Automatic Skeleton-Based Compilation through Integration with an Algorithm Classification

Cedric Nugteren, Pieter Custers, and Henk Corporaal

Eindhoven University of Technology, The Netherlands
c.nugteren@tue.nl, pieterjjmcusters@gmail.com, h.corporaal@tue.nl

Abstract. This paper presents a technique to fully automatically generate efficient and readable code for parallel processors. We base our approach on skeleton-based compilation and '*algorithmic species*', an algorithm classification of program code. We use a tool to automatically annotate C code with species information where possible. The annotated program code is subsequently fed into the skeleton-based source-to-source compiler 'Bones', which generates OpenMP, OpenCL or CUDA code and optimises host-accelerator transfers. This results in a unique approach, integrating a skeleton-based compiler for the first time into an automated flow. We demonstrate the benefits of our approach on the PolyBench suite by showing average speed-ups of 1.4x and 1.6x for GPU code compared to ppcg and Par4All, two state-of-the-art compilers.

Keywords: Parallel Programming, Algorithm Classification, Algorithmic Skeletons, Source-to-Source Compilation, GPUs

1 Introduction

The past decades of processor design have led to an increasingly heterogeneous computing environment, in which multi-core CPUs 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 parallel languages and are required to have detailed architectural knowledge to fully optimise their applications. Apart from programming, maintainability, portability in general, and performance portability in particular have become issues of major importance. Despite a significant amount of work on compilation, auto-parallelisation and auto-tuning (e.g. [2, 6, 7, 11, 13, 21, 22, 23]), many programmers are still struggling with these issues, having to deal with low-level languages such as OpenCL and CUDA.

Existing compilers for parallel targets fall short in at least one of the following areas: 1) they are not fully automatic and require code 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 architectures and concurrent programming, while the second shortcoming mostly affects savvy

programmers who are leveraging compilers to perform the initial parallelisation and are willing to further optimise the resulting code. The third shortcoming affects all types of users. The goal of this work is to address these shortcomings. We take a unique approach: we use an algorithm classification to drive a source-to-source compiler based on *algorithmic skeletons*.

In this work we present a technique to automatically generate efficient and readable parallel code for parallel architectures (with a focus on GPUs). We base this technique on '*algorithmic species*' [16, 17], an algorithm classification of program code based on the polyhedral model [8]. Algorithmic species encapsulate information such as data re-use and memory access patterns. Algorithmic species form the backbone of our approach (see Fig. 1), which includes a tool to automatically extract species from static affine loops (ASET) and a source-to-source compiler based on skeletons (BONES). The contributions of this work are summarised as follows:

- We present a unique integration of a skeleton-based compiler (BONES) with an algorithm classification (algorithmic species) in Sect. 3. With this combination, skeleton-based compilers can be used in fully-automatic compilation flows, as they no longer require manual identification of skeletons.
- We optimise host-accelerator transfers (e.g. CPU-GPU) in ASET and extend BONES as presented in [15] with new targets, skeletons and optimisations, including register caching, thread coarsening, and zero-copy transfers (Sect. 4).
- We discuss and demonstrate the benefits of our unique approach (Sect. 5) by generating OpenMP, OpenCL and CUDA code for the PolyBench benchmark suite. We focus on the CUDA target, for which we show a speed-up compared to two state-of-the-art polyhedral compilers for individual kernels (1.4x and 1.6x on average) and for complete benchmarks (1.2x and 3.0x on average). Additionally, we demonstrate the importance of host-accelerator transfer optimisations (1.8x speed-up on average).

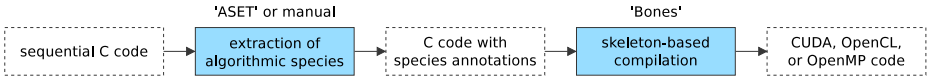

Fig. 1. Overview of the approach taken in this work.

2 Related Work

There is a large body of work targeted at (partially) automating parallel programming (e.g. [1, 2, 6, 7, 11, 13, 15, 21, 22, 23]). However, existing work targeting code generation for OpenMP, OpenCL or CUDA often requires annotations in the form of directives [6, 11, 15, 23] or requires major code restructuring [7, 21]. Furthermore, they often produce human unreadable source code for further optimisations [6, 23] or no modifiable source code at all [7]. Exceptions are the

polyhedral-based compilers Par4All [1] and ppcg [22] that are able to fully-automatically compile sequential code into readable parallel code. We compare our approach in terms of performance with these state-of-the-art compilers for CUDA in Sect. 5. Par4All [1] is based on the theory of convex array regions and is implemented using the PIPS source-to-source compiler. ppcg [22] is an optimising polyhedral compiler based on PET and ISL. Both Par4All and ppcg are able to perform host-accelerator (e.g. CPU-GPU) transfer optimisations.

Among the large amounts of existing algorithm classifications (e.g. [4, 7]), we identify only one classification, named *idioms* [18], that provides a tool to perform automatic extraction of classes from source code. However, only six basic classes are provided (stream, transpose, gather, scatter, reduction and stencil), resulting in a significantly lower amount of detail compared to algorithmic species and aset's automatic extraction.

