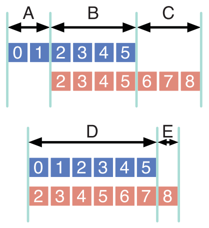
A-DARWIN tests for the legality of fusion according to Algorithm 1. The algorithm iterates over all combinations of inputs and outputs (line 1), verifying the three earlier discussed cases: independence (line 2), an equal access pattern (line 3), and a compatible access pattern (lines 4 and 5). If the test fails for one of the combinations, kernel fusion is not legal (line 6). Only if all combinations pass can kernel fusion be applied (line 12). The test for a compatible access pattern uses the dependence test of Equation (1) and is implemented as a combination of the GCD test and the Banerjee test as is also used within A-DARWIN [Nugteren et al. 2013a].

ALGORITHM 1: Legality of kernel fusion in the context of algorithmic species.

```
   Input: array reference characterisations $R_X$ and $R_Y$ for kernels X and Y
1  foreach combination of $\mathcal{R}_x \in R_X$ and $\mathcal{R}_y \in R_Y$ do
2      if $(\mathcal{N}_x = \mathcal{N}_y)$ and $(\mathcal{A}_x = w$ or $\mathcal{A}_y = w)$ then
3          if $\mathcal{D}_x \neq \mathcal{D}_y$ or $\mathcal{E}_x \neq \mathcal{E}_y$ or $\mathcal{S}_x \neq \mathcal{S}_y$ then      // equal access patterns
4              if $\mathcal{D}_x \cap \mathcal{D}_y \neq \emptyset$ then                  // non-intersecting domains
5                  if $\mathcal{R}_x$ and $\mathcal{R}_y$ are not compatible then        // equation 1
6                      return false
7                  end
8              end
9          end
10     end
11 end
12 return true
```

6.2. Performance Considerations

After testing for legality of kernel fusion, a decision needs to be made whether and how to apply kernel fusion. Deciding whether to apply kernel fusion is a problem dependent on many factors, including architectural properties. Furthermore, loop fusion in general



```
for(i=0; i<7; i++) {
  if (i<6) {
    f(blue[i]);
  }
  f(red[i+2]);
}
```

```
for(i=0; i<6; i++) {
  f(blue[i]);
  f(red[i+2]);
}
for(i=6; i<7; i++) {
  f(red[i+2]);
}
```

Fig. 13. Kernel fusion for nonmatching loop bounds. Shifting the second loop creates two sections (D and E), which can be implemented either using a conditional statement (middle) or a second loop nest (right).

(with respect to optimizing performance) has been theoretically proved to be a difficult problem, even for specific objectives such as maximizing data reuse [Darte 2000]. Because of the complexity and the number of variables involved, BONES allows the user to optionally disable kernel fusion: Fusion is enabled by default when legal. Section 7.1 evaluates the benefits of fusion.

There are different ways kernel fusion can be implemented when loop headers are not perfectly matching. An example of nonmatching loop bounds is given on the left-hand side of Figure 13: The first loop runs from 0 to 5 (blue) and the second from 2 to 8 (red). This results in tree code sections: (A) only the first loop is active, (B) both loops are active, and (C) only the second loop is active. If there are no restricting dependencies, the second loop can be shifted, obtaining two sections: (D) both loops are active, and (E) only the second loop is active. In the latter case, fusion can be applied in two different ways, as shown in Figure 13: (1) A conditional statement in the body bounds the execution of the D section, or (2) a second loop nest is created to complete the E section separately from the main body. The first technique is applied in BONES. Although this will create a number of idle threads (depending on the length of the E section) and introduces additional control flow, it does save the overhead of launching a second (typically much smaller) kernel. In particular for a GPU, the overhead of idle threads is negligible, and the conditional statement will result in at most one warp with branch divergence.

7. EXPERIMENTAL RESULTS

This section evaluates BONES experimentally by (1) measuring the benefits of the compiler optimizations, (2) comparing the different targets of BONES, and (3) comparing the GPU-CUDA target of BONES to state-of-the-art C-to-CUDA compilers.

All experiments are based on the PolyBench/C 3.2 benchmark suite [Pouchet and Grauer-Gray 2013]. This suite allows a head-to-head comparison against polyhedral-based compilers [Amini et al. 2012; Verdoolaege et al. 2013] and enables automatic extraction of algorithmic species using A-DARWIN. From PolyBench's 30 benchmarks, 28 contain parallelism in their current form. In these benchmarks, a total of 60 species are identified (not including nested species). However, several species are inner loops with a small amount of iterations and little work, resulting in execution times of less than a millisecond, dominated by start-up and measurement overheads. We therefore exclude `adi`, `cholesky`, `dynprog`, `durbin`, `fdtd2d-apml`, `gramschmidt`, `lu`, `ludcmp`, `reg_detect`, `symm`, `trmm`, and `trisolv`. This exclusion can be automated by integrating a performance model (e.g., Dastgeer et al. [2013]). All in all, 34 (not necessarily unique) species are included, spread across 16 benchmarks. Within a benchmark, multiple species are numbered sequentially.

Our experiments include all of BONES's five targets, with a focus on CUDA. An overview of the test setup is given in Table II. The OpenCL and C compilers are instructed to autovectorize code in order to use the CPU's AVX vector instructions.

