Breaking the GPU Programming Barrier with the Auto-Parallelising SAC Compiler

Jing Guo
University of Hertfordshire,Hatfield, UK
j.guo@herts.ac.uk

Jeyarajan Thiyagalingam
Oxford e-Research Centre, University of Oxford, Oxford, UK
jeyarajan.thiyagalingam@oerc.ox.ac.uk

Sven-Bodo Scholz
University of Hertfordshire,Hatfield, UK
s.scholz@herts.ac.uk

Abstract

Over recent years, the use of Graphics Processing Units (GPUs) for general-purpose computing has become increasingly popular. The main reasons for this development are the attractive performance/price and performance/power ratios of these architectures.

However, substantial performance gains from GPUs come at a price: they require extensive programming expertise and, typically, a substantial re-coding effort. Although the programming experience has been significantly improved by existing frameworks like CUDA and OpenCL, it is still a challenge to effectively utilise these devices. Directive-based approaches such as *hi*CUDA or OPENMP-variants offer further improvements but have not eliminated the need for the expertise on these complex architectures. Similarly, special purpose programming languages such as Microsoft's Accelerator try to lower the barrier further. They provide the programmer with a special form of GPU data structures and operations on them which are then compiled into GPU code.

In this paper, we take this trend towards a completely implicit, high-level approach yet another step further. We generate CUDA code from a MATLAB-like high-level functional array programming language, Single Assignment C (SAC). To do so, we identify which data structures and operations can be successfully mapped on GPUs and transform existing programs accordingly. This paper presents the first runtime results from our GPU backend and it presents the basic set of GPU-specific program optimisations that turned out to be essential. Despite our high-level program specifications, we show that for a number of benchmarks speedups between a factor of 5 and 50 can be achieved through our parallelising compiler.

Categories and Subject Descriptors D.3.4 [*Software*]: Programming Languages Processors [Compilers]

General Terms Algorithms, Design, Languages, Performance

Keywords Compiler, Optimization, CUDA, GPU, Code, Generation

1. Introduction

Modern Graphics Processing Units (GPUs) contain hundreds of computational cores and have become one of the most commonly used many-core architectures. The architectural aspects of these GPUs are far more complex than mainstream multi-core CPUs and programming is often facilitated by programming models such as CUDA [4] and OpenCL [10]. The ability to program GPUs with these programming models has led to the proliferation of using GPUs for accelerating many scientific applications, for example [2]. Furthermore, the relative measure of performance/price and performance/ power ratios between GPU-based architectures and CPU-based architectures further encourages the choice of GPUs.

However, these gains are not without significant challenges: Firstly, the identification and exploitation of any parallelism in the application is the responsibility of programmers. Often, this requires extensive re-factoring work rather than simple program transformations. Secondly, the GPU programming model is not oblivious to the underlying architecture. Detailed knowledge of the architecture is fundamental for writing effective GPU-based applications. Consequently, in GPU-based programs application logic is inter-twined with device-specific logic. Difficulties in separating the concerns of application logic from device logic prevents application portability.

The OpenCL [10] programming model aims to address the portability issue and offers a route for separating the device logic from the application logic. However, the application logic cannot be entirely free from low-level device logic. Compiler directive-based approaches such as *hi*CUDA [9] or support from compilers, such as PGI [13], have enabled application developers to retain application logic in the source language such as C and/or Fortran. Essentially this approach eliminates low-level device-specific logic from the application but expects the application developers to hint the compiler. Although this is a significant improvement in the direction of offering a simple programming model, the developer is still required to be familiar with the hardware to provide hints.

In this paper, we describe our auto-parallelising compiler framework for GPUs, which is achieved by augmenting our existing functional and data-parallel compiler framework, Single Assignment C (SAC) [12], with GPU-specific capabilities. In our earlier work [8], we proposed a compilation scheme for mapping high-level SAC programs to CUDA-enabled GPUs. The key idea of the scheme is to seek performance benefits from GPUs while retaining the higher-level abstractions supported by the Single Assignment C. This approach enables supporting rather high-level applications while seeking GPU-level parallelisation. In particular, the contributions of this paper are:

- Extended analysis for automating the mapping process of high-level abstractions to CUDA equivalents.
- An effective optimisation strategy for optimising memory transfers between host and GPU memories.

DAMP'11, January 23, 2011, Austin, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0486-3/11/01. . . $10.00

- Initial performance evaluations of the auto-parallelising compiler framework and of the optimisation using a suite of benchmarks on a modern GPU platform.

The rest of this paper is organised as follows: In Section 2 we provide a short background on CUDA programming model. Section 3 discusses our source language SAC. We then briefly outline the overall compilation scheme presented in our earlier work [8] in Section 4. This is then followed by the presentation of the compilation scheme for the memory transfer optimisation in Section 5. We then evaluate the performance of the framework and of the optimisation in Section 6. As part of the same section, we highlight another prototype optimisation scheme with improved results. We discuss related work in Section 7 and conclude the paper with directions for further work in Section 8.