3 Combining Skeletons with Algorithmic Species

In our two-step approach (see Fig. 1), the first step is to extract relevant information from source code. This information is encapsulated in the form of *algorithmic species*[1] [16, 17]: an algorithm classification based on polyhedral analysis [8]. In this section, we first briefly provide background on algorithmic species and algorithmic skeletons. Following, we discuss the combination of skeleton-based compilation with algorithmic species. Finally, we illustrate the integration of skeleton-based compilation and species by discussing example skeletons.

3.1 Classifying Code using Algorithmic Species

We illustrate algorithmic species by showing an example: matrix-vector multiplication $\boldsymbol{r} = \mathbf{M} \cdot \boldsymbol{s}$ (see Listing 1). To produce a single *element* of the result $\boldsymbol{r}$, we need a complete row from matrix $\mathbf{M}$ and the entire vector $\boldsymbol{s}$. In terms of access patterns used to build algorithmic species, we name the row access from $\mathbf{M}$ a *chunk* access. The vector $\boldsymbol{s}$ is needed entirely to calculate a single element of the result $\boldsymbol{r}$, which we characterise with the *full* pattern. The final algorithmic species is shown in line 1 of Listing 1, including array names and access ranges.

Listing 1. Matrix-vector multiplication classified using algorithmic species.

```
1 M[0:127,0:63]|chunk(-,0:63) ∧ s[0:63]|full → r[0:127]|element
2 for (i=0; i<128; i++) {
3   r[i] = 0;
4   for (j=0; j<64; j++) {
5     r[i] += M[i][j] * s[j];
6   }
7 }
```

The example covers three access patterns, forming a single species when combined. In total, five patterns (*element*, *neighbourhood*, *chunk*, *full*, and *shared*) can be combined into an unlimited amount of different species. Species and their

[1] Species is both the English plural and singular form.

patterns are defined formally based on the polyhedral model. Since the patterns are descriptive and intuitive, they may be derived from source code by hand in certain cases. However, to ease the work of a programmer, to avoid mistakes, and to be able to automatically generate parallel code, species are derived automatically using ASET, an algorithmic species extraction tool [5][2]. We make a note that this tool merely identifies species in source code, it does not perform any transformations to extract (more) parallelism from the code. Therefore, our two-step approach could be combined with existing parallelising compilers (e.g. [19]), which can be seen as additional pre-processing.

3.2 Compilation Based on Algorithmic Skeletons

Algorithmic skeletons [4] is a technique that uses parametrisable 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 processor. Users of previous skeleton-based compilers were required to select a skeleton suitable for their algorithm and target processor by hand, and could subsequently invoke the skeleton to generate program code for the target processor. If no skeleton implementation was available for the specific class or processor, it could be added manually. Future algorithms of the same class could then benefit from re-use of the skeleton code. Benefits of skeleton-based compilation are among others the flexibility to extend to other targets, and the performance potential: optimisations can be performed in the native language within the skeletons. Examples of recent skeleton-based compilers are [3, 7, 15, 21].

3.3 From Species to Skeletons

In contrast to existing skeleton-based compilers, we use algorithmic species information to select a suitable skeleton. This does not only enable automation of the whole tool-chain, but also overcomes common critique for skeleton-based compilers illustrated by questions such as 'how difficult is it to select a suitable skeleton' or 'what if the user selects an incompatible skeleton'.

We modified the BONES skeleton-based source-to-source compiler [15] to be combined with algorithmic species. The compiler takes C code annotated with species information as an input. The algorithmic species (extracted by a pre-processor) are used directly to determine the skeleton to use. Additionally, they are used to enable/disable additional transformations and optimisations, although most optimisations can be made within the skeletons themselves.

Algorithmic species map one-to-one to skeletons: the compiler needs to supply a skeleton for every species it wants to support. These skeletons should be constructed in such a way that they are correct (and preferably optimised) for all algorithms belonging to a given species. For practical reasons however (to save work/code duplication), we provide a species-to-skeleton mapping, such that

[2] ASET can be replaced by the more recent A-DARWIN tool [16].

multiple species can map to the same skeleton. For example, for a GPU target, algorithmic species of the form '`neighbourhood` → `element`' and '`neighbourhood` ∧ `element` → `element`' both map to a single skeleton that enables explicit caching of the neighbourhood in the GPU's scratchpad memory.

We illustrate the working of Bones through Fig. 2. In this figure, we show an example input with two loop nests identified as two different species ('species X' and 'species Y' in the figure). The compiler first loads and invokes the corresponding skeletons for a given target. Then, the skeleton-specific transformations are performed, and finally, Bones combines the results to obtain target code. The optimisations as described in Sect. 4 are performed in the last stage.

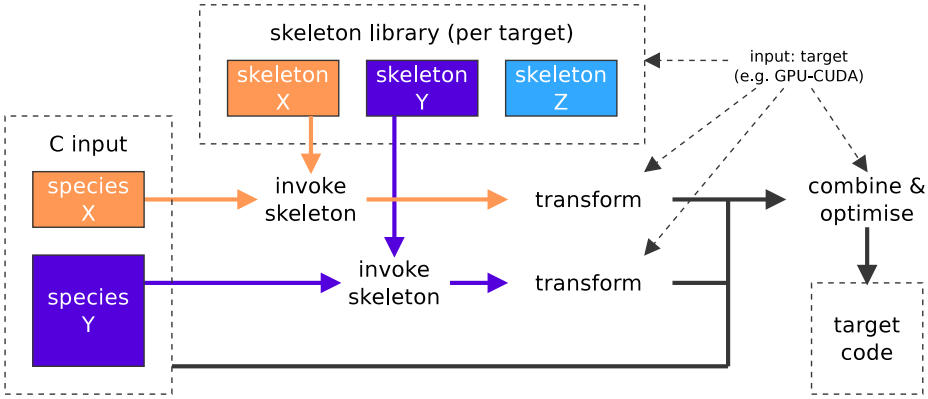

Fig. 2. Illustration of the structure of the Bones compiler for an example input with two different species.

