Heterogeneous Computing without Heterogeneous Programming

Miguel Diogo and Clemens Grelck

University of Amsterdam
Institute of Informatics
Science Park 904, 1098XH Amsterdam, Netherlands
diogo@science.uva.nl c.grelck@uva.nl

Abstract. From laptops to supercomputer nodes hardware architectures become increasingly heterogeneous, combining at least multiple general-purpose cores with one or even multiple GPU accelerators. Taking effective advantage of such systems' capabilities becomes increasingly important, but is even more challenging.
SaC is a functional array programming language with support for fully automatic parallelization following a data-parallel approach. Typical SaC programs are good matches for both conventional multi-core processors as well as many-core accelerators. Indeed, SaC supports both architectures in an entirely compiler-directed way, but so far a choice must be made at compile time: either the compiled code utilizes multiple cores and ignores a potentially available accelerator, or it uses a single GPU while ignoring all but one core of the host system.
We present a compilation scheme and corresponding runtime system support that combine both code generation alternatives to harness the computational forces of multiple general-purpose cores and multiple GPU accelerators to collaboratively execute SaC programs without explicit encoding in the programs themselves and thus without going through the hassle of heterogeneous programming.

1 Introduction

Single Assignment C (SaC)[1] is a purely functional array programming language for compute-intensive and performance-sensitive applications. SaC has a syntax that very much resembles C, yet it is a fully-fledged programming language in its own right with context-free substitution of expressions as the driving principle of program execution. SaC features truly multidimensional arrays as the main data aggregation principle as well as shape- and dimension-invariant definitions of array operations. Arrays are truly stateless and, thus, functions (conceptually) consume their array arguments and produce new array values. The focus of the SaC project is on compiler technology that transforms high-level functional specifications into executable code that competes well with C or Fortran through aggressive compiler optimization and fully compiler-directed parallelization[2].

Heterogeneous computing receives increased attention due to the rise of powerful and affordable accelerator hardware, such as general purpose GPUs, the IBM Cell processor, and FPGAs [3, 4]. These accelerators can provide speedups of one or two orders of magnitude depending on the problem [5, 6, 7]. Most available heterogeneous programming environments [8, 9, 10] provide a way to offload computations to one of many types of accelerators. These approaches mainly solve portability issues, but leave programmers alone with low-level, architecture-specific coding of applications, and thus very much limit productivity the more heterogeneous the target architectures become.

With all the above approaches, a selected accelerator does the heavy work while the host CPU is typically idle, as well as any other available accelerators. Making effective use of heterogeneous computing environments may well provide additional performance gains, as shown by the MAGMA project [11] in the area of algebraic computations. Further studies demonstrate performance gains by sharing computations between multiple GPUs and CPU cores [11, 12, 13].

Our approach taken for SaC is more ambitious than the ones above: from the very same declarative and architecture-agnostic program we fully automatically generate efficient parallel code for multi-core CPUs [14] and many-core GPUs [15]. However, for now a choice needs to be made at compile time to either utilize one or multiple multi-core CPUs or a single CUDA-enabled GPU.

In this paper we propose the necessary compilation and runtime system technology to realize additional performance gains by simultaneously using one or more multicore CPUs and one or more GPUs to cooperatively execute array operations without sacrificing our declarative, compiler-directed approach. This requires the integration of the currently separate multicore and CUDA compilation paths within the SaC compiler. Various issues need to be addressed from high-level program transformations over code generation to dealing with explicit memory transfers between the various memories involved.

The CUDA backend of the SaC compiler generates CUDA-kernels from SaC array operations. Only such kernels can run on CUDA-enabled devices. They are subject to several constraints imposed by the underlying hardware. For example, the absence of a stack on GPUs rules out function calls, and branches are costly as they result in parts of the GPU being masked out during execution. Consequently, CUDA kernels must follow relatively simple patterns of nested loops, and complex array operations must often be split into several consecutive kernels. In contrast, the SaC multicore backed puts a lot of effort into improving data locality in cache-based systems [20], which often results in fairly complex codes that would be impossible or at least inefficient to run on GPUs. This example shows that the choice of target architecture does not only affect the final compilation stages of target code generation, but already needs to be taken into account in central parts of the compilation process.

GPUs have their own memory, which makes heterogeneous computing systems also distributed memory systems. Explicit data transfers between the host memory and the various GPU device memories must be introduced into the generated code. For good performance it is crucial to avoid redundant data transfers.