2. CUDA Architecture and Programming Model

A Compute Unified Device Architecture (CUDA) enabled GPU is connected to the host system via a high-speed shared bus, such as PCI Express. We show an internal arrangement of a typical GPU in Figure 1. Each GPU consists of an array of Streaming Multiprocessors. Each streaming multiprocessor is packed with a number of scalar processing cores, named Streaming Processors. These scalar processors are the fundamental computing units which execute CUDA threads. For example, the Nvidia Tesla C1060 GPU has 30 streaming multiprocessors and each streaming multiprocessor consists of eight streaming processors, yielding 240 processing cores in total. In CUDA, all threads are created and managed by the hardware. As a result, the overheads are almost negligible and this leads to the possibility of executing a large number of threads at a time.

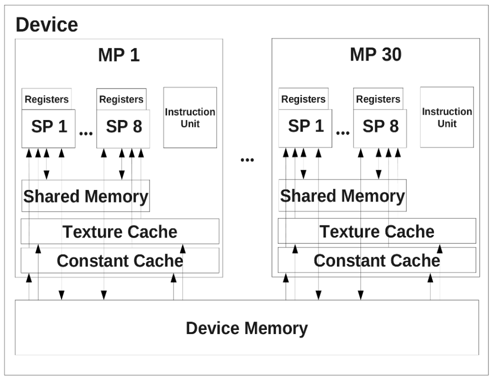


Figure 1. GPU hardware architecture

The CUDA programming model, which is an extension of the C programming language, relies on this hardware support to provide concurrency. In the model, computations are expressed as special functions known as *kernels*. A kernel is launched from the host-CPU and executed by N threads using the available computational cores (and N is usually in the range of several thousands). All threads are organised as a 1- or 2-dimensional grid of thread blocks. Each block can be 1-, 2- or 3-dimensional. Threads in a block are assigned to the same streaming multiprocessor during execution. With a unique numbering scheme for threads, each thread can be made to compute on a different subset of the input data so that the execution leads to the Single Program Multiple Data (SIMD) style parallelism.

Computational units are arranged alongside a hierarchical memory system, which has no sharing scheme with the host memory system. A GPU typically has the following memory sub-systems: global memory, constant cache, texture cache and shared memory. The global memory (also known as device memory) is common to all blocks (and thus to all threads) and has a high access latency. Each streaming multiprocessor has a low-latency small shared memory which can be accessed from threads within the same block. This is very useful to exploit potential data locality among threads.

The programming model relies on manual placement of data — which means that the application developer is solely responsible for moving the data from the host memory to the device memory (or in reverse direction) and to exploit any re-use by relying on shared memory or constant cache. It is also the responsibility of the programmer to ensure that the latencies are hidden and memory requests to the device memory are linearised for best bandwidth exploitation (hardware memory coalescing). In line with the conventional parallel programming models, memory transfers (corresponding to communication overheads) may offset the benefits of parallelisation, if it dominates the execution time. As a result, it is performance critical that memory transfers around the system and within the GPU are minimised as much as possible. For example, if a kernel feeds another kernel with its output, it is beneficial to retain the data in the GPU device memory without any intermediate transfers to the host.

3. The SAC Programming Language

SAC is a purely functional and data-parallel programming language. All its basic language constructs are identical to those of C, not only with respect to their syntax but also with respect to their semantics. Despite this rather imperative look and feel, a side-effect free semantics is enforced by the exclusion of a few features of C, most notably the absence of pointers. As a replacement, extensive support for compiler-managed n-dimensional arrays has been added as well as one very powerful language construct for expressing data-parallel operations, the WITH-loop.

Various application studies demonstrate that this setting enables (i) sequential runtimes competitive with those of hand-written C and FORTRAN codes, and (ii) almost linear speedups from auto-parallelisation for shared memory systems [3].

More detailed introductions into SAC can be found elsewhere (e.g. in [12]). Here, we deem it sufficient to present a small example that demonstrates the combination of SAC's array support with standard C code.

```
1    double[.,.] relax( int n, double[.,.] A) {
2      for( i=0; i<n; i++) {
3        A = with {
4          ( [1,1] <= iv < shape( A)-1) {
5            res = 0.25*(A[iv-[0,1]]+A[iv+[0,1]]+
6                  A[iv-[1,0]]+A[iv+[1,0]]);
7          }:res;
8        }:modarray( A);
9      }
10     return( A);
11   }
```

Figure 2. Program fragment for typical scientific code.

Figure 2 shows a program fragment archetypical for many scientific codes which we will use throughout this paper to exemplify our transformations. It contains the definition of a function `relax` which takes two arguments: a scalar integer `n` and a two-dimensional array (matrix) of double elements `A`. In the function body, the array `A` is repeatedly modified within a `for`-loop. Each such modification is expressed by means of the WITH-loop in lines 3-8. It specifies that all inner elements of the new version of `A` are computed as a mean of the four direct neighbour elements of the old version of `A`. Note here, that indexing into arrays in SAC is done by means of index vectors rather than sequences of scalar indices; the operation `shape` used in line 4 returns a vector containing the number of elements of the given array; the subtraction of 1 is applied to all elements of that vector, and indexing starts at 0. As a consequence, all indices between [1,1] (inclusive) and shape(A)-1 (exclusive) denote exactly all inner elements of A. Furthermore, it is important to note that the semantics of the WITH-loop-construct ensures that all these array modifications, conceptually, happen at the same time, i.e., they can be executed in parallel.