3.4 Example Skeletons

To illustrate the use of skeletons within Bones, we first show a simplified OpenMP skeleton in Listing 2 and its instantiation for the matrix-vector multiplication example (Listing 1) in Listing 3. The skeleton in Listing 2 shows highlighted keywords, which are filled in as follows: `parallelism` represents the parallelism found in the species, `ids` computes the identifier corresponding to the current iteration, and `code` fills in the transformed code. For illustration purposes, the example skeleton is heavily simplified, excluding comments, boundary and initialisation code, function calls and definitions, and makes several assumptions, such as the divisibility of the amount of parallelism by the thread count.

Additionally, we show an example of a skeleton for the CUDA target in Listing 4. 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 kernel shown in Listing 1. A naive mapping to CUDA will result in *uncoalesced* accesses to the *chunk* array (**M** in the example): subsequent accesses will be made by the same

Listing 2. A simplified skeleton for OpenMP. Details are left out for clarity.

```
1 int count;
2 count = omp_get_num_procs();
3 omp_set_num_threads(count);
4 #pragma omp parallel
5 {
6   int tid, i;
7   int work, start, end;
8   tid = omp_get_thread_num();
9   work = <parallelism>/count;
10  start = tid*work;
11  end = (tid+1)*work;
12
13  // Start the parallel work
14  for(i=start; i<end; i++) {
15    <ids>
16    <code>
17
18
19
20 }
```

Listing 3. Instantiated code for the Listing 1 example. Optimisations are omitted.

```
1 int count;
2 count = omp_get_num_procs();
3 omp_set_num_threads(count);
4 #pragma omp parallel
5 {
6   int tid, i;
7   int work, start, end;
8   tid = omp_get_thread_num();
9   work = 128/count;
10  start = tid*work;
11  end = (tid+1)*work;
12
13  // Start the parallel work
14  for(i=start; i<end; i++) {
15    int gid = i;
16    r[gid] = 0;
17    for (j=0; j<128; j++)
18      r[gid] += M[gid][j]*s[j];
19  }
20 }
```

thread. To re-enable coalescing in these cases, which is paramount for performance, a special skeleton with a pre-shuffling kernel is designed. The skeleton in Listing 4 shows a kernel for the actual work (lines 1-8) and a kernel to reorder the input array by re-arranging data in the on-chip memory of the GPU (lines 9-20). The use of this skeleton implies a transformation in the original code as well (e.g. from `M[i][j]` into `M[j][i]`), which is handled by the compiler. Again, this skeleton is heavily simplified for illustration purposes, e.g. not showing boundary checks nor the host code to launch the kernels.

Listing 4. A simplified skeleton for the CUDA target to enable coalesced accesses.

```
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>) {
5     <ids>
6     <code>
7   }
8 }
9 // CUDA kernel for pre-shuffling (simplified)
10 __global__ void kernel1(...) {
11   int tx = threadIdx.x; int ty = threadIdx.y;
12   __shared__ <type> b[16][16];
13   int gid0 = blockIdx.x*blockDim.x + tx;
14   int gid1 = blockIdx.y*blockDim.y + ty;
15   int nid0 = blockIdx.y*blockDim.y + tx;
16   int nid1 = blockIdx.x*blockDim.x + ty;
17   b[ty][tx] = in[gid0 + gid1*<dims>/<params>];
18   __syncthreads();
19   out[nid0 + nid1*<params>] = b[tx][ty];
20 }
```

4 Compiler Optimisations

In this section we discuss the host-accelerator transfer optimisations made by ASET and BONES, and illustrate some of the optimisations possible within BONES. The discussed optimisations in this section are new to BONES: they are not present in earlier non-species based versions [15]. Both ASET, and BONES are open-source and are freely available[3]. The compiler BONES is programmed in Ruby and uses the C-to-AST module CAST[4]. BONES currently supplies a total of 15 skeletons for 5 different targets: OpenMP for CPUs, CUDA for NVIDIA GPUs, OpenCL for AMD GPUs, and OpenCL for CPUs (AMD and Intel SDK).

4.1 Host-Accelerator Transfer Optimisations

Many of today's parallel processors are designed as an *accelerator*: they require a *host* processor to dispatch tasks (or: kernels). Furthermore, they often have a separate memory (e.g. GPUs, Intel MIC), requiring host-accelerator transfers of input and output arrays. When executing multiple kernels, this gives opportunities to optimise these transfers in several ways [10, 12]: 1) transfers might be omitted (e.g. subsequent kernels use the same data), 2) transfers can run in parallel with host code (e.g. start the copy-in as soon as the data is ready), and 3) transfers unrelated to a specific kernel can run in parallel with kernel execution.