The SaC CUDA backend schedules entire array computations on the (single) GPU. With this trivial static scheduling, the compiler can make all data movement decisions statically, and only entire arrays are transferred between host and device memories. If multiple GPUs and host cores are to collaborate in a single array operation, we must dispense with this static model. Instead (conceptual) arrays must be distributed across the various memories of a heterogeneous computing system. Consequently, as such an array becomes the argument of a subsequent array operation, parts of this array are likely not to be present in the memory of the host or device that needs the data. Heterogeneity naturally limits the applicability of static scheduling in such a context, and we expect dynamic approaches to be more effective in the long run [16, 17, 13]. Therefore, the runtime system needs to keep track of where which parts of an array are stored and initiate the necessary data transfers to make the data available where it is needed. In light of these problems, we specifically aim to:

- track location of computed array slices dynamically;
- perform as few copies from/to device memory as possible;
- avoid redundant copying between different memories;
- generate two code versions for each array operation, one optimized for CUDA and the other for multicore execution.

The remainder of the paper is organized as follows. In Section 2 we provide some background information on SaC and in Section 3 on relevant existing work in the SaC compiler. Section 4 describes how to concert the CUDA and multicore compilation paths. In Section 5 we introduce the notion of distributed arrays. Section 6 shows how this proposed scheme integrates with existing SaC dynamic scheduling facilities. In Section 7 we present and analyze preliminary experimental results. Eventually, we discuss some related work in Section 8 and draw conclusions in Section 9.

2 Single Assignment C

As the name suggests, SaC leaves the beaten track of functional languages and adopts a C-like syntax to ease adoption by imperative programmers. Core SaC is a functional, side-effect free variant of C: we interpret assignment sequences as nested let-expressions, branches as conditional expressions and loops as syntactic sugar for tail-end recursive functions [1, 18]. Despite the radically different underlying execution model (context-free substitution of expressions vs. step-wise manipulation of global state), all language constructs show exactly the operational behaviour expected by imperative programmers. This allows programmers to choose their favourite interpretation while the compiler exploits the side-effect free semantics for advanced optimization and automatic parallelization.

On top of this language kernel SaC provides genuine support for processing truly multidimensional and truly stateless/functional arrays advocating a rank- and shape-generic programming style. Array types include arrays of fixed shape, e.g. `int[3,7]` for a 3x7-matrix of integers, arrays of fixed rank, e.g. `float[.,.]`

for a float matrix of any size and arrays of any rank, e.g. `double[*]` for a double array of any rank, which could be a vector, a matrix, etc. SaC only provides a small set of built-in array operations, essentially to retrieve the rank (`dim(`*array*`)`) or shape (`shape(`*array*`)`) of an array and to select elements or subarrays (*array*`[`*idxvec*`]`). All aggregate array operations are specified using with-loop expressions, a SaC-specific array comprehension:

```
with {
  ( lower_bound <= idxvec < upper_bound) : expr;
   ...
  ( lower_bound <= idxvec < upper_bound) : expr;
}: genarray( shape, default)
```

The key word `genarray` makes this with-loop define an array whose shape is given by the ***`shape`*** expression and whose elements default to the value of the ***`default`*** expression. The body consists of multiple (disjoint) *partitions*. Here, *lower_bound* and *upper_bound* denote expressions that must evaluate to integer vectors of equal length. They define a rectangular (generally multidimensional) index set. The identifier *idxvec* represents elements of this set, similar to induction variables in for-loops. We call the specification of such an index set a *generator* and associate it with some potentially complex SaC expression. Thus, we define a mapping between index vectors and values, in other words an array. We deliberately use index *sets*, which make any with-loop expose fine-grained concurrency and thus the ideal basis for our parallelization efforts.

```
...
foo = with {
        (. <= iv <= .): 20;
      }: genarray([1000, 1000]);

y = foo[[1,20]] + 1;

bar = with {
        ([0,10] <= iv <= .): foo[iv] + y;
      }: modarray(foo);

bar[[1,2]] = 10;
...
```

Fig. 1: Example SaC code, consisting of a `genarray`, a `modarray`, and a few array operations in between.

We illustrate SaC code in general and the with-loop in particular by the (artificial) code fragment of Fig. 1. This code creates a `1000x1000` matrix where each element is set to `20` using the `genarray` with-loop. The dots in the lower and upper bound expression positions are syntactical sugar for the least and the greatest legal index vector of the defined array. So, the generator covers the entire index space, thus making an explicit default expression obsolete. We then

access this new array at some index and name the result `y`. Next, we define a new matrix `bar` based on the existing matrix `foo` using a `modarray` with-loop, a variant of the `genarray`-with-loop introduced above. Matrix `bar` has the same rank and shape as matrix `foo`. Likewise the values of the leftmost 10 columns are taken from the corresponding values of `foo`. The remaining columns covered by the generator are also taken from `foo`, but incremented by the value of `y`. Finally, we change element `[1,2]` of the new matrix to `10`. More precisely, we create a third matrix also named `bar` that is identical to the previously existing matrix `bar` except for the value at row 1, column 2.