4. Compilation of SAC into CUDA

In our parallelising compiler, instead of performing whole-program transformation, we focus on translating individual data-parallel WITH-loops to equivalent CUDA kernel functions. The transformation consists of three main steps: (1) identifying WITH-loops that are eligible to be executed on GPUs (i.e. CUDA-WITH-loops), (2) inserting data transfer instructions for arrays accessed in and generated by the CUDA-WITH-loops, and (3) outlining CUDA-WITH-loop code as CUDA kernel functions. In the following sections, we use the same WITH-loop example from Section 3. See [8] for a more thorough and formal discussion of the compilation schemes.

4.1 Identifying Eligible WITH-loops

In SAC, WITH-loops are guaranteed to be free from dependencies. Therefore, it may appear that all WITH-loops are immediately parallelisable at a first glance. However, inherent limitations of the CUDA architecture and programming model prevents us from doing so. Firstly, the absence of stack in CUDA prevents function invocations within kernels (except device functions). This limits our selection to WITH-loops without function invocations. Secondly, a thread in CUDA cannot spawn sub-threads. This limits the parallelisation to be at one level and inhibits hierarchical parallelism. Therefore, a nested WITH-loop can only be parallelised at a single level. Theoretically, WITH-loop at any nesting level (provided it meets the first condition) can be parallelised because it is guaranteed to be dependency free. However, to amortise the overheads of thread creation and synchronisation, we favours coarser granularity and therefore parallelise only the outermost WITH-loop. Candidate WITH-loops to be translated to CUDA are referred to as CUDA-WITH-loops in the following sections.

4.2 Inserting Data Transfers

When translating CUDA-WITH-loops to kernels, free array variables found inside the loops are in fact host variables and they should eventually be mapped to the device memory before being accessed in kernels. For this purpose, we extended the type system to accommodate both host- and device-typed variables With this notion, the compiler performs a data flow analysis to find all free host array variables (FVs) in a CUDA-WITH-loop. These variables are then converted to corresponding device type variables (denoted by $\mathtt{var}^D$) via the dedicated instruction, host2device . The other instruction, device2host , converts the result from device type to host type. Data transfers between host and device memory spaces are implied during the conversion. In our running example, a host2device instruction is inserted before the CUDA-WITH-loop to map a free variable A of host-type to a device-type variable A^D and of all occurrences there in. Similarly, result of a CUDA-WITH-loop(device-typed variable, B^D in our case), should be transferred back to host via a device2host instruction. In our case, the promise of single assignment form and our extended type system make this transformation readily achievable. All local variables, such as vals, are mapped to registers or local memory assigned to each thread. The outcome after applying this transformation to the example is shown in Figure 3.

```
A^D = host2device( A);
B^D = cuda_with {
          ( [1,1] <= iv=[i,j] < [4095,4095]) {
              res = 0.25*(A^D[i][j-1]+A^D[i][j+1]+
                     A^D[i-1][j]+A^D[i+1][j]);
          }:res;
    }:modarray( A^D);
B = deivce2host( B^D);
```

Figure 3. Example CUDA-WITH-loop with data transfers inserted.

4.3 Creating CUDA Kernels

After proper data transfer instructions are inserted, each WITH-loop partition is outlined as a kernel function and replaced by the corresponding invocation to it (See Figure 4). The lower and upper bounds of each partition determine the total number of threads to be created. Threads are organised as a hierarchy consisting of a 2D grid of 2D blocks. Each block is of size $\mathtt{BLOCKSZ} \times \mathtt{BLOCKSZ}$, where BLOCKSZ is a predefined constant and can be changed by setting a command line option. Length of the grid along each dimension is calculated by dividing the corresponding length of the partition by BLOCKSZ. Note that we also need to add the result by one to take care of the cases when the length is not evenly divisible. Both device variables B^D and A^D, along with all shape and bound information, are passed to the kernel as parameters. Inside the kernel, each thread calculates its absolute indices (i and j) from CUDA predefined variables blockIdx, threadIdx and blockDim. Guarding code is also added to ensure threads do not access array positions beyond the upper bound. Each thread then computes a linear memory offset, wlidx, from both its absolute indices and the shape information. After performing the computation, the final result res is assigned to B^D at a position specified by wlidx.

```
cudaMemcpy(A,A^D,size(A),cudaMemcpyHostToDevice);
int d0 = 4095 - 1;
int d1 = 4095 - 1;
dim3 grid( d1/BLOCKSZ+1, d0/BLOCKSZ+1);
dim3 block( BLOCKSZ,BLOCKSZ);
CUDA_kernel<<<grid, block>>>
(B^D,4096,4096,1,1,4095,4095,A^D);
cudaMemcpy(B,B^D,size(B),cudaMemcpyDeviceToHost);

__global__ void CUDA_kernel
(float B^D,int shp0,int shp1,int lb0,int lb1,
 int ub0,int ub1,float A^D)
{
    int i=blockIdx.y*blockDim.y+threadIdx.y+lb0;
    if( i >= ub0) return;
    int j=blockIdx.x*blockDim.x+threadIdx.x+lb1;
    if( j >= ub1) return;
    int wlidx = i*shp1+j;
    res = 0.25*(A^D[i][j-1]+A^D[i][j+1]+
           A^D[i-1][j]+A^D[i+1][j]);
    B^D[wlidx] = res;
}
```

Figure 4. Translating example CUDA-WITH-loop to kernel function.