Table II. Configuration Setup for the Parallel Targets and the Reference Target

Target	Language	Compiler	Hardware	Core Count
GPU-CUDA	CUDA	NVCC 5.0	NVIDIA GTX 470	448 CUDA
GPU-OpenCL-AMD	OpenCL	AMD APP 2.7	AMD HD7950	1,792 stream
CPU-OpenCL-AMD	OpenCL	AMD APP 2.7	Intel Core i7-3770	4 (8 threads)
CPU-OpenCL-Intel	OpenCL	Intel OpenCL '12	Intel Core i7-3770	4 (8 threads)
CPU-OpenMP	OpenMP	GCC 4.6.3	Intel Core i7-3770	4 (8 threads)
CPU-reference	C	GCC 4.6.3	Intel Core i7-3770	4 (8 threads)

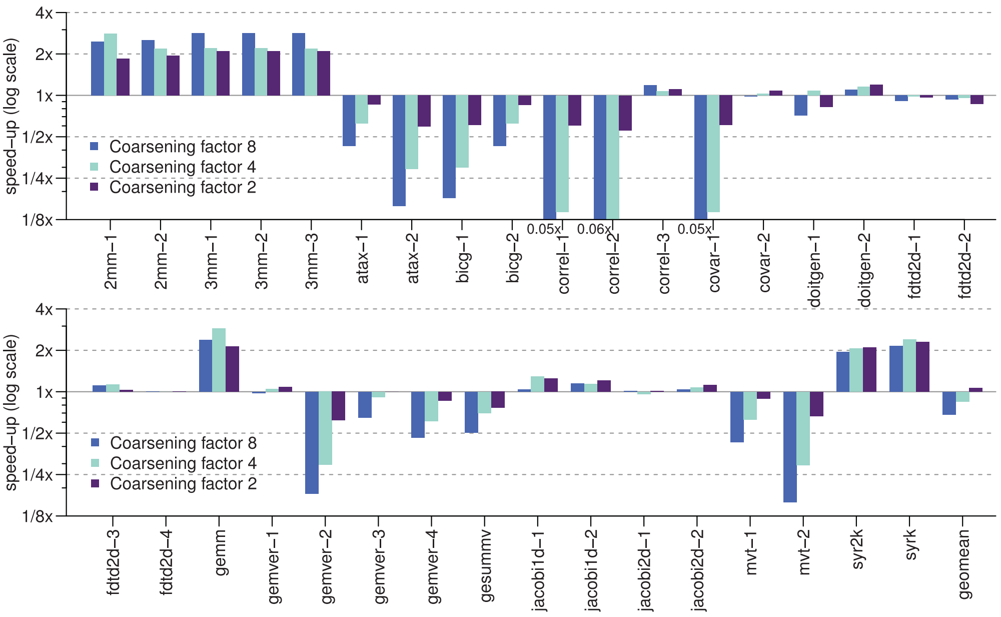


Fig. 14. Speed-ups of **coarsened** code over noncoarsened code for the GPU-CUDA target.

The CPU's Turbo Boost technology is disabled. PolyBench is configured to use "large datasets" and single-precision floating point. Results of multiple runs are averaged, and each run starts with a warmup dummy computation followed by a cache flush.

Two state-of-the-art polyhedral-based compilers are included here for comparison: PAR4ALL (version 1.4.1, May 2012) [Amini et al. 2012] and PPCG (version 0.02, April 2014) [Verdoolaege et al. 2013]. Apart from C-to-CUDA [Baskaran et al. 2010] (limited to kernel generation only) and Pluto [Bondhugula et al. 2008] (evolved into PPCG), these are the only fully automatic compilers generating CUDA code directly from C. Tiling for PPCG is set to default to keep the results fully automatic for all compilers: Note that manual tuning could potentially improve performance in favor of PPCG.

7.1. Evaluating Compiler Optimizations

Section 4.2 introduced thread coarsening, register caching, and zero-copy host-accelerator transfers. Furthermore, Sections 5 and 6 introduced host-accelerator transfer optimizations and kernel fusion. This section evaluates these optimizations.

Thread coarsening is evaluated for the GPU-CUDA target. Figure 14 shows the speed-ups when applying coarsening ($2\times$, $4\times$, and $8\times$) over noncoarsened code. Several benchmarks (`2mm`, `3mm`, `gemm`, `syr2k`, and `syrk`) benefit significantly from coarsening. In fact, these benchmarks are the cases for which coarsening is enabled by default based on the species (see Section 4.2 for details) with a factor 4 (based on experimental

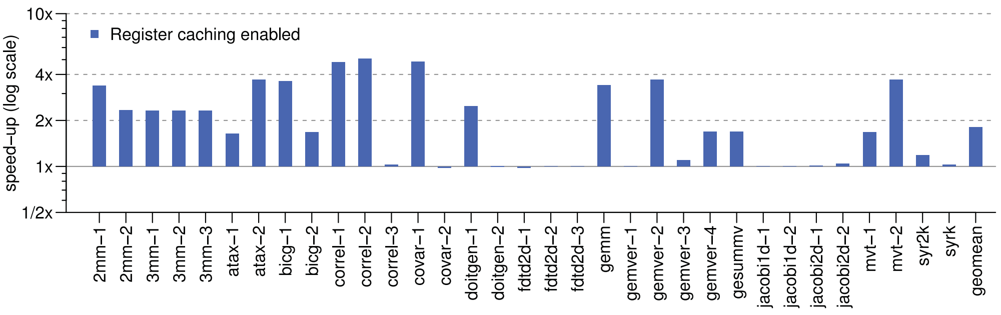


Fig. 15. Speed-ups of code with **register caching** enabled over code without register caching for the GPU-CUDA target.

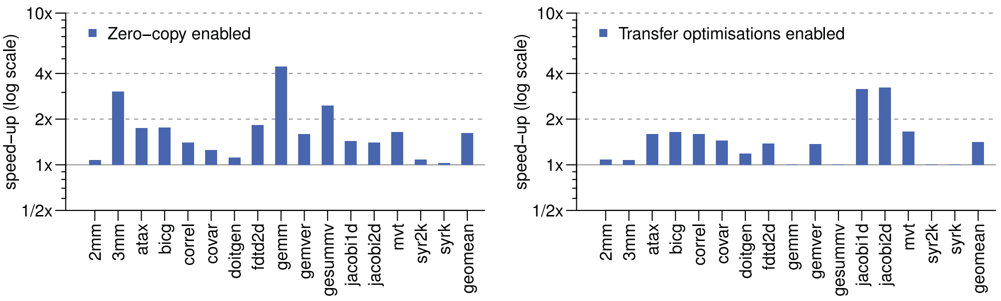


Fig. 16. Speed-ups of code with **zero-copy** enabled over code with data transfers for the Intel OpenCL CPU target (left) and speed-ups of **host-accelerator transfer optimizations** over naive data transfers for the GPU-CUDA target (right). Execution time of the entire benchmark is considered rather than performance of the individual kernels.