3 Existing Work in the SAC Compiler

3.1 Generating Multi-threaded Code

The SAC-compiler takes considerable effort to condense multiple (often computationally light-weight) with-loops into fewer, more heavy-weight ones by means of high-level code transformations [19]. This typically results in complex multi-generator with-loops. Subsequently, such with-loops undergo a transformation into what is called the *canonical order* [20]. This canonical order is an internal representation of with-loops as a single, fairly complex (imperfect) nesting of loops, whose execution follows the way elements are stored in memory, rather than as a sequence of simple, perfect loop nests that would arise from the individual compilation of generators. This technique considerably improves performance on conventional architectures through increased cache utilization.

The multicore code generator then selects individual with-loops for multi-threaded execution based on a simple cost model and creates parallel sections with the necessary communication and synchronization facilities. At runtime, the main thread executes any sequential code as normal, while worker threads remain inactive. Before starting one of the parallel sections and waking up the worker threads, the main thread sets up a memory area, known as task frame, where it inserts any variables the worker threads require. On a shared memory system arrays are not copied to the task frame, only their references. For with-loops in parallel sections, a scheduler assigns disjoint regions of the array index space to each thread for computing. Threads repeat the with-loop computation with different regions until the whole iteration space is assigned by the scheduler. The default strategy is block scheduling, but programmers can assign other static and dynamic ones to each with-loop.

3.2 Generating CUDA kernels

The first step towards generating CUDA kernels lies in selecting those with-loops that can actually run on the GPU, e.g. it is impossible to call functions within kernels. With-loops with function calls in the body (after optimizations like inlining) thus take the standard compilation route, while selected with-loops go through CUDA-specific transformations.

The SaC CUDA backend [21, 15] maps individual with-loops to CUDA kernels. The canonical order, however, is not suitable for CUDA as the corresponding loop structure becomes too complicated and the GPUs can much less benefit from this form of data locality. Hence, each generator is individually compiled into one kernel instead.

In order to statically distinguish between host-allocated and device-allocated arrays in a sound way, the CUDA backend employs a simple type system where every array type is tagged as either *host type* or *device type*. Relatively free array variables inside CUDA with-loops are converted from a host type to the corresponding device type by means of the primitive function `host2device`. Conversely, arrays computed by a CUDA with-loop (thus having a device type) can be converted to a host type by the corresponding primitive function `device2host`. At runtime these type conversion functions take care of the necessary memory transfers. It is noteworthy that the SaC CUDA backend takes considerable effort to avoid unnecessary memory transfers through high-level code transformations.

Consider the code from the example in Fig. 1. As both with-loops are CUDA-compatible, the code is transformed to look like in Fig. 2. Note that to compute `bar_cuda` there is no need to transfer `foo` back to device memory as `foo_cuda` already resides there and as a functional data structure remains unchanged.

```
...
foo_cuda = with_cuda {...}: genarray([1000, 1000]);

foo_host = device2host(foo_cuda);
y = foo_host[[1,20]] + 1;

bar_cuda = with_cuda {...}:modarray(foo_cuda);
...
```

Fig. 2: Excerpt from the code of Fig. 1, both with-loops selected for CUDA

4 Compiling With-Loops for Multiple Targets

The same with-loop will produce very different code whether it is compiled for CUDA or for multicore execution. To solve this incompatibility, we generate both versions side-by-side. The first step is to allow existing compiler procedures to mark all with-loops capable of running on CUDA. Then, these with-loops are duplicated, and the copy is deselected for CUDA execution. This way, one of the with-loops can go through the CUDA-specific transformations while the other goes through the multicore-specific transformations and among others is transformed into canonical order.

The resulting code for our running example is shown in Fig. 3. To simultaneously represent the two with-loop variants, these are encapsulated in a conditional. For illustration we use the keywords `with_cuda` and `with_host`. The

conditional will later be moved to parallel sections executed by multiple threads in SPMD-style, and the predicate will be evaluated by each thread individually. One host thread per GPU will launch the CUDA kernels. The remaining worker threads will run multithread code on the host directly.

```
...
y = foo_host[[1,20]] + 1;

if(cudaBranch) {
  bar_cuda = with_cuda {...}:modarray(foo_cuda);
} else {
  bar_host = with_host {...}:modarray(foo_host);
}

bar_host[[1,2]] = 10;
...
```

Fig. 3: Excerpt from the SaC code in Fig. 1 after with-loop duplication

5 Managing Multiple Memories

The existing CUDA backend only transfers arrays in their entirety between host and (one) device memory. This is not viable if we are to execute with-loops simultaneously on multiple devices or on device and host. To keep track of distributed arrays scattered across multiple memories, we introduce a third type: distributed variables. To access the array, it first needs to be converted from distributed type to a concrete type through explicit conversion. Whenever these arrays are written to, however, they again must be converted to the distributed type to make the runtime system aware of any changes. To represent these conversions we introduce dedicated primitive functions: `dist2host` and `host2dist` for the host, `dist2device` and `device2dist` for CUDA devices.