We perform such optimisations within ASET. After delimiting the identified algorithmic species by pragma's, ASET is instructed to: 1) mark inputs and outputs as copy-ins and copy-outs for the current kernel, and 2) add synchronisation barriers after the transfers. BONES then generates a second host thread, receiving transfer requests and performing synchronisations. An example of non-optimised output is shown in the form of pseudo code in Listing 5, in which each copy-in and copy-out implies that it is allowed to start the copy at that specific point, and must be finished before the given deadline (see also Fig. 3).

After producing the non-optimised form (e.g. Listing 5), the tool performs different types of optimisations in an iterative way, including: 1) copy-ins directly after copy-outs are removed (e.g. from Listing 5 to Listing 6), 2) copy-ins are moved to the front if the data is not written by the previous species, 3) deadlines of copy-outs are increased if the data is not written by the next species (e.g. from Listing 6 to Listing 7), 4) unused synchronisation barriers are omitted, and 5) transfers are moved to an outer loop if possible. The effects of the transfer optimisations are discussed in Sect. 5.

4.2 Optimisations within Bones

The compiler BONES performs various optimisations to the source code before it instantiates a skeleton. Most optimisations are conditionally applied based on the algorithmic species. For example, array indices and the corresponding array

[3] BONES and ASET can be found at: `http://parse.ele.tue.nl/species/`.
[4] CAST can be found at `http://cast.rubyforge.org/`.

Listing 5. Transfer example (original).

```
 1 copy-in(A,1)
 2 sync(1)
 3 kernel: B ← A
 4 copy-out(B,2)
 5 sync(2)
 6 copy-in(B,3)
 7 sync(3)
 8 kernel: C ← B
 9 copy-out(C,4)
10 sync(4)
```

Listing 6. Transfer example (partly optimised).

```
1 copy-in(A,1)
2 sync(1)
3 kernel: B ← A
4 copy-out(B,2)
5 sync(2)
6 kernel: C ← B
7 copy-out(C,4)
8 sync(4)
```

Listing 7. Transfer example (fully optimised).

```
1 copy-in(A,1)
2 sync(1)
3 kernel: B ← A
4 copy-out(B,4)
5 kernel: C ← B
6 copy-out(C,4)
7 sync(4)
```

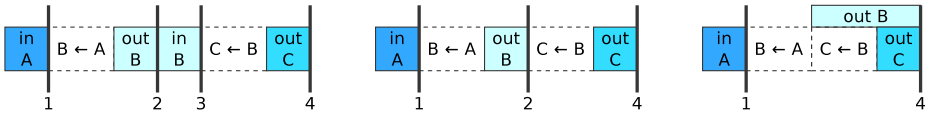

Fig. 3. Illustrating the optimisation steps: original (left, Listing 5), partly optimised (middle, Listing 6), fully optimised (right, Listing 7).

names for input arrays in *neighbourhood*-based skeletons for GPUs are replaced by local indices and local array names. This transformation is a simple matter of name-changing, the actual definition of the local indices and the pre-fetching into local memories is performed within the corresponding skeletons.

Bones also performs several performance-oriented transformations, including caching in the register file and *thread coarsening*. Register file caching replaces array accesses (mapped to off-chip memories) with scalar accesses (mapped to registers) in certain cases. For example, in Listing 1, the accesses to vector $\boldsymbol{r}$ in lines 3 and 5 can be replaced by scalar accesses under the condition that a final store to $\boldsymbol{r}$ is added after line 6. Thread coarsening (or merging) is a technique to increase the workload per thread. This comes at the cost of parallelism, but could increase data re-use through locality or factor out common instructions [14]. In Bones, coarsening is enabled if the species encapsulates re-use over a complete data structure. For example, in the case of '`0:M,0:N|chunk(-,0:N)` → `0:M,0:N|element`', a total of $N \cdot M$ elements are produced, while only M chunks are available as input, resulting in the re-use of the entire input data structure with a factor N. Coarsening is only enabled for kernels without divergent control flow and with sufficient data re-use and parallelism.

To further improve the performance of the generated code, Bones enables *zero-copy* for the OpenCL targets using an aligned memory allocation scheme. In OpenCL, data needs to be explicitly copied from *host* (typically a CPU) to *device* (the accelerator). In some cases, the host and device share the same memory, e.g. for CPU targets or for fused CPU/GPU architectures. In these cases, a memory copy can be saved by performing a pointer-only copy, i.e. a *zero-copy*. Bones enables zero-copying in OpenCL for Intel CPUs by fulfilling two requirements: 1) using specific OpenCL memory map and memory un-map functions, and 2), aligning all memory allocations to 128-byte boundaries. To

ensure aligned memory allocations in the original code, BONES provides a custom `malloc` implementation and furthermore replaces existing stack allocations with aligned dynamic allocations in a matter similar to [12].

Furthermore, BONES flattens data structures to a single dimension when generating OpenCL or CUDA code. Parallel loops are also flattened, decoupling the amount of loops from the thread or work-item structures provided by OpenCL and CUDA. In contrast, many existing approaches (e.g. [1, 7, 22]) map multi-dimensional loops to the multi-dimensional thread or work-item structures provided by OpenCL and CUDA. Although this might be a straightforward solution, it limits the applicability of these approaches to 2 or 3-dimensional loops and data structures. By flattening, BONES omits these limitations and is able to handle any degree of loop nesting and arrays of any dimension.