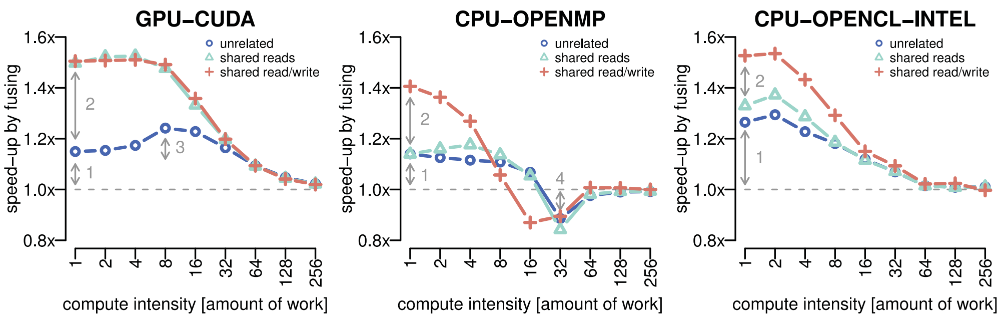
observations). Several other benchmarks that use a coarsening factor of 4 per default benefit only slightly: `doitgen`, `fdtd2d`, `jacobi1d`, and `jacobi2d`. Other benchmarks (not using coarsening per default) are either not affected or suffer from performance loss because of a reduced amount of parallelism and/or degraded cache performance. A detailed study of the effects of coarsening is beyond the scope of this work. Other work (e.g., Magni et al. [2013]) discusses the effects of coarsening on divergent control flow.

Figure 15 shows the impact of register caching for the GPU-CUDA target. The results are either positive or neutral and show a geometric mean speed-up of 1.8×. Register caching is in particular beneficial for PolyBench because its benchmarks favor the use of array references above the introduction of temporary variables. The introduction of such variables can therefore greatly reduce the number of memory accesses.

Zero-copy performs a pointer-only copy for OpenCL programs with a shared memory. The left-hand side of Figure 16 shows the benefits of omitting data transfers for the Intel OpenCL CPU target. The benefit is minimal for cases where the kernel execution time dominates the total execution time (e.g., `syr2k` and `syrk`). In other cases (e.g., `jacobi1d` and `jacobi2d`), the pointer-copies save significant time. In a few cases (e.g., `3mm`), the elimination of the data transfers can even improve cache behavior.

The right-hand side of Figure 16 shows the speed-ups obtained by performing the host-accelerator data transfer optimizations as described in Section 5. The speed-up is significant in cases where data transfers can be moved outside of loops that contain species (e.g., `jacobi1d` and `jacobi2d`). Overall, the geometric mean speed-up is 1.4×.

Kernel fusion is evaluated in two parts. First, three combinations of two artificial loop nests are evaluated, which are shown in the bottom half of Figure 17. The first



```
for(i=0; i<N; i++)
  B[i] = work( A[i] );
for(i=0; i<N; i++)
  D[i] = work( C[i] );
```

```
for(i=0; i<N; i++)
  B[i] = work( A[i] );
for(i=0; i<N; i++)
  D[i] = work( A[i] );
```

```
for(i=0; i<N; i++)
  B[i] = work( A[i] );
for(i=0; i<N; i++)
  D[i] = work( B[i] );
```

Fig. 17. Speed-up results when applying **kernel fusion** for three different targets (top) for artificially created code (bottom): two unrelated loop nests (bottom left), two loop nests sharing reads from `A` (bottom middle), and two loop nests sharing accesses to `B` (bottom right). The numbered arrows in the graphs refer to phenomena discussed in Section 7.1.

two loops contain unrelated computations (`B` $\leftarrow$ `A` and `D` $\leftarrow$ `C`), the second set shares a read (`B` $\leftarrow$ `A` and `D` $\leftarrow$ `A`), and the last two share a read/write access (`B` $\leftarrow$ `A` and `D` $\leftarrow$ `B`). The dimension `N` is set to 2048^2, and the `work` function performs a configurable amount of self-multiplications after loading the input into the register (e.g. `r*r*r`). The top half of Figure 17 shows the speed-ups of fused code over nonfused code for three targets, timing kernel execution only. The following observations are made:

(1) There is a measurable gain when fusing two kernels, even when they are unrelated. This is mainly due to the increased amount of work per thread, allowing for a more efficient execution. For example, common instructions can be shared (e.g., the calculation of the thread index), the compiler has a larger scope for optimizations, and latencies can be hidden more effectively. Kernel launch overhead is also reduced, which is particularly beneficial for OpenCL.
(2) Fusing two kernels can be beneficial for data locality when they access the same data. For the GPU-CUDA target, this saves expensive off-chip accesses for both sharing cases. For the CPU cases, however, the only performance increase comes from a shared read/write: OpenMP and OpenCL load the value of the read twice.
(3) The benefit of fusion for two unrelated CUDA kernels actually increases when reaching a moderate amount of work (8–16). This is due to the fact that the (memory) latencies can be hidden better. The relative benefit of fusion decreases as work increases further because the computations themselves become the bottleneck.
(4) Fusion can also decrease performance, as is observed in the OpenMP case. Performance of the fused code is slightly worse (by around 10%) at the transition between memory bandwidth-limited execution and compute-limited execution (amount of work ±32). This could be due to the increased register usage of the fused version.

Second, kernel fusion is evaluated on the PolyBench suite. Based on algorithm 1, four cases of fusion are found: `3mm` (first two kernels), `bicg` (both kernels), `correl` (first two kernels), and `mvt` (both kernels). In other cases, loop fusion in general is possible (as available in PPCG), but kernel fusion in the context of species is not. An example is `2mm`, which allows the outer loops of the two kernels to be fused at the cost of parallelism: The inner loops now become part of the kernel body. Another example is `atax`, which allows fusion after loop interchange of the second loop nest, again at the cost of parallelism.