5.1 Distributed Variables

To keep track of distributed arrays scattered over different memories we introduce a control structure inside distributed variables. This structure has references to the concrete array addresses on the host and on the GPUs as well as a table recording which parts of an array are present where. Currently, we restrict ourselves to dividing arrays into blocks along the outermost axis.

To avoid unnecessary copying, the dynamic tracking scheme should also be able to detect cases where data is available in several memories simultaneously. The proposed scheme addresses this using techniques derived from cache coherency protocols [22]. The idea is to keep a table in host memory tracking the state of each block on each device. The possible states are (M)odified, (S)hared and (I)nvalid, mimicking the simple MSI cache coherence protocol [23], as illustrated in Fig. 4.

This proposed control structure allows for some scaling, as it is possible to track more devices simply by adding another row to the control structure table. Each block currently corresponds to a position along the outermost dimension. For a matrix, for example, each block would correspond to a row. We find this to be reasonable in most situations, and it simplifies implementation.

	Blocks ⇒				
Host	S	M	I	I	...
CUDA 1	S	I	M	I	...
CUDA 2	S	I	I	M	...
...					

Fig. 4: Dynamic tracking control structure.

5.2 Dedicated Conversion Functions

The primitive functions to convert from a distributed to a concrete type or vice-versa make use of the control structure in distributed variables to copy only the required parts to the memory system that needs them. These functions can be thought of as extensions to the `device2host` and `host2device` primitive functions introduced by the SaC CUDA backend. Instead of explicitly copying the whole array from one memory to another, the new primitive functions transparently perform the copying of only those data blocks needed but not present in some memory. This is possible because the compiler always knows which parts of an array will be accessed, even if just in a symbolic manner. We can thus pass a range of elements as an argument to the conversion functions.

```
int_dist[*] host2dist(int_dist[*] dist_array, int[*] host_array, int[*] range)
{
  foreach (block i in range) {
    mark_block(i, host, Modified)
    mark_block(i, cuda, Invalid)
  }
  return dist_array;
}

int[*] dist2host(int_dist[*] dist_array, int[*] host_array, int[*] range){
  foreach (block i in range) {
    if( get_block_mark(i, host) == Invalid) {
      copy_block(i, cuda, host)
      mark_block(i, cuda, Shared)
      mark_block(i, host, Shared)
    }
  }
  return host_array;
}
```

Fig. 5: Pseudo-code for the `host2dist` and `dist2host` primitive functions

Pseudo code implementations of `host2dist` and `dist2host` are shown in Fig. 5. For each block that falls into the specified range, `host2dist` sets the corresponding entry in the distributed variable table to *Modified* on the host and to *Invalid* on the CUDA devices. For each block that falls into the specified range, `dist2host` checks the corresponding entry in the distributed variable

table for whether or not the host line is set to *Invalid*. If not, it is done with the block; otherwise, the data is copied from a CUDA device, and both host and device entries are set to *Shared*. The CUDA device functions `device2dist` and `dist2device` are very similar. Note that the distribution control data structures always resides in host memory.

5.3 Making Use of the Type Extensions

Continuing the example of Fig. 3, a new code transformation creates the distributed variables and inserts the required conversion functions. This transformation makes the code look like in Fig. 6. With-loops now operate only on a certain range, as the conversion functions need this information. This range information will eventually come from the scheduler, but for now we use a placeholder function. Note that unlike in the CUDA-only example of Fig. 2, it is now necessary to explicitly convert `bar_dist` to `bar_host`. Also note that distributed variables have to be initialized before the concrete arrays, so that we can update the control structure afterwards.

```
...
y = foo_host[[1,20]] + 1;

bar_dist = initialize_dist_var(...);
range = rangeOracle();
if (cudaBranch) {
  foo_cuda = dist2device(foo_dist, foo_cuda, range);
  bar_cuda = with_cuda { ...  | range}:modarray(foo_cuda);
  bar_dist = device2dist(bar_dist, bar_cuda, range);
} else {
... /* same as above, using host code */
}

bar_host = dist2host(bar_dist, bar_host, [1,2]);
bar_host[[1,2]] = 10;
bar_dist = host2dist(bar_dist, bar_host, [1,2]);
...
```

Fig. 6: Excerpt of the SaC code from Fig. 3 after insertion of distributed variables and primitive conversion functions

6 Code Generation

During code generation both with-loop versions and the corresponding type conversions must be integrated with the multicore parallel sections and scheduler. Our approach is to create a parallel section around the conditionals with both with-loop code variants, rather than single with-loops. The predicate becomes a check for a specific kind of threads, which either launch CUDA kernel or execute sections of a with-loop directly on the host.