5. Optimising Memory Transfers

In Section 4, we outlined the basic mechanisms underpinning the auto-parallelisation of SAC programs to CUDA. However, the baseline compilation scheme is not sufficient to obtain noticeable speedups. In literature, it can be found that one of the key optimisations for achieving satisfying performance in CUDA applications is minimising the redundant data transfers between different memories of the system. Among these, transfers between host and device memories are considerably expensive. Since data in global memory is persistent across kernel invocations, a large portion of the data exchanges can be eliminated altogether provided that subsequent kernel invocations re-use the retained data. In our case, the promise of single assignment and the support from type system to differentiate host- and device-side arrays makes this optimisation readily achievable.

We implemented two different optimisation passes to minimise such data transfers. They are repeatedly applied to the program until no more transfers can be eliminated.

5.1 Retention of Invariant Arrays

This case is prevalent in the intermediate program with several data transfers concerning the same host array. Examples include when the result of a CUDA-WITH-loop is consumed by a subsequent CUDA-WITH-loop without being modified in between; and when repeated read-only accesses to one or more arrays among a number of CUDA-WITH-loops. In these cases, we perform only the first transfer and subsequent transfers are considered as references to the same array, which can be attributed to a device variable. We show the compilation scheme for this case is shown in Figure 5.

```
   [[ a = device2host(  a^D);      ]]
 C [[ assigns;                     ]]
   [[ b^D = host2device(  a);      ]]

     { a = device2host( a^D );
   = { assigns;
     { b^D = a^D;
```

Figure 5. Compilation Scheme for *Memory Transfer Optimisation*. The original notion of variables remain the same: variable x^D represents a device counterpart of x.

The notation of the compilation scheme is such that $\mathcal{C} = [[S_i]] = S_o$ generates a code block S_o when the translation scheme *memopt* is applied to a statement block S_i. In particular, the optimisation schemes look for host2device transfers on unmodified variables derived from device2host transfers (Figure 5) or for redundant host2device transfers of the same variable. Since the intermediate representation of the code is in SSA form, transfers in both cases can easily be detected and replaced by device variable assignments. Note that one of the existing optimisations in the SAC compiler framework — *Common Subexpression Elimination* has been leveraged to achieve this effect.

An additional opportunity for optimising memory transfers occurs when data transfers are enclosed within *for* loops. Since the optimisation scheme discussed in Section 5.1 focuses only on minimising transfers within the same basic block, it does not attempt to move transfers between different blocks (in this case, from *for* loop to its enclosing context). The purpose of the optimisation outlined in this section is to hoist those transfers out of the *for* loop subject to certain conditions.

5.2 Hoisting Data Transfers from FOR-Loops

Due to the fact that SAC converts all *for* loops into tail-end recursive functions, hoisting data transfers actually requires modifying the function signature to accept/return device-typed variables. We show the compilation scheme in Figure 6 for one of such cases (hoisting host-to-device transfer) and the other case is implemented similarly. A host2device memory transfer can be lifted out of the current enclosing *for* loop if the following three conditions are satisfied:

- **Condition 1**: The host2device transfers an incoming argument ($\texttt{arg}_k$). In other words, any other transfers of variables whose scope is limited to the enclosing *for*-loop cannot be hoisted.
- **Condition 2**: The *k*th parameter, $\texttt{param}_k'$, of the recursive loop function invocation is either defined by a device2host transfer instruction or $\texttt{arg}_k$ itself.
- **Condition 3**: The $\texttt{arg}_k$ is not referenced in any instructions of the loop other than host2device transfers. The first level optimiser will detect and replace them by appropriate device variable assignments.

```
   [[ res_1...res_m = loop_fun( param_1...param_k...param_n);                ]]
   [[                                                                         ]]
   [[ T_1...T_m loop_fun( arg_1...arg_k...arg_n) {                            ]]
   [[   assigns_1;                                                            ]]
   [[   a^D = host2device( arg_k);                                            ]]
   [[   assigns_2;                                                            ]]
 C [[   param'_k = device2host( v^D);                                         ]]
   [[   assigns_3;                                                            ]]
   [[   if( cond)                                                             ]]
   [[   { res'_1...res'_m = loop_fun( param'_1...param'_k...param'_n); }      ]]
   [[   res_1 = ( cond ? res'_1  : v_1);                                      ]]
   [[   ...                                                                   ]]
   [[   res_m = ( cond ? res'_m  : v_m);                                      ]]
   [[   return( res_1...res_m);                                               ]]
   [[ }                                                                       ]]

     { a^D = host2device( param_k);
     { res_1...res_m = loop_fun( param_1...a^D...param_n);
     {
     { T_1...T_m loop_fun( arg_1...arg^D_k...arg_n) {
     {   assigns_1;
     {   assigns_2;
   = {   param'_k = device2host( v^D);
     {   assigns_3;
     {   if( cond)
     {   { res'_1...res'_m = loop_fun( param'_1...v^D...param'_n);}
     {   res_1 = ( cond ? res'_1 : v_1)
     {   ...
     {   res_m = ( cond ? res'_m : v_m)
     {   return( res_1...res_m);
     { }
```

Figure 6. Compilation Scheme for *Memory Transfer Optimisation* in *for* loops by hoisting host2device transfers.