Table III. Speed-ups when Applying Kernel Fusion to PolyBench

Target	3mm	bicg	correl	mvt
GPU-CUDA	1.00x	0.61x	1.01x	0.52x
CPU-OPENMP	0.96x	1.08x	1.00x	1.00x
CPU-OPENCL-INTEL	0.38x	1.07x	1.08x	1.03x

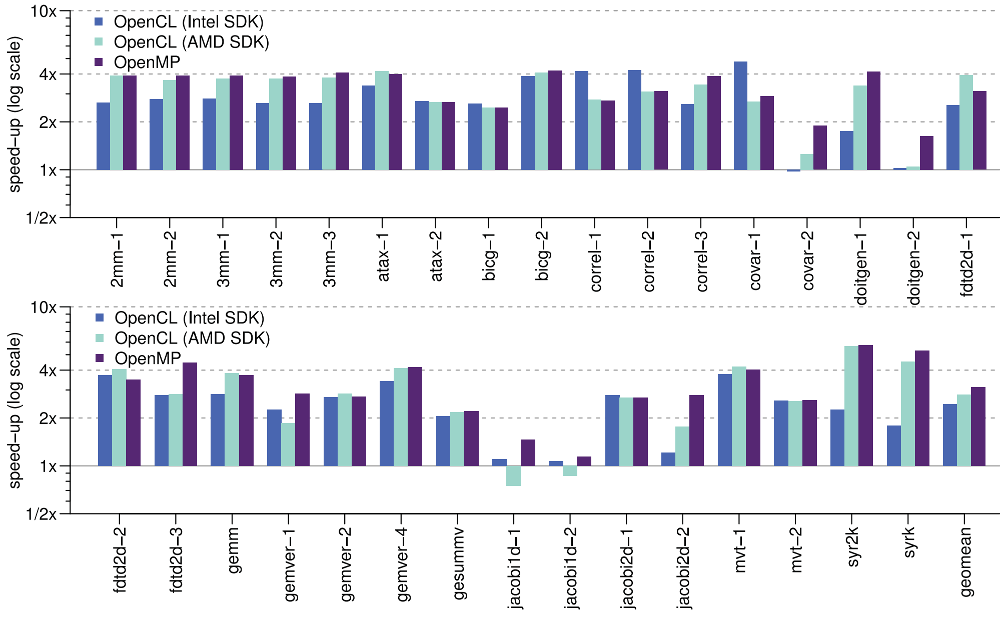


Fig. 18. Comparison of 3 multithreaded CPU targets against single-threaded naive CPU code (the reference input code to BONES).

Table III shows the results of the cases where kernel fusion is legal. In three cases, kernel fusion results in a significant slow-down due to worsened cache performance. In other cases, minor speed-ups are obtained (up to 8%).

7.2. Comparison of Multiple Targets

BONES has three targets that execute on the same hardware: two OpenCL targets and an OpenMP target. Figure 18 shows a comparison of these targets for individual kernels, compared to the reference C code (the input to BONES). On average, this results in speed-ups of 2.4× (Intel SDK OpenCL), 2.8× (AMD SDK OpenCL), and 3.1× (OpenMP). Despite the use of the same hardware, there are still significant performance variations. Differences include the lower thread creation cost for OpenMP, different thread grouping and scheduling policies, and different compilers (and thus also different auto-vectorizers). A detailed comparison of the different targets is beyond the scope of this work but can be found for other benchmarks in Shen et al. [2012]. We furthermore note that performance can be improved by first applying a parallelizing and optimizing compiler, such as Pluto [Bondhugula et al. 2008] or the work by Park et al. [2011].

The PolyBench/C reference code used to produce Figure 18 is naive and unoptimized. In fact, simply changing the default compiler (GCC) to ICC (version 14.0.0) yields speed-ups in most cases (geometric mean 1.5×). This become even more significant when using ICC instead of GCC for the OpenMP target (geometric mean 1.7×).

Next, Figure 19 shows a comparison of performance across different hardware: two GPUs and a CPU. The two orders of magnitude speed-up observed for some of the

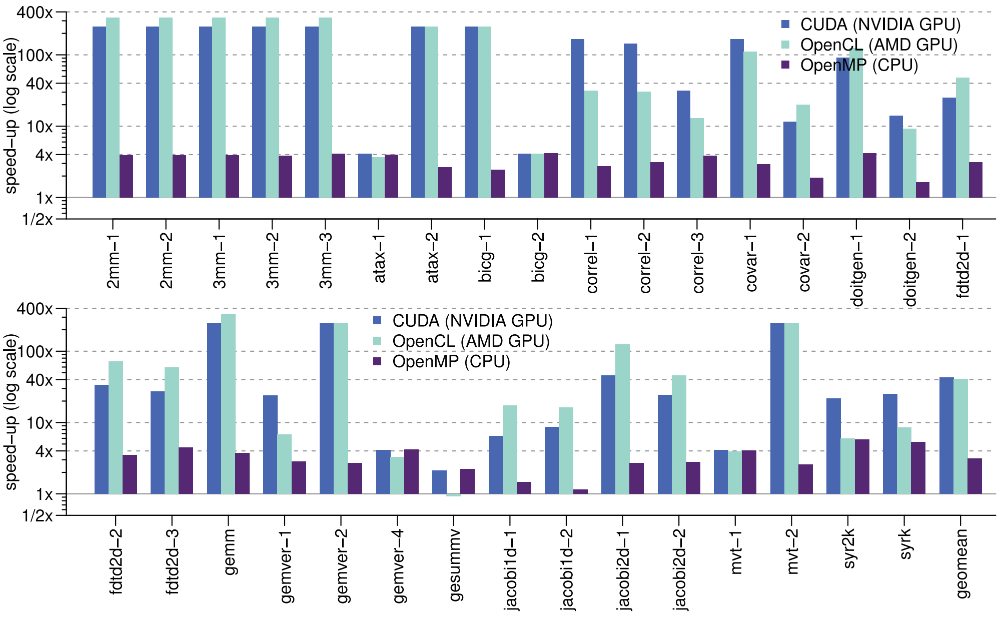


Fig. 19. Comparison of performance across different hardware: an NVIDIA GPU running CUDA code, an AMD GPU running OpenCL code, and a CPU running OpenMP code. The reference is the single-threaded naive CPU code that is used as input to BONES.