Continuing the example from Fig. 6, the above transformation yields the code in Fig. 7. For brevity, we again only consider the `bar` with-loop; effects on the `foo` with-loop are very similar. First, in the master thread, we start a parallel section, so that the worker threads wake up and run the worker function. We pass the function and its arguments through the task frame to the worker threads before the master thread joins parallel execution as worker #0.

```
...
  y = foo_host[[1,20]] + 1;

  bar_dist = initialize_dist_var(...);
  _start_parallel_section(&_spmd_bar, bar_dist, foo_dist, foo_cuda, foo_host);
  _spmd_bar(0, bar_dist, foo_dist, foo_cuda, foo_host);

  bar = dist2host(bar_dist, bar_host, [1,2]);
...

void _spmd_bar( thread_id, bar_dist, foo_dist, foo_cuda, foo_host)
{
  if( _isCUDAthread( thread_id) ) {
    do {
      range, continue = _getSubset( thread_id, ...);

      foo_cuda = dist2device(foo_dist, foo_cuda, range);
      bar_cuda = with_cuda {... | range}:modarray(foo_cuda);
      bar_dist = device2dist(bar_dist, bar_cuda, range);
    } while (continue);
  } else {
    do {
      range, continue = _getSubset( thread_id, ...);

      foo_host = dist2device(foo_dist, foo_host, range);
      bar_host = with_host {... | range}:modarray(foo_host);
      bar_dist = device2dist(bar_dist, bar_host, range);
    } while (continue);
  }

 _sync_barrier( thread_id)
}
```

Fig. 7: The `bar` with-loop of Fig. 6, after creating worker thread functions: original with-loop context (top) and lifted SPMD-style worker function (bottom)

In the parallel section we have the conditional with the two with-loop variants. Both branches are very similar, the only differences are that one uses the CUDA conversion functions and the CUDA with-loop code while the other uses the host conversion functions and the multicore with-loop code. The predicate `_isCUDAthread` uses the thread id to choose the correct branch for each thread.

All threads then enter a loop. Each thread obtains some subset of the elements to compute from the scheduler (represented here as the `_getSubset()` function). The next step is converting the with-loop dependencies from a distributed type to the appropriate concrete type. We do not convert the whole array though, only the parts we need for computing the specific iteration space the scheduler assigned to this thread. The conversion functions will take care of

copying any required missing data from the assigned area. Then, we can perform the computation on this partial concrete array, generating a partial concrete result. The last step in the loop is to update the distributed variable. As long as there are sections of the array to compute, the scheduler sets the output flag `continue` to true, so threads will continue to query the scheduler for a new subset of the array to compute. Once the whole array has been assigned, threads will start to exit the loop and hit a synchronization barrier (`_sync_barrier()`). When all threads are done, the program resumes sequential execution.

7 Preliminary Results

7.1 Benchmark Code

To test the viability of the system described in the previous section, we decided to implement it manually based on a relatively simple but nonetheless representative program. We chose a prototypic stencil computation where over a number of iterations each element in a matrix is set to the arithmetic mean of its four direct neighbours in the previous iteration. There are many ways to write this code in SaC (for examples see [18]), but in one way or another high-level program transformations lead to code similar to the program fragment in Fig. 8. Note that even though the computation itself is rather simple, communication and synchronization are inevitable between iterations of the for-loop, which makes this a clearly non-trivial scenario.

```
for (k=0; k<LOOP; k++) {
  A = with {
      ( . < x < . ) : 0.25f*( A[x+[1,0]] +A[x-[1,0]] +A[x+[0,1]] +A[x-[0,1]]);
    } : modarray( A );
}
```

Fig. 8: Computational kernel of 2-d convolution example

We compiled this program with both the multicore and the CUDA backend to obtain original intermediate C codes. We then manually combined these codes implementing the concepts presented so far. Instead of fully dynamic scheduling, however, we divided arrays statically into two parts: one part is divided equally among the CPU cores, the other among the GPUs. Nonetheless, locations of array slices are indeed tracked "dynamically", following the scheme presented in Section 5. The distributed memory control structure is consulted on every iteration, even though the mapping remains the same throughout execution.