Note that unlike the cases when $\texttt{param}_k'$ is the same as $\texttt{arg}_k$, traditional *Loop Invariant Removal* (lir) optimisation is not able to hoist the transfer when $\texttt{param}_k'$ is defined by a device2host instruction (show in Figure 6). However, this CUDA specific optimisation is capable of detecting such pattern and replacing $\texttt{param}_k'$ by its device counterpart. Such transformation converts the host2device transfer into a loop invariant instruction and can therefore be hoisted by a subsequent (lir) optimisation. The hoisted host2device is placed immediately before the loop function call. The corresponding arguments and parameters are also changed to their device counterparts.

Figure 7 illustrates how the *for* loop in our example can benefit from this optimisation (assuming that it has already been converted into loop function). The host2device at line 4 transfers the last argument of the function, A, which is not referenced anywhere in the function body. Furthermore, the last parameter of the recursive function call, A_new, is defined by a device2host transfer. Since all three conditions discussed above are satisfied, the compiler can replace the parameter A_new by $\texttt{A}^\texttt{D}$_new. This exposes a loop invariant host2device instruction to the (lir) optimisation which eventually hoists it from the loop (See the lower half of Figure 7).

```
int, double[4096,4096]
relax_loop( int i, int n,
            double[4096,4096] A) {
  A^D = host2device( A);
  A^D_new = with...( A^D)...;
  A_new = device2host( A^D_new);
  i_new = i+1;
  p = i_new < n;
  if( p)
  { i_res, A_res =
      relax_loop( i_new,n,A_new);}
  i_out = ( p ? i_res : i_new);
  A_out = ( p ? A_res : A_new);
  return( i_out, A_out);
}
```

```
int, double[4096,4096]
relax_loop( int i, int n,
            double[4096,4096] A^D) {

  A^D_new = with...( A^D)...;
  A_new = device2host( A^D_new);
  i_new = i+1;
  p = i_new < n;
  if( p)
  { i_res, A_res =
      relax_loop( i_new,n,A^D_new);}
  i_out = ( p ? i_res : i_new);
  A_out = ( p ? A_res : A_new);
  return( i_out, A_out);
}
```

Figure 7. Hoisting data transfers from our example *for* loop.

6. Performance Evaluation

The main aim of our evaluation is to quantify the overall performance benefits of our GPU backend. We deliberately chose to measure the "gross" effect including all startup and transfer overheads as that is the impact that eventually can be observed by programmers in terms of wall-clock runtime effects. Therefore, we present speedups of generated CUDA programs (referred to as SAC-CUDA) against their sequential counterparts (referred to as SAC-SEQ). We contrast these figures to results from our multi-threaded implementation (referred to as SAC-MT) using POSIX threads [7]. In addition to the speedup figures, we also provide giga-flops per second for all benchmarks. This provides the reader with a lower bound of the effectiveness that is achieved. For several benchmarks, we investigate different problem sizes in order to demonstrate the impact of the corresponding overheads on the overall performance.

It is important to notice that both SAC-SEQ and SAC-MT implementations are run entirely on the host CPU whereas the SAC-CUDA version is run both on host CPU (sequential code) and on the GPU (parallel code). We use the SAC-SEQ performance as the baseline performance. We also assume that the entire dataset required by each benchmark fits into the main memories of both systems (CPU and GPU). The UNIX *time* command is used to measure the wall-clock runtime of each benchmark and we use the median of the measurements to compute the speedups. We apply the optimisations discussed in earlier sections and report our findings below.

We have selected two suites of benchmarks: one contains a subset of the Livermore loops and the other contains various complete applications from different domains. Out of 24 Livermore loops which are traditionally used to measure parallel program performance, we select only a subset of them for our evaluation, leaving out loops that require parallel reductions which are currently not supported by the compiler. The complete applications include Matrix Multiply, NBody Simulation, Mandelbrot computation, a 3D PDE solver (PDE1),a 2D successive over relaxation scheme (SOR), Gauss-Jordan elimination, KPI (Numerical solver for an oscillatory differential equation using Kadomtsev Petviashvili method) and insertion sort.

All benchmarks use double precision floating point numbers and are executed on a system with an nVidia Tesla C1060 GPU. The device has 30 streaming multiprocessors. Each multiprocessor has 8 streaming processors clocked at 1.3 GHz. The total amount of device memory is 4 GB. The theoretical peak performance of this device on its own is 76 GFLOPS/sec in double precision.