GPU targets should be taken with a grain of salt: The comparison is against naive single-threaded CPU code. The two GPUs show comparable performance, a result of the higher theoretical performance of the AMD GPU on one hand and less optimized OpenCL GPU skeletons on the other hand. The GPU targets achieve a geometric mean speed-up of an order of magnitude over OpenMP code, limited by several kernels that do not benefit as much as others from GPU acceleration. Examples are the limited parallelism of `correl-3` and the strided (uncoalesced) memory accesses of `mvt-1`.

7.3. Comparison Against the State-of-the-Art

Two experiments are performed to compare BONES with the state-of-the-art C-to-CUDA compilers PAR4ALL and PPCG. The first experiment evaluates the individual kernels only: Kernels are generated for each of the algorithmic species in isolation using BONES, PAR4ALL, and PPCG. Figure 20 shows the results in terms of speed-up of BONES compared to PAR4ALL and PPCG. The following observations are made:

—BONES shows significantly better performance (2× or more) for 21 kernels compared to PAR4ALL and for 7 kernels compared to PPCG.
—PPCG is measurably faster only for `covar-2` (1.4×). This is a result of parallelization of only the outer loop (in contrast to both loops for BONES), which allows tiling of the inner loop (to improve data locality).
—Several kernels use a skeleton in the form of Figure 10 to ensure coalesced memory accesses, yielding significant speed-ups over both PAR4ALL and PPCG. The advantages over the default skeleton are 2.4× (`atax-1`), 1.8× (`bicg-2`), 2.3× (`gemver-4`), 1.8× (`mvt-1`), and 7.8× (`syrk`). A related skeleton with two preshuffling kernels is used for two other kernels, yielding speed-ups over the default skeleton of 3.2× (`gesummv`) and 15.8× (`syr2k`). The other PolyBench benchmarks use the default skeleton.

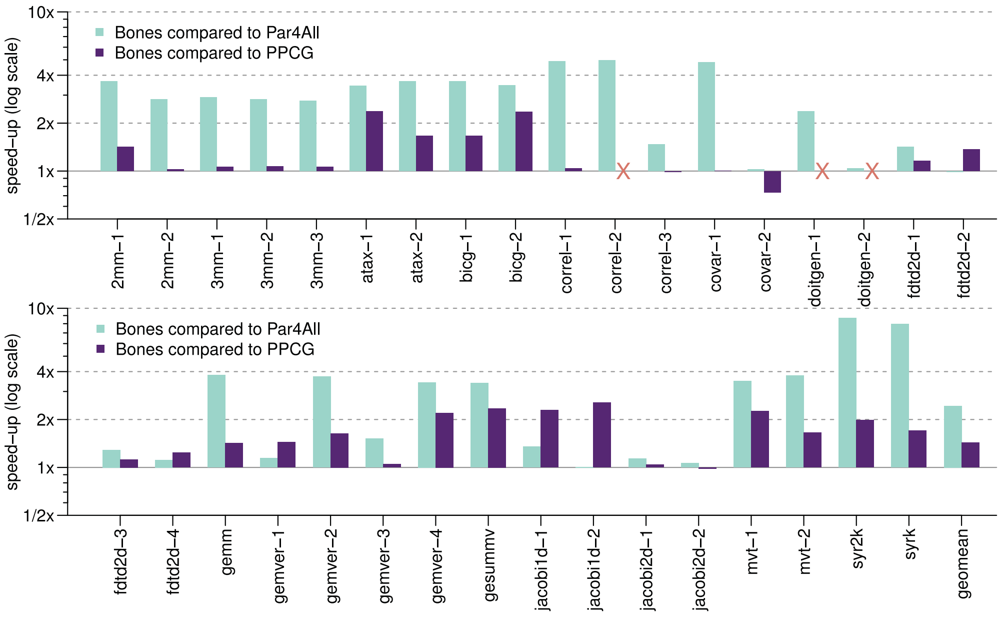


Fig. 20. Performance of BONES compared to PAR4ALL and PPCG for GPU-CUDA (higher is in favor of BONES). PPCG was unable to generate code for `correl-2`, `doitgen-1`, and `doitgen-2` (marked by crosses).

—In the cases of matrix-multiplication variants `2mm` and `3mm`, BONES is on-par with PPCG. In these cases, BONES relies on caches and thread coarsening, whereas PPCG performs loop tiling and uses the local memory.
—In many benchmarks (`2mm`, `3mm`, `doitgen`, `fdtd2d`, `gemm`, `jacobi1d`, `jacobi2d`, `syrk`, `syr2k`), BONES performs thread coarsening, outperforming PAR4ALL. PPCG also performs thread coarsening in most of these cases, but by a factor 2 instead of 4.
—The geometric mean speed-up of kernels generated by BONES is 2.4× compared to PAR4ALL and 1.4× compared to PPCG.

The second set of experiments considers the entire benchmark (except for data-initialisation), which corresponds to the static control part (or "scop") of PolyBench. Figure 21 shows the speed-up of BONES compared to PAR4ALL and PPCG. Additionally, PolyBench/GPU [Grauer-Gray et al. 2012] is included, which provides handwritten nonoptimized CUDA code (hand-optimized is not available). We observe that it is in many cases significantly slower when compared to generated code: on average 7.4× compared to BONES. We furthermore note that the (experimental) option to optimize CPU-GPU transfers for PAR4ALL only supports static arrays and does not work for these benchmarks. The right-hand side of Figure 16 gives an idea of the performance gains future versions of PAR4ALL might achieve when such optimizations are fully implemented. Figure 21 shows that, in almost all cases, performance of BONES is either on-par or better when compared to the state-of-the-art. The geometric mean speed-ups are 1.9× and 1.0× when compared to PAR4ALL and PPCG, respectively. These results are not as good as those of Figure 20. There are two reasons for this:

(1) **Interkernel optimizations:** Kernel fusion is the only interkernel optimization performed by BONES. In contrast, the other compilers are able to distribute loops (fission) and to interchange nested loops (whereas BONES considers the kernels to be fixed). The polyhedral model is well-suited for such optimizations, allowing PPCG

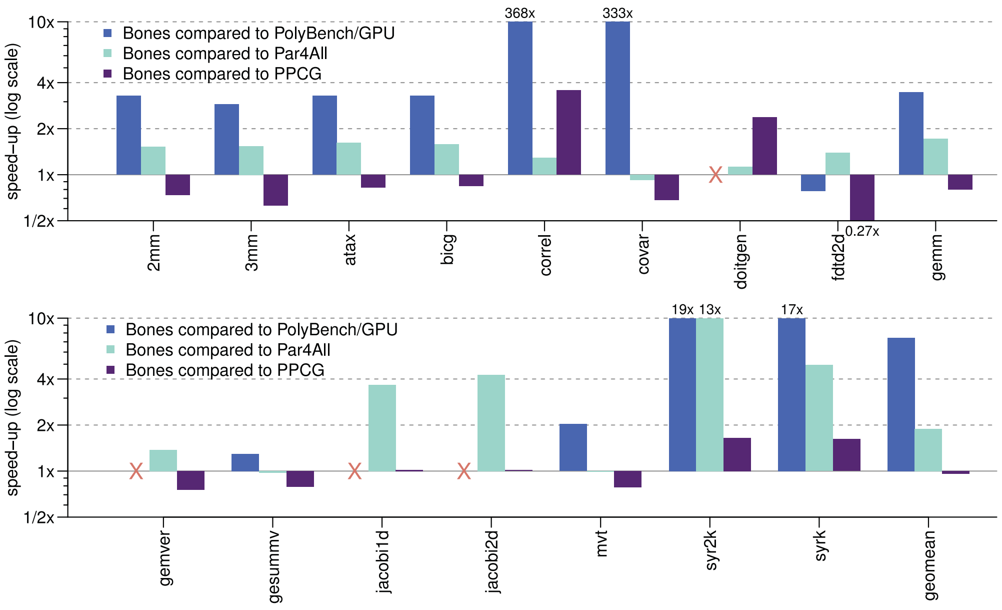


Fig. 21. Comparison of BONES with PAR4ALL, PPCG and handwritten PolyBench/GPU code for the complete benchmarks. PolyBench/GPU is not available for `doitgen`, `gemver`, `jacobi1d`, and `jacobi2d` (crosses).

(for example) to do well on `atax` despite the fact that both of its kernels individually perform worse compared to BONES. For this particular example, PPCG creates four kernels out of the original two, enabling loop interchange and resulting in coalesced accesses. Such optimizations (e.g., distribution) are currently not possible within BONES but could be enabled by using Pluto as a front end.

(2) **Host-accelerator data transfers:** Although BONES already performs host-accelerator transfer optimizations, PPCG is able to find additional optimizations. An example is `fdtd2d`, for which PPCG moves several CPU-GPU transfers to the outer "time" loop. This demonstrates that there is still room for further host-accelerator data transfer optimizations in BONES.

7.4. Discussion: Evaluation on Larger Benchmarks

Since BONES is a proof-of-concept rather than an industrial compiler, some language features and corner-cases are not directly supported by the tool chain. Supporting all cases does not pose fundamental problems but requires significant engineering time. Examples of such cases are C structures, nested preprocessor directives, and do-while loops. To still be able to test benchmarks outside of PolyBench, the smallest 6 out of the 19 CUDA benchmarks from the Rodinia benchmark suite were adapted for this purpose (rewriting do-while loops, separating structures, in-lining function calls). The 6 benchmarks[2] (`bfs`, `hotspot`, `kmeans`, `nw`, `pathfinder`, `srad`) are tested with BONES and compared with PAR4ALL, PPCG, and the corresponding Rodinia CUDA versions (although they are hand-written, they are not fully tuned for performance).

The performance results are shown in Figure 22, with individual kernels on the left and the full benchmark on the right. Several observations are made:

[2]C99 versions are included as part of the BONES examples at http://github.com/cnugteren/bones/.

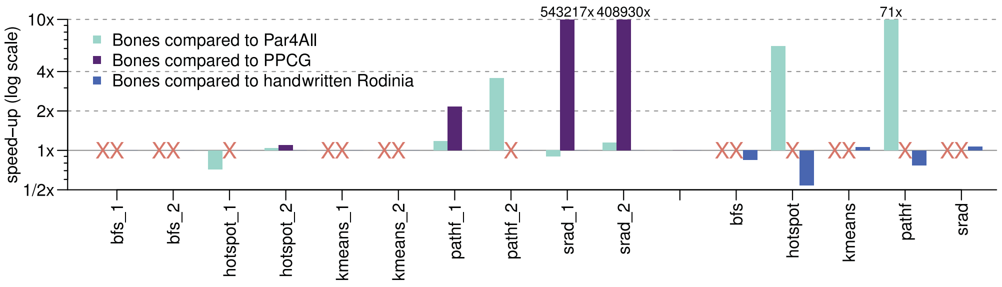


Fig. 22. Performance of BONES's GPU-CUDA target compared to PAR4ALL and PPCG for individual Rodinia kernels (left) and for the entire application (right: also compared against handwritten code). Higher is in favor of BONES. Compilation failures of PAR4ALL and PPCG are marked by a red cross.