5 Evaluation and Experimental Results

To evaluate the applicability of algorithmic species and to validate ASET and BONES, we test against the PolyBench/C 3.2 benchmark suite[5], which is a popular choice for evaluating polyhedral-based compilers [1, 22]. We choose this suite because it allows us to compare against polyhedral-based compilers, and allows us to perform automatic extraction of algorithmic species. Nevertheless, BONES can still be used for code that does not fit the polyhedral model, albeit that the classification of code has to be performed manually (see [16] for examples).

The PolyBench suite contains 30 algorithms, in which we identify a total of 110 species[6] with ASET, or 60 if we exclude those found within other species (i.e. nested species). The classified code for these 60 species can now be fed into BONES. However, there is a large number of benchmarks with species in the form of inner-loops with little work, executing in a fraction of a millisecond, making start-up and measurement costs dominant. For our evaluation, we therefore exclude `adi`, `cholesky`, `dynprog`, `durbin`, `fdtd-2d-apml`, `gramschmidt`, `lu`, `ludcmp`, `reg_detect`, `symm`, `trmm` and `trisolv`. The exclusion of these benchmarks can be automated by integrating a basic roofline-like performance model. Additionally, we exclude `floyd-warshall` and `seidel-2d` because they contain no parallelism in their current form. All in all, we include 34 species (not necessarily unique) spread across 16 benchmarks. Within a benchmark, we number the found algorithmic species sequentially[7].

Although GPUs are currently the primary target of BONES, we also include an evaluation of our multi-core CPU targets: we perform a comparison of the 3 CPU targets against single-threaded code. Next, we compare our approach for the CUDA target against two state-of-the-art polyhedral-based compilers: PAR4ALL [1] and PPCG (based on the PLUTO algorithm) [22]. To the best of our knowledge, these are the only available fully-automatic compilers able to

[5] PolyBench website: `www.cse.ohio-state.edu/~pouchet/software/polybench/`.
[6] PolyBench annotated with species: `http://parse.ele.tue.nl/species/`.
[7] See `http://parse.ele.tue.nl/species/` for the corresponding species.

generate CUDA code directly from C (we exclude C-to-CUDA [2], as it is limited to kernel generation only). Finally, we discuss the benefits of our unique combination of a skeleton-based compiler and an algorithm classification.

5.1 OpenCL and OpenMP on a Multi-core CPU

We perform experiments on a 4-core CPU comparing the 3 CPU targets (OpenMP, Intel SDK OpenCL, AMD SDK OpenCL) against single-threaded C code. We use an Intel Core i7-3770 (4 cores, 8 threads) with support for AVX. We use the auto-vectorising GCC 4.6.3 (`-O3`) compiler for the single-threaded and OpenMP targets. Furthermore, we use AMD APP 2.7 and Intel OpenCL 2012 for the OpenCL targets. We disable the CPU's *Intel Turbo Boost* technology. We use the 'large dataset' as defined in PolyBench with single-precision floating point computations. We average over multiple runs and start each run with a warm-up dummy computation followed by a cache flush.

We show the results of the experiments in Fig. 4. We see geometric mean speed-ups of 2.1x (Intel SDK OpenCL), 2.4x (AMD SDK OpenCL), and 2.7x (OpenMP). Although the three targets use the same hardware, we still observe significant performance variation for the individual benchmarks. Differences are among others the lower thread creation cost for OpenMP, the different thread scheduling policies, and different auto-vectorisers. For a detailed comparison of OpenMP against OpenCL in general, we refer to other work, such as [20]. Furthermore, we expect to see improved speed-ups by applying a pre-processing parallelisation pass first (such as PLUTO or [19]), which we leave for future work.

5.2 Comparison of the CUDA Target Against the State-of-the-Art

For our CUDA experiments, we use an NVIDIA GeForce GTX470 GPU (448 CUDA cores) and GCC 4.6.3 (`-O3`) and NVCC 5.0 (`-O3 -arch=sm_20`) for compilation. We test BONES version 1.2 against the latest versions of PAR4ALL (version 1.4.1) and PPCG (pre-release commit 94af357, Feb 2013)[8]. We test BONES with and without host-accelerator transfer optimisations. Furthermore, we include as a reference PolyBench/GPU [9], hand written CUDA code[9]. As before, we use the single-precision 'large dataset' and average over 10 runs, starting with a warm-up.

We perform two tests. For our first test, we evaluate the quality of the generated CUDA kernels only. We generate kernels for each of the found algorithmic species in isolation using BONES, PAR4ALL, and PPCG. As before, within a benchmark, we number the found algorithmic species incrementally, while averaging the execution time of kernels executed multiple times. The results of the kernel quality test can be found in Fig. 5 in terms of speed-up of BONES compared to PAR4ALL and PPCG. We make the following remarks:

[8] We test PPCG with default tiling options. Manual tuning of the tiling parameter for each benchmark is omitted in this work to keep the obtained results fully automatic.

[9] PolyBench/GPU provides non-optimised CUDA code. Unfortunately, fully optimised hand-written code is not available for these benchmarks.