7.2 Experiment 1: Host plus 2 GPUs

We use the Distributed ASCI Supercomputer (DAS-4) [24] for evaluation. Our first experiment runs on a dual hexa-core 2.67 GHz Intel Xeon node equipped

with two NVidia GTX480 GPUs. We run our benchmark for 2000 iterations on a double precision floating point matrix of 9000x9000 elements with different SaC backends: sequential, multicore, CUDA and our proposed hybrid scheme. We use `gcc` with optimization level `-O3` as binary code generator and report the shortest time out of 3 runs. We experiment with different work division schemes and compare the performance we achieve using the hybrid scheme with that of the existing CUDA and multicore backends. Results are shown in Fig. 9.

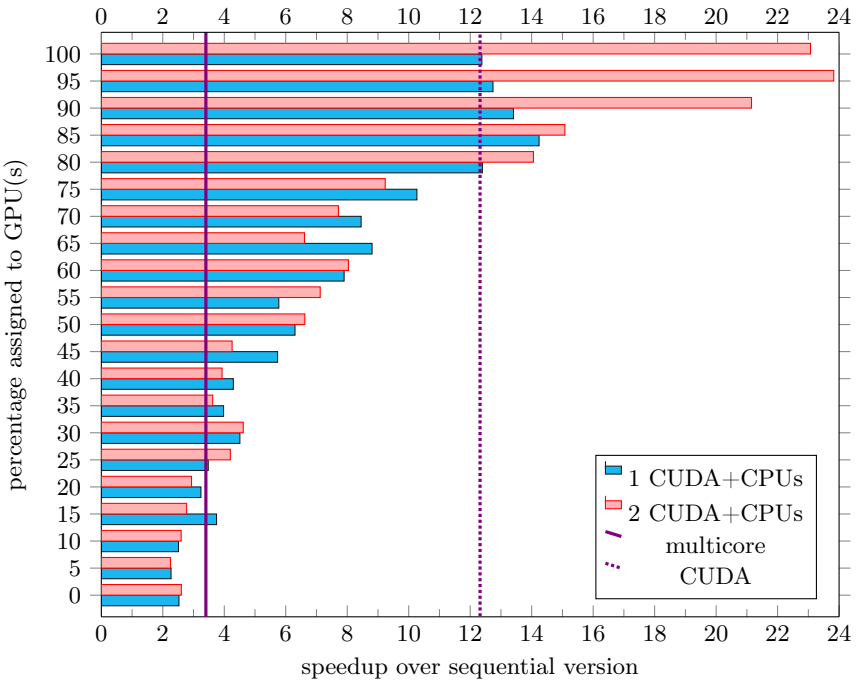

Fig. 9: Comparison of the runtimes between different work division percentages; results for the CUDA and multicore backends are shown as vertical lines

In Fig. 9, we see that while the CUDA backend provides a reasonable speedup of about 12, the multicore backend performs very poorly, with a speedup of just over 3 for 12 threads. The bottleneck is most likely the host memory system. There is so little computing that much CPU time is spent waiting for memory load instructions to complete. In previous experiments we did achieve almost linear speedups for the same kind of code on smaller matrices.

Comparing the performance of our implementation with the others, we see that using the GPU(s) with less than 25% of the iteration space introduces so much overhead that it is better to just use the CPUs. To actually outperform the CUDA-only implementation, one has to use the GPU(s) for at least 80% of the

workload. Assigning the whole array to the CPUs with our implementation is slower than the existing multicore backend. The other way around, assigning the whole array to one GPU, achieves practically the same run time as the existing CUDA backend. We believe this is due to the fact that the CUDA-bound threads can perform the bookkeeping while its assigned GPU is computing the result, thus overlapping overhead with computing. In contrast, CPU-bound threads must do both computation and bookkeeping on the CPU.

With one GPU, our solution achieves the best results with 15% of the workload assigned to the host and 85% assigned to the one GPU. The added benefit from using the CPUs is small, but still significant. Using only the two GPUs, we practically double the performance compared with the existing CUDA backend. The best results are achieved with 95% of the workload assigned to the GPUs and 5% assigned to the CPUs. This, however, results in a marginal benefit only. Although these results show that it is not worthwhile to use the CPUs when more than one GPU is available for this particular problem, other problems, containing sections that cannot be parallelized for the GPU, for example, can still benefit from being able to use all the available CPU cores, rather than running sequentially as is happening today.

7.3 Experiment 2: Up to 8 GPUs

To explore the scalability of simultaneously using multiple GPUs we run experiments on another node of the DAS-4 system equipped with two quad-core 2.4 GHz Intel Xeon CPUs and 8 NVidia GTX580 GPUs. In this experiment we increase the problem size to 10000 iterations and 13000x13000 elements. Given the results of the first experiment we leave out the host and focus on increasing numbers of GPUs instead.