—All three compilers were unable to generate parallel code for `nw` (not shown in the figure). This is caused by the data dependencies in the inner loops of the basic implementation: A manual rewrite of the algorithm will be required.
—The poor results of PPCG for `srad` are caused by running the two parallelizable loops on the CPU and launching multiple GPU kernels without parallelism.
—The `bfs` and `kmeans` benchmarks contain indirect memory accesses, which makes them fall outside the scope of the polyhedral model: There are no results for PAR4ALL and PPCG. When automatically classifying these benchmarks with species, A-DARWIN asks the user for the bounds of these specific variables, enabling successful (semiautomatic) parallelization with BONES.
—For all five entire benchmarks (on the right in the figure), PPCG is unable to generate CUDA code. PAR4ALL does succeed in two cases, but performance results are worse compared to BONES due to CPU-GPU data transfers in the inner loops.
—BONES generates code that performs close to or is on par with handwritten Rodinia GPU code (right-hand side of the figure; on average 1.2× slower). Although these benchmarks were explicitly supported by BONES, this proves the concept and the approach that has been presented in this work. Of course, further engineering effort will be required to support other (Rodinia) benchmarks out of the box.

8. DISCUSSION

The integration of algorithmic species with a skeleton-based compiler has resulted in a unique source-to-source compilation approach. This novel combination can be seen as a way to profit from the benefits of skeletons without their main drawbacks.

Skeleton-based compilation has several benefits. First, compilation requires only basic transformations that can be performed at the abstract syntax tree level, omitting the need for intermediate representations that often lose code structure and variable naming. This allows the compiler to generate readable code, enabling opportunities for further fine tuning and manual optimization. Furthermore, the skeletons themselves can be formatted to include structure and code comments to improve readability. Second, skeleton-based compilation benefits from the flexibility of being able to improve the compiler (by modifying the skeletons) or extend to other targets (by writing new skeletons). An example is the recent addition of an FPGA High-Level Synthesis (HLS) target. Finally, several "optimizations" within skeletons are not permutations of the original code. An example is the additional preprocessing kernel of Figure 10 (lines 10–21), which cannot be described as a transformation of the original code.

Compared to other skeleton-based compilers, BONES is the first that can be used in an automatic tool chain because of the integration of algorithmic species. This removes the requirements of existing skeleton approaches (e.g., SkePU and SkeCL) to manually

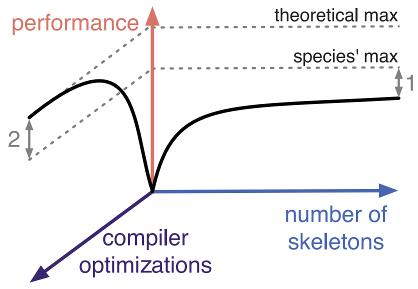


Fig. 23. Comparison of the theoretical achievable performance of traditional compiler optimizations with skeleton-based optimizations. The numbers refer to performance limitations discussed in this section.

select skeletons and perform code modifications. On top of this, the integration with algorithmic species provides a clear, structured, and formally defined way of using skeletons, which can be beneficial in cases where manual classification is unavoidable.

However, the possibility of generating efficient code using skeletons and species is limited. This is illustrated by Figure 23, numbering the two main reasons: (1) not all optimizations can be expressed in the form of a skeleton, and (2) algorithmic species do not contain all performance-relevant details. An example of the first is thread coarsening, which can be applied (yes/no) based on the species but cannot be implemented in the form of a skeleton. An example of the second is register caching, which is a traditional compiler optimization independent of the code's algorithmic species. BONES is therefore designed as a combination between a skeleton-based compiler (with the discussed benefits) and a traditional compiler (allowing for competitive performance).

Another aspect of skeleton-based compilation shown in Figure 23 is performance with respect to the number of (optimised) skeletons implemented. The more skeletons provided for the different species, the higher the aggregated theoretical performance. However, performance is limited by the earlier mentioned aspects (the two numbers in Figure 23), but with a good reason: If an increasing amount of detail is captured by the algorithm classification, implementing skeletons converges toward implementing a library. In practise, a small number of skeletons can already deliver good performance (combined with compiler optimizations). An example is the "default" skeleton for CUDA, which maps the species' parallel loop iterations onto GPU threads. For this target and for the PolyBench suite, the additional performance benefit of the four specialized skeletons has a geometric mean of 1.3×, ranging from 1.0× to 15.8×.

Apart from being a combination of a skeleton-based compiler and a traditional compiler, BONES is also unique in the sense that it explicitly uses species as an intermediate step. In this way, code analysis (A-DARWIN) is separated from the actual compilation (BONES). This has two main advantages: (1) either of the two steps can be replaced or reused in other work, and (2) the programmer can help the compiler where needed by modifying species or manually adding species to unclassified code.

We make a final note that BONES is a proof-of-concept: Many of the optimizations discussed in this work are in reality complex problems that require thorough analysis. Examples are thread coarsening for divergent code [Magni et al. 2013], host-accelerator transfer optimizations [Guelton et al. 2012; Jablin et al. 2012], and kernel fusion [Darte 2000; Kennedy and McKinley 1994; Pouchet et al. 2010]. Nonetheless, the experimental results for PolyBench have shown that the combination of skeletons and compiler optimizations within BONES can deliver competitive performance. In comparison to two state-of-the-art automatic compilers (PAR4ALL and PPCG), BONES obtains geometric mean speed-ups of 2.4× and 1.4× for CUDA kernels. Future work will be required to verify these numbers for case studies and other benchmarks.

9. SUMMARY

BONES is a new source-to-source compiler based on the algorithmic species classification. The compiler transforms sequential C code into CUDA, OpenCL, or OpenMP by providing a preoptimized template implementation (an "algorithmic skeleton") for each algorithmic species. Previous to this work, compilers based on algorithmic skeletons used "classifications" without formal definitions, which were meant to be extended as new classes were encountered. This is no longer the case with integrating algorithmic species within BONES. Furthermore, by automating the extraction of species from source code, BONES is the first skeleton-based compiler integrated in a fully automatic flow. Although most of the code generation is performed based on skeletons within BONES, the compiler also includes traditional optimization passes such as register caching, thread coarsening, host-accelerator transfer optimizations, and kernel fusion. Section 7 showed the importance of these optimizations: A compiler relying on skeletons alone will not be able to deliver competitive performance. Compared to two state-of-the-art C-to-CUDA compilers, BONES is able to achieve geometric mean speed-ups of 1.4× (versus PPCG) and 2.4× (versus PAR4ALL) for CUDA kernels in the PolyBench/C suite. Due to the lack of extensive interkernel optimizations, this reduces to speed-ups of 1.0× and 1.9× for complete CUDA programs. For five Rodinia benchmarks, BONES leaves an average gap of only 1.2× compared to hand-optimized code. Because BONES is based on an abstract classification and is limited to static analysis, there is still room for optimizations after compilation. Therefore, BONES generates editable and readable source code, allowing further fine tuning by expert programmers or autotuners.