The CPU is a 1.6 GHz Xeon 5110 dual core processor with 4 MB L2 cache. This architecture can deliver a maximum floating point performance of 12.8 GFLOPS/sec when fully utilising both cores.

The GPU is connected to the CPU through a 16x PCI express bus. We use CUDA version 3.0 and enable *-O3* option for all compilations. The SAC version of the compiler is v1.00-beta-r-17140.

6.1 Performance of the Livermore Loops

Speedups of SAC-CUDA code over SAC-SEQ code are shown in Figure 8 with and without the `memopt` .

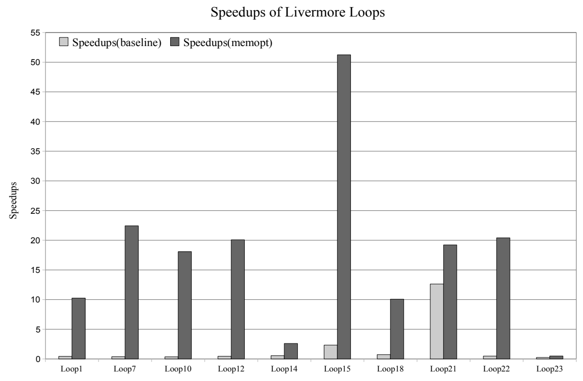


Figure 8. Speedups of Livermore loops.

Except for Loop15 and Loop21, the baseline approach without optimisations leads to no speedups. However, when `memopt` is applied, performance of all loops are improved significantly, except Loop23, for which the speedup is rather negligible. An analysis of Loop23 shows that it consists of a for-loop with loop-carried dependency inside the main stepping loop, which prevents the `memopt` optimisation. The sequential workload of this loop occupies more than 90% of the total execution time and thus eliminating any noticeable improvement due to parallelisation. Loop14 achieved relatively low speedups compared to other loops because it contains indirect subscripts in array expressions resulting in rather random access patterns of the device memory hindering effective hardware memory access coalescing. All other loops show considerable performance gains when `memopt` is enabled with the maximum speedup of $52\times$ for Loop15, which is mainly due to the high compute intensity of its kernel.

Table 1 shows the absolute performance of these loops in giga flops per second. We can observe that the best case, i.e. Loop21, gets close to about 40% of the peak performance of the GPU card despite taking all overheads into account.

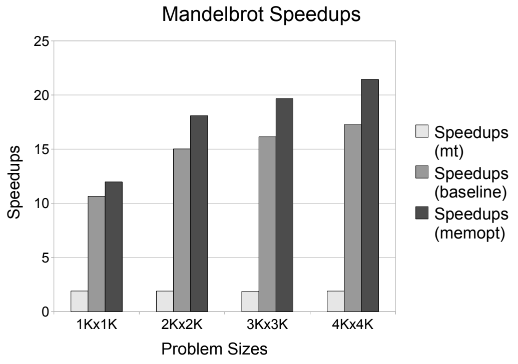
6.2 Performance of Complete Applications

The performance of complete applications are shown in Figures 9, 10, 11 and 12. One of the applications which demonstrates real benefits of auto-parallelisation is Mandelbrot (See Figure 9). It contains three WITH-loops, each with a *for* loop inside. This, when parallelised, provides an ample amount of parallelism exploitable

Loop	Seq	Baseline	Memopt
Loop1	0.93	0.4	9.5
Loop7	0.43	0.16	9.73
Loop10	0.1	0.1	0.1
Loop12	0.08	0.04	1.68
Loop14	0.19	0.1	0.49
Loop15	0.28	0.66	14.55
Loop18	0.71	0.52	7.13
Loop21	1.52	19.22	29.24
Loop22	0.56	0.26	11.45
Loop23	0.01	0.01	0.01

Table 1. Gflops/sec of Livermore loops

by the GPU. Increasing the problem sizes leads to a better workload/overhead ratio, achieving the best speedup of 22× against the sequential implementation. Table 2 shows the absolute performance of all complete applications. For Mandelbrot, we achieve about 40% of the theoretical peak performance of the GPU, despite measuring wall-clock times.



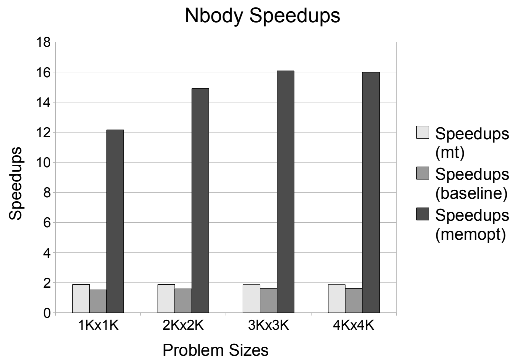


Figure 9. Speedups of Mandelbrot and NBody Simulation.

For the NBody simulation, basic CUDA auto-parallelisation yields a speedup of less than 2× (lower than SAC-MT speedups), which is almost invariant to problem sizes. However, the benchmark includes several WITH-loops embedded within a *for* loop, which initiates redundant memory transfers between host and the device for each iteration. As a result, benefit from `memopt` is rather significant, improving the overall performance by several folds. Speedups increase with increasing problem sizes as more concurrency is exposed.