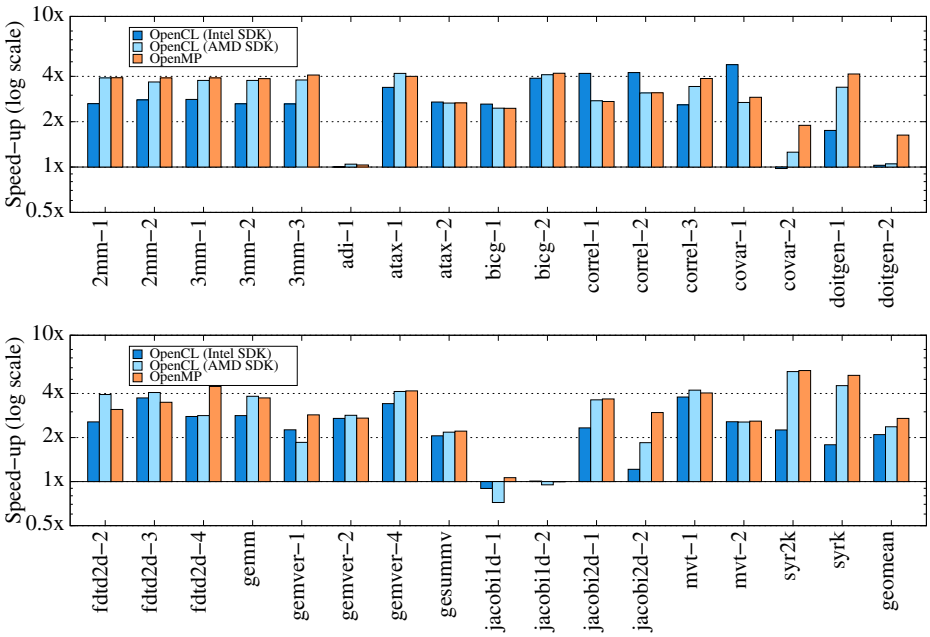

Fig. 4. Speed-ups of the OpenCL (■ for the Intel SDK and ■ for the AMD SDK) and OpenMP (■) targets compared to single-threaded code on a 4-core CPU.

- BONES shows significantly better performance (2x or more) for 21 kernels (compared to PAR4ALL) and 11 kernels (compared to PPCG).
- In a few cases (e.g. `correl-3`,`covar-2`,`jacobi2d-2`) PPCG and/or PAR4ALL are slightly ahead due to better data locality enabled by loop tiling.
- A number of kernels (e.g. `atax-1`, `bicg-2`, `mvt-1`, `syr2k`, `syrk`) use a skeleton in the form of Listing 4 to ensure coalesced memory accesses, yielding a significant speed-up over PAR4ALL and a moderate speed-up over PPCG.
- In the cases of `2mm` and `3mm` (both matrix multiplication), BONES is on-par with PPCG. In these cases, BONES relies on the hardware cache, while PPCG performs loop tiling and explicit caching through the GPU's local memories.
- In several cases (`2mm`, `3mm` and `gemm`), BONES and PPCG both perform *thread coarsening*, gaining performance over PAR4ALL.
- BONES achieves a 1.6x (vs. PAR4ALL) and 1.4x (vs. PPCG) average speed-up.

For the second test, we measure the full benchmark ('scop' in PolyBench terminology). We show the results in Fig. 6 in terms of speed-up compared to optimised BONES code. The (experimental) option to optimise CPU-GPU transfers for PAR4ALL (`--com-optimization`) does not work for these benchmarks. For a fair comparison, PAR4ALL should therefore be compared to BONES without transfer optimisations. We make the following remarks with respect to Fig. 6:

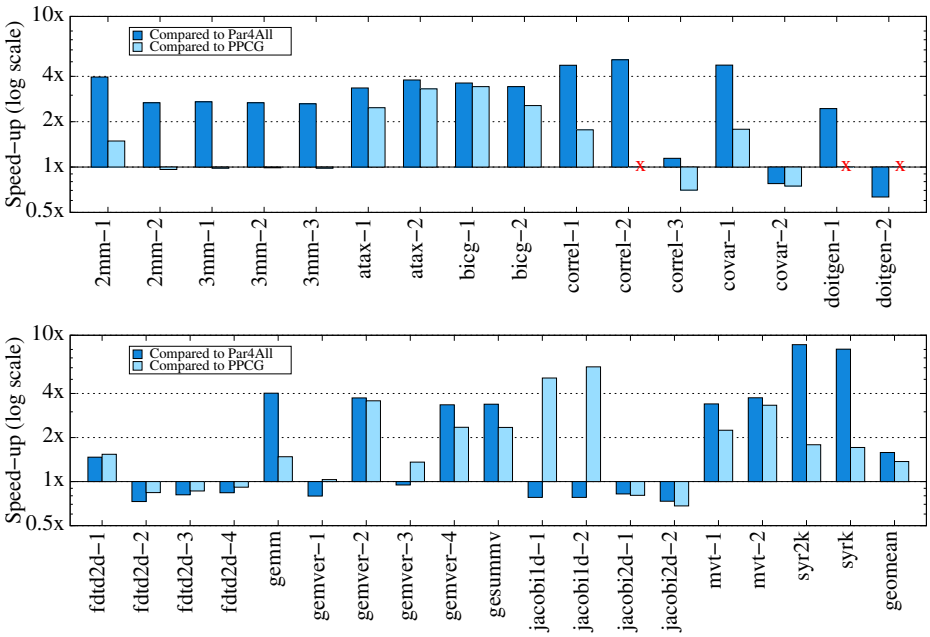

Fig. 5. Speed-up of CUDA kernel code generated by BONES compared to PAR4ALL (■) and PPCG (■) for the PolyBench suite (higher is in favour of BONES). PPCG was unable to generate code for `correl-2`, `doitgen-1`, and `doitgen-2` (marked by an x).