REFERENCES

Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 2013. FastFlow: High-level and efficient streaming on multi-core. *Programming Multi-core and Many-core Computing Systems* 13 (January 2013). Wiley.

Mehdi Amini, Béatrice Creusillet, Stéphanie Even, Ronan Keryell, Onig Goubier, Serge Guelton, Janice Mcmahon, François-Xavier Pasquier, Grégoire Péan, and Pierre Villalon. 2012. Par4All: From convex array regions to heterogeneous computing. In *Proceedings of the IMPACT Workshop*.

Soufiane Baghdadi, Armin Größlinger, and Albert Cohen. 2010. Putting automatic polyhedral compilation for GPGPU to work. In *Proceedings of the CPC Workshop*. INRIA.

Muthu Baskaran, J. Ramanujam, and P. Sadayappan. 2010. Automatic C-to-CUDA code generation for affine programs. In *Proceedings of the International Conference on Compiler Construction*. Springer.

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the International Conference on Programming Language Design and Implementation*. ACM.

Murray Cole. 1991. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press.

Batrice Creusillet and Franois Irigoin. 1997. Exact versus approximate array region analyses. In *Proceedings of the LCPC Workshop*. Springer.

Alain Darte. 2000. On the complexity of loop fusion. *Parallel Computing* 26, 9 (2000), 1175–1193.

Usman Dastgeer, Lu Li, and Christoph Kessler. 2013. Adaptive implementation selection in the skepu skeleton programming library. In *Proceedings of the APPT Conference*. Springer.

Johan Enmyren and Christoph Kessler. 2010. SkePU: A multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the International Conference on High-level Parallel Programming and Applications*. ACM.

Steffen Ernsting and Herbert Kuchen. 2011. Data parallel skeletons for GPU clusters and multi-GPU systems. *Advances in Parallel Computing* 22 (2011), 509–518.

Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *Proceedings of INPAR*.

Serge Guelton, Mehdi Amini, and Béatrice Creusillet. 2012. Beyond do loops: Data transfer generation with convex array regions. In *Proceedings of the LCPC Workshop*. Springer.

Tianyi David Han and Tarek S. Abdelrahman. 2011. hiCUDA: High-level GPGPU programming. *IEEE Transactions on Parallel and Distributed Systems* 22 (January 2011), 78–90.

Thomas Jablin, James Jablin, Prakash Prabhu, Feng Liu, and David August. 2012. Dynamically managed data for CPU-GPU architectures. In *Proceedings of the International Conference on Code Generation and Optimization*. ACM.

Ken Kennedy and Kathryn McKinley. 1994. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of the LCPC Workshop*. Springer.

Yunsup Lee, Ronny Krashinsky, Vinod Grover, Stephen Keckler, and Krste Asanovic. 2013. Convergence and scalarization for data-parallel architectures. In *Proceedings of the International Conference on Code Generation and Optimization*. IEEE.

Alberto Magni, Christophe Dubach, and Michael O'Boyle. 2013. A large-scale cross-architecture evaluation of thread-coarsening. In *Proceedings of SC*. ACM.

Gabriel Noaje, Christophe Jaillet, and Michael Krajecki. 2011. Source-to-source code translator: OpenMP C to CUDA. In *Proceedings of the International Conference on High Performance Computing and Communications*. IEEE.

Cedric Nugteren, Rosilde Corvino, and Henk Corporaal. 2013a. Algorithmic species revisited: A program code classification based on array references. In *Proceedings of the International Workshop on Multi-/Many-core Computing Systems*. IEEE.

Cedric Nugteren, Pieter Custers, and Henk Corporaal. 2013b. Algorithmic species: An algorithm classification of affine loop nests for parallel programming. *ACM Transactions on Architecture and Code Optimisations* 9, 4 (2013), Article 40.

Cedric Nugteren, Pieter Custers, and Henk Corporaal. 2013c. Automatic skeleton-based compilation through integration with an algorithm classification. In *Proceedings of the APPT Conference*. Springer.

Eunjung Park, Louis-Noel Pouchet, John Cavazos, Albert Cohen, and P. Sadayappan. 2011. Predictive modeling in a polyhedral optimization space. In *Proceedings of the International Conference on Code Generation and Optimization*. IEEE.

Louis-Noel Pouchet, Uday Bondhugula, Cedric Bastoul, Albert Cohen, J. Ramanujam, and P. Sadayappan. 2010. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Proceedings of SC*. IEEE.

Louis-Noel Pouchet and Scott Grauer-Gray. 2013. PolyBench: The Polyhedral Benchmark Suite. (2013). On-line: http://www.cse.ohio-state.edu/~pouchet/software/polybench/.

Shigeyuki Sato and Hideya Iwasaki. 2009. A skeletal parallel framework with fusion optimizer for GPGPU programming. In *Proceedings of the Asian Symposium on Programming Languages and Systems*.

Jie Shen, Jianbin Fang, Henk Sips, and Ana-Lucia Varbanescu. 2012. Performance gaps between OpenMP and openCL for multi-core CPUs. In *Proceedings of the International Conference on Parallel Processing Workshops*. IEEE.

Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. 2011. SkelCL - A portable skeleton library for high-level GPU programming. In *Proceedings of the International Symposium on Parallel and Distributed Processing Workshops*. IEEE.

Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimisations* 9, 4, Article 54 (Jan. 2013).

Michael Wolfe. 2010. Implementing the PGI accelerator model. In *Proceedings of the Workshop on General Purpose Processing on Graphics Processing Units*. ACM.

Received November 2013; revised June 2014; accepted August 2014