The next application we benchmarked is a blocked version matrix multiplication (See Figure 10). As shown in the figure, even basic CUDA auto-parallelisation brings noticeable speedups by a factor of 10× to 13×. However, further improvements are marginal with increasing problem sizes. This is because: 1) to compute each output element, a CUDA thread performs n multiply-add operations and accesses $2n$ data, where n is the matrix dimension. 2) accesses to one row of a matrix is un-coalesced among all threads, which leads to large amount of memory transactions to the device memory. To evaluate the absolute performance, we also measured the giga flops on both CPU and GPU. As we can see, the SAC-CUDAversion achieves a significantly higher performance than either SAC-SEQ or SAC-MT. This also represents the maximally achievable performance of a typical unoptimised matrix multiplication on Tesla C1060.

Benchmark	Size	Seq	MT	Baseline	Memopt
Mandelbrot	1Kx1K	1.4	2.7	15.4	17.4
	2Kx2K	1.4	2.7	21.8	26.3
	3Kx3K	1.4	2.7	23.6	28.8
	4Kx4K	1.4	2.7	25.1	31.2
Nbody	1Kx1K	0.4	0.8	0.6	5.1
	2Kx2K	0.4	0.8	0.7	6.3
	3Kx3K	0.5	0.9	0.8	8.1
	4Kx4K	0.5	0.9	0.8	7.5
PDE1	32^3	1.0	1.8	0.3	0.6
	64^3	1.0	1.9	0.5	2.5
	128^3	0.9	1.4	0.7	4.6
	256^3	0.8	0.8	0.7	8.0
Relaxation	1Kx1K	0.4	0.5	0.26	4.3
	2Kx2K	0.4	0.5	0.26	5.3
	3Kx3K	0.5	0.5	0.26	5.5
	4Kx4K	0.5	0.5	0.26	5.5
Matmul	1Kx1K	1.0	1.9	9.1	10.1
	2Kx2K	1.0	1.9	11.4	12.1
	3Kx3K	1.0	1.9	11.8	12.3
	4Kx4K	1.0	1.9	11.9	12.4

Table 2. Gflops/sec of complete applications

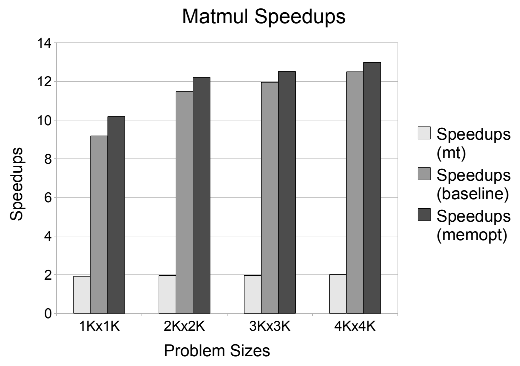
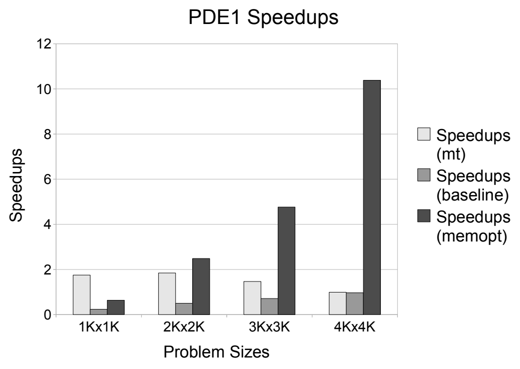


Figure 10. Speedups of Blocked Matrix Multiplication.

Our next set of benchmarks are PDE1 and SOR, both of which are stencil-based applications (See Figure 11). SAC-MT achieves some speedups. However, CUDA baseline performance is slower than SAC-SEQ. This is mainly because the stencil-like memory access patterns of these applications hinders effective memory coalescing. The `memopt` improves the speedups significantly by retaining the data across successive iterations.

The final set of applications, where we found slowdowns instead of speedups are Gauss-Jordan elimination, Insertion sort and KP1. (See Figure 12). Although `memopt` improves the performance of Gauss-Jordan elimination significantly, in overall it was slower than