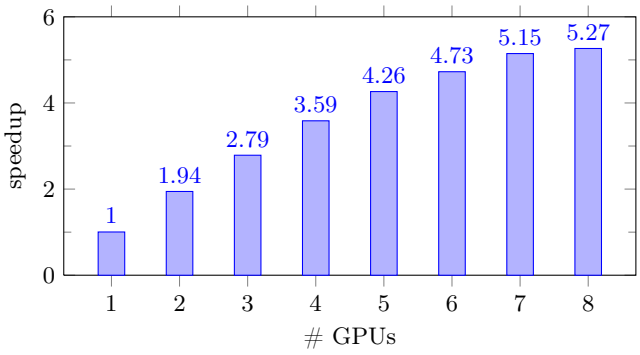

Fig. 10: Speedups over the CUDA backend obtained with our hybrid approach for a varying number of GPUs, without CPU assistance

Considering the results shown in Fig. 10, we see that while using two GPUs provides a nearly two-fold speedup, using 3 or more GPUs provides significantly less improvements. This can be attributed to an increase in communication while the problem size remains constant. As the iteration space is block-wise divided between the different GPUs, two GPUs have only one communication partner while any additional GPU has two. We try to minimize the impact of communication by not communicating redundant data. We believe it does a good job, as the speedup obtained with 8 GPUs over the single GPU version is on par with that obtained with the multicore version over the sequential one.

8 Related Work

We are not aware of other functional approaches to programming heterogeneous systems such as the one we presented in this paper. Recent projects such as Accelerate [25] for Haskell or Microsoft's Accelerator for F# [26] facilitate using one GPU, but lack support for using the CPU simultaneously or for multiple GPUs. This work also differs from our's in general spirit: while Accelerate offers a form of high-level CUDA programming with skeletons in Haskell, with SaC we strive for a completely compiler-directed solution.

On the imperative programming side, a number of heterogeneous computing environments have recently been developed. The approach has mostly been to define and implement APIs. One such heterogeneous environment is Qilin[27]. Qilin is built on top of the Intel Threading Building Blocks (TBB) and the Nvidia CUDA libraries to support both multicore CPUs and Nvidia GPUs. Compilation is done dynamically, Qilin generates the machine code at runtime using static scheduling of operations between the CPU and GPU. This static scheduling however is adaptive. The Qilin system keeps track of operation runtimes in a database, and uses this information to generate a better static schedule on each program run. Qilin was shown to adapt well to varying input sizes, but the amount of offline training required is unclear.

StarPU[22, 28] defines *codelets* as non-preemptible offloadable tasks. Programmers must provide the corresponding kernel functions themselves though. They also have to specify the data dependencies of the *codelets*, which then allows the StarPU runtime system to efficiently schedule tasks to each of the available computation resources. The runtime system also provides a data management facility. Data is divided into disjoint subsets using *filters*, which can be applied recursively. Memory consistency uses a write-back model and a directory-based protocol so that each piece of data has a state associated with it: modified, shared, or invalid. Additionally, it is possible to define custom schedulers and *codelet* drivers for different architectures. With our work in SaC, we intend to introduce a similar system, yet simpler. One significant difference is that we intend to make existing SaC code readily able to make use of heterogeneous computing environments. Specifically, we do not want the programmer to specify codelets or data dependencies, rather should the SaC-compiler infer them automatically.

Song et al. [29] developed a new methodology for matrix computations using multi-core CPUs and multiple GPUs. In this work, a new tiling algorithm is introduced as well as corresponding partition and load balancing schemes. From a SaC perspective, the tiling schemes of Song et al. are rather specific, and out of scope of current research. The work on minimizing communication would be interesting to apply on a SaC scheduler, however.

Ravi et al. [13] describe a heterogeneous computing system for map-reduce applications. Programmers only need to annotate the reduction processing structure, the compiler then generates code for multi-core CPUs and GPUs. At runtime, the work is distributed dynamically. Support for reduction operations is an integral part of SaC. Some results from their scheduling scheme may prove useful for further work in a SaC scheduler.

9 Conclusion

Nowadays, a typical computer contains at least one multicore CPU and one GPU. Together these resources provide significant computing power, but their heterogeneity requires a heroic programming effort to effectively harness it. In this paper we presented compiler and runtime systems extensions for SaC to effectively exploit heterogeneous hardware resources by concerting the existing separate compilation paths for multicore CPUs and single GPUs and adding support for arrays dynamically distributed across multiple memories. Thus, we create support for systems with multiple multicore CPUs and multiple GPUs without requiring programmers to explicitly code for such systems. Preliminary experiments based on a hand-coded prototype show encouraging results.