- In almost all cases, BONES with transfer optimisations outperforms the other compilers. On average, BONES achieves a speed-up of 3.0x and 1.2x over PAR4ALL and PPCG respectively (excluding the 100x or worse cases).
- Hand written (non-optimised) PolyBench/GPU code [9] is in many cases significantly slower compared to compiler generated code (on average 4.2x compared to BONES).
- A 1.8x average speed-up is achieved by performing CPU-GPU transfer optimisations with ASET and BONES, although further optimisations are still possible. For example, in the case of `fdtd-2d`, PPCG is able to move additional CPU-GPU transfers to outer loops, resulting in a significant speed-up.

In general, we conclude that the supplied skeletons for BONES already outperform PAR4ALL and PPCG on average for the CUDA kernels found in these benchmarks. For the 34 kernels shown in Fig. 5, BONES uses 5 different skeletons, each used at least twice. Furthermore, we note that BONES, in contrast to polyhedral compilers, can be used for code containing non-affine loop nests.

5.3 Benefits of Skeleton-Species Integration

By integrating algorithmic species with a skeleton-based compiler, we have created a unique approach to automatically generate code for parallel targets. We

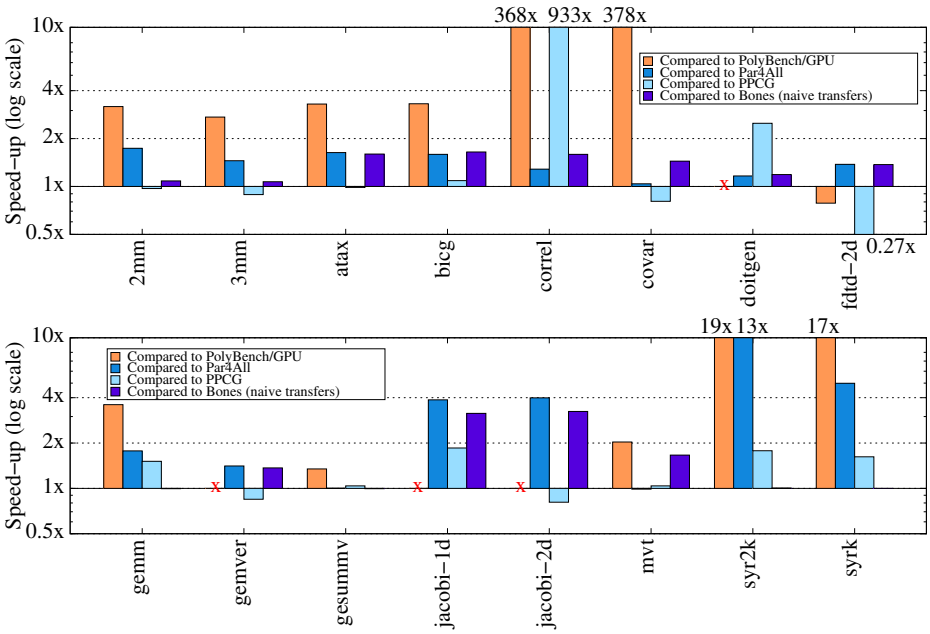

Fig. 6. Speed-up of BONES compared to PolyBench/GPU (■), PAR4ALL (■), PPCG (■), and BONES without transfer optimisations (■) (higher is in favour of BONES). PolyBench/GPU is not available for `doitgen`, `gemver`, `jacobi-1d`, and `jacobi-2d` (X).

see this novel combination as a way to profit from the benefits of skeleton-based compilation, without having its drawbacks.

Skeleton-based compilation has several benefits [4]. Firstly, compilation requires only basic transformations that can be performed at abstract syntax tree level, omitting the need for intermediate representations which often lose code structure and variable naming. This allows the compiler to generate readable code, enabling opportunities for further fine-tuning and manual optimisation. Furthermore, the skeletons themselves can be formatted to include structure and code comments, greatly benefiting readability. Secondly, skeleton-based compilation benefits from the flexibility of improving the compiler or extending to other targets: simply adjust or write the appropriate skeletons. Finally, several optimisations applied within skeletons in BONES cannot be applied as code transformations on the original code. For example, polyhedral compilers could generate the first kernel of the example skeleton in Listing 4 (lines 1-8), but will not be able to generate an additional pre-processing kernel (lines 10-23), as it is not a permutation of the original code.

Because of the integration of algorithmic species, BONES is the first skeleton-based compiler that can be used in a fully-automatic tool-flow. This removes the requirements of existing skeleton-based approaches such as SkePU [7] and SkeCL [21] to manually identify a skeleton and modify the code such that the skeleton can be used. Furthermore, algorithmic species provides a clear, struc-

tured, and formally defined way of using skeletons, which can be beneficial in cases where manual classification is unavoidable.

6 Conclusions and Future Work

In this work we have demonstrated the successful integration of an algorithm classification (algorithmic species) with skeleton-based compilation. This results in a novel method to perform automatic generation of parallel code, transforming sequential C code into readable CUDA, OpenCL or OpenMP code. With ASET, we are able to automatically extract species from affine loop nests. The species are used within the skeleton-based compiler BONES to select a skeleton. BONES is able to produce readable code, provides flexibility for new targets and optimisations, and generates efficient code. Furthermore, the combination of ASET and BONES allows us to perform host-accelerator transfer optimisations.

Many existing skeleton-based compilers [3, 4, 7, 21] have failed to become popular, despite their flexibility and high performance potential. By combining skeletons with an algorithm classification, we overcome their drawbacks: we do not require users to select a skeleton, and we have a fixed and structured classification. Compared to polyhedral compilers on the other hand (e.g. [1, 22]), we have shown significant speed-ups: 1.6x and 1.4x for CUDA kernel code versus PAR4ALL and PPCG respectively. Additionally, BONES generates readable code, leaving the user with further possibilities for optimisation or fine-tuning of the algorithm. Furthermore, BONES is applicable outside the scope of affine loop nests, although species may have to be identified manually.

In this work, we have focused mainly on the CUDA GPU target. As part of future work, we plan to compare our OpenCL targets against the state-of-the-art as well. Furthermore, we plan to extend BONES by investigating the benefits of kernel fusion and fission.

References

[1] M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, J. O. Mcmahon, F.-X. Pasquier, G. Péan, and P. Villalon. Par4All: From Convex Array Regions to Heterogeneous Computing. In *IMPACT 2012 : Second International Workshop on Polyhedral Compilation Techniques*, 2012.

[2] M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In *CC '10: 19th International Conference on Compiler Construction*. Springer, 2010.

[3] W. Caarls, P. Jonker, and H. Corporaal. Algorithmic Skeletons for Stream Programming in Embedded Heterogeneous Parallel Image Processing Applications. In *IPDPS: Int. Parallel and Distributed Processing Symposium*. IEEE, 2006.

[4] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1991.

[5] P. Custers. Algorithmic Species: Classifying Program Code for Parallel Computing. Master's thesis, Eindhoven University of Technology, 2012.

[6] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A Hybrid Multi-core Parallel Programming Environment. In *GPGPU-1: 1st Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[7] J. Enmyren and C. W. Kessler. SkePU: A Multi-backend Skeleton Programming Library for Multi-GPU Systems. In *HLPP '10: 4th International Workshop on High-level Parallel Programming and Applications*. ACM, 2010.

[8] P. Feautrier. Dataflow Analysis of Array and Scalar References. *Springer International Journal of Parallel Programming*, 20:23–53, 1991.

[9] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Autotuning a High-Level Language Targeted to GPU Codes. In *Workshop on Innovative Parallel Computing*, 2012.

[10] S. Guelton, M. Amini, and B. Creusillet. Beyond Do Loops: Data Transfer Generation with Convex Array Regions. In *LCPC '12: Languages and Compilers for Parallel Computing*. Springer, 2012.

[11] T. Han and T. Abdelrahman. hiCUDA: High-Level GPGPU Programming. *IEEE Transactions on Parallel and Distributed Systems*, 22:78 –90, Jan 2011.

[12] T. Jablin, J. Jablin, P. Prabhu, F. Liu, and D. August. Dynamically Managed Data for CPU-GPU Architectures. In *CGO '12: International Symposium on Code Generation and Optimization*. ACM, 2012.

[13] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame. A Script-Based Autotuning Compiler System to Generate High-Performance CUDA Code. *ACM Transactions on Architecture and Code Optimisations*, 9(4):Article 31, Jan. 2013.

[14] Y. Lee, R. Krashinsky, V. Grover, S. W. Keckler, and K. Asanovic. Convergence and Scalarization for Data-Parallel Architectures. In *CGO '13: International Symposium on Code Generation and Optimization*. IEEE, 2013.

[15] C. Nugteren and H. Corporaal. Introducing 'Bones': A Parallelizing Source-to-Source Compiler Based on Algorithmic Skeletons. In *GPGPU-5: 5th Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2012.

[16] C. Nugteren, R. Corvino, and H. Corporaal. Algorithmic Species Revisited: A Program Code Classification Based on Array References. In *MuCoCoS '13: International Workshop on Multi-/Many-core Computing Systems*, 2013.

[17] C. Nugteren, P. Custers, and H. Corporaal. Algorithmic Species: An Algorithm Classification of Affine Loop Nests for Parallel Programming. *ACM TACO: Transactions on Architecture and Code Optimisations*, 9(4):Article 40, 2013.

[18] C. Olschanowsky, A. Snavely, M. Meswani, and L. Carrington. PIR: PMaC's Idiom Recognizer. In *ICPPW '10: 39th International Conference on Parallel Processing Workshops*. IEEE, 2010.

[19] E. Park, L.-N. Pouchet, J. Cavazos, A. Cohen, and P. Sadayappan. Predictive Modeling in a Polyhedral Optimization Space. In *CGO '11: International Symposium on Code Generation and Optimization*. IEEE, 2011.

[20] J. Shen, J. Fang, H. Sips, and A. Varbanescu. Performance Gaps between OpenMP and OpenCL for Multi-core CPUs. In *ICPPW: International Conference on Parallel Processing Workshops*. IEEE, 2012.

[21] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In *IPDPSW '11: International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*. IEEE, 2011.

[22] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Transactions on Architecture and Code Optimisations*, 9(4):Article 54, Jan. 2013.

[23] M. Wolfe. Implementing the PGI Accelerator Model. In *GPGPU-3: 3rd Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2010.