We are currently implementing the proposed techniques in the SaC-compiler. Once completed, we will investigate a larger range of existing SaC benchmarks and applications. Future work beyond this will be in two directions. Firstly, we aim at completing fully dynamic scheduling between CPU and GPUs. Secondly, we would like to generalize the proposed techniques from host/device memories to network-connected cluster nodes, potentially each equipped with GPU accelerators.

References

[1] Grelck, C., et al: SAC: a functional array language for efficient multi-threaded execution. International Journal of Parallel Programming **34**(4) (2006) 383–427

[2] Wieser, V., et al: Combining High Productivity and High Performance in Image Processing Using Single Assignment C on Multi-core CPUs and Many-core GPUs. Journal of Electronic Imaging **21**(2) (2012)

[3] Chamberlain, R., et al: Visions for application development on hybrid computing systems. Parallel Computing **34**(4) (2008) 201–216

[4] Kumar, R., et al: Heterogeneous chip multiprocessors. Computer **38**(11) (2005)

[5] Guo, Z., et al: A quantitative analysis of the speedup factors of FPGAs over processors. In: Field Programmable Gate Arrays, 2004. Monterrey, CA, USA

[6] Che, S., et al: A performance study of general-purpose applications on graphics processors using CUDA. Journal of Parallel and Distributed Computing **68**(10) (2008) 1370–1380
[7] Williams, S., et al: The potential of the cell processor for scientific computing. In: Computing Frontiers, 2006. 3rd Conference On, Ischia, Italy, ACM (2006)
[8] RapidMind Inc.: Writing Applications for the GPU Using the RapidMind™ Development Platform. (2006)
[9] Papakipos, M.: The PeakStream platform: High-Productivity software development for multi-core processors. Technical report, PeakStream Inc. (2007)
[10] Dolbeau, R., et al: HMPP™: A hybrid multi-core parallel programming environment. In: General Purpose Processing on Graphics Processing Units, 2007. Boston, MA, USA
[11] Tomov, S., et al: MAGMA Users' Guide. University of Tennessee. (2010)
[12] Horton, M., et al: A Class of Hybrid LAPACK Algorithms for Multicore and GPU Architectures. In: Application Accelerators in High-Performance Computing. Knoxville, TN, USA (2011)
[13] Ravi, V., et al: Compiler and runtime support for enabling reduction computations on heterogeneous systems. Concurrency and Computation: Practice and Experience **24**(5) (2011) 463–480
[14] Grelck, C.: Shared memory multiprocessor support for functional array processing in SAC. Journal of Functional Programming **15**(3) (2005) 353–401
[15] Guo, J., et al: Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In: Declarative Aspects of Multicore Programming, 2011. Austin, TX, USA
[16] Hummel, S., et al: Load-sharing in heterogeneous systems via weighted factoring. In: Parallel Algorithms and Architectures, 1996. Padua, Italy, ACM 318–328
[17] Boyer, M., et al: Automatic Intra-Application Load Balancing for Heterogeneous Systems. In: AMD Fusion® Developer Summit 2011, Bellevue, Washington, USA
[18] Grelck, C.: Single Assignment C (SAC): High Productivity meets High Performance. In Horvath, Z., Zsok, V., eds.: 4th Central European Functional Programming Summer School, Budapest, Hungary, LNCS **7241** (2011)
[19] Grelck, C., Scholz, S.B.: Merging compositions of array skeletons in SAC. Journal of Parallel Computing **32**(7+8) (2006) 507–522
[20] Grelck, C., et al: On code generation for multi-generator with-loops in SAC. IFL-99, LNCS **1868** (2000) 77–94
[21] Guo, J.: Compilation of SAC to CUDA. PhD thesis, University of Hertfordshire, Hatfield, UK (2012)
[22] Augonnet, C., et al: A unified runtime system for heterogeneous multi-core architectures. In: Euro-Par 2008. Las Palmas, Spain
[23] Papamarcos, M., et al: A low-overhead coherence solution for multiprocessors with private cache memories. Computer Architecture News **12**(3) (1984) 348–354
[24] DAS-4: Distributed ASCI Supercomputer 4. http://www.cs.vu.nl/das4/
[25] Chakravarty, M., et al: Accelerating Haskell array codes with multicore GPUs. In: Declarative Aspects of Multicore Programming, 2011. Austin, TX, USA (2011)
[26] Microsoft Research: An Introduction to Microsoft Accelerator v2. (July 2012)
[27] Luk, C.K., et al: Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: Microarchitecture, 2009. New York, NY, USA
[28] Augonnet, C., et al: StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In: Euro-Par 2009. Delft, Netherlands
[29] Song, F., et al: Efficient Support for Matrix Computations on Heterogeneous Multi-core and Multi-GPU Architectures. University of Tennessee (2011)