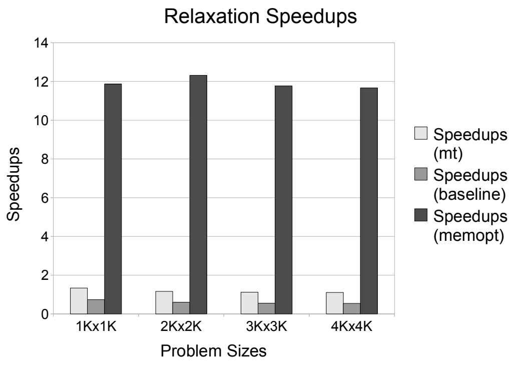
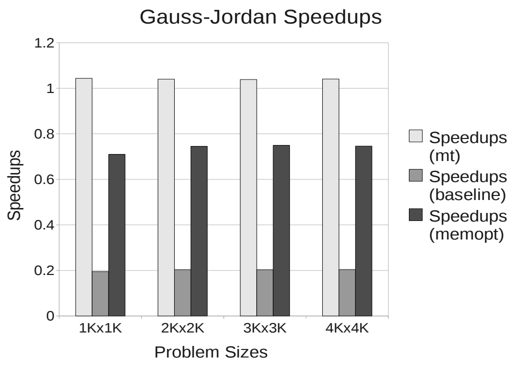
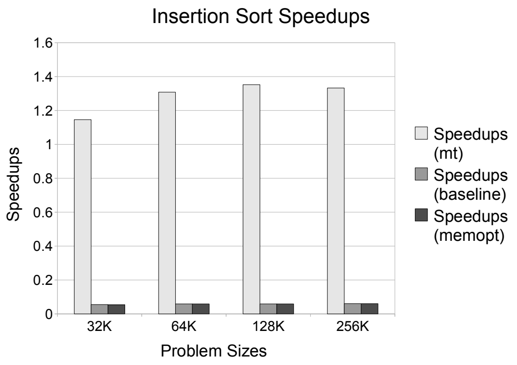


Figure 11. Speedups of PDE1 and Relaxation applications.





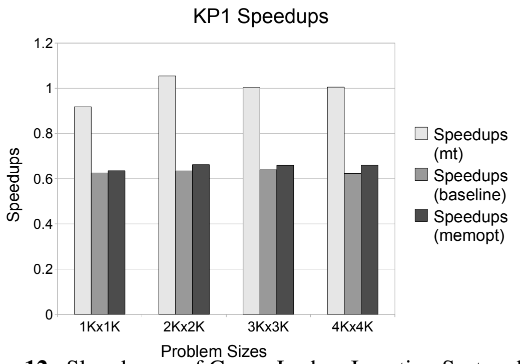


Figure 12. Slowdowns of Gauss-Jordan, Insertion Sort and KP1.

the SAC-SEQ and SAC-MT version. In other two cases, we did not find any improvements when applying `memopt`.

Upon further investigation we found that these applications contain parallel CUDA-WITH-loops whose results are consumed by the interleaved sequential code. This results in memory transfers between successive CUDA-WITH-loops not eliminated by the `memopt` transformations. Such redundant transfers and sequential code blocks (i.e. *for* loops with carried dependencies) are detrimental to the performance.

6.3 Further Analysis

Figure 13 shows an example when such execution pattern can result in extra data transfers.

Two CUDA-WITH-loops (lines 1 and 10) will be executed in parallel on the GPU. Instructions in lines 7 and 8 comprises a serial code block (selecting an array element from `a` and modifying an array element in `a`) which should be executed on the CPU. This demands two memory transfers from(to) GPU to(from) CPU (lines 6 and 9). These transfers cannot be eliminated by the `memopt` scheme described in Section 5. However, if the array selection is sinked into the second CUDA-WITH-loop (executed redundantly by all threads) and the array modification is executed in a single threaded CUDA kernel, both transfers can be eliminated (assuming that neither `val` nor `a_SSA0` is referenced later in the program). The final optimised code is shown in Figure 14. In the optimised version, the transfers are replaced by temporary device variable which is subsequently passed to a kernel with single thread.

In order to be able to quantify the impact of these sequential code snippets, we re-ran our measurements on Gauss-Jordan and Insertion Sort with hand-patched codes that avoid those transfers. We refer to these codes as `expar`. Both benchmarks benefit vastly from this yielding speedups of $14\times$ and $4.5\times$ respectively (See Figure 15).

However, this does not yet solve the performance issues found for KP1. In KP1 the problems stem from an unfavourable nesting of *for* loops and WITH-loops. They lead to a *for* loop which contains WITH-loops, whose results are partially data-dependent from one iteration to the next. In the context of traditional POSIX-thread based concurrent executions on shared-memory systems, this is not a problem as the sharing of the entire data comes for free. In the context of CUDA, the situation is different. Here, our desire to minimise transfers now leads to the necessity to transfer between individual loop iterations. Further investigation into the code revealed that by manually applying some optimisations between *for* loops and WITH-loops similar to the traditional loop interchange these problems can be overcome. Figure 16 shows our hand-optimised results.

The absolute performance achieved by these modifications are shown in Table 3.

7. Related Work

The work described in this paper partly inherits the original approach from Grelck *et. al.* [7] where they consider auto-parallelising individual WITH-loop for SMP systems. In their approach, the thread assignments are made to disjoint partitions of the iteration

```
a^D = cuda_with {
          ( lb <= iv < ub) {
              assigns;
          }:res;
      }:genarray( shp, def);
a = device2host( a^D);
val = idx_sel( a, i);
a_SSA0 = idx_modarray( a, j, 100);
a_SSA0^D = host2device( a_SSA0);
b^D = cuda_with {
          ( lb <= iv < ub) {
              .. = ..val..;
              .. = ..a_SSA0^D..;
          }:res;
      }:genarray( shp, def);
```

Figure 13. Program with interleaved parallel and sequential code.

```
T CUDA_single_thread_kernel( T tmp^D, T j) {
    a_SSA1^D = idx_modarray( tmp^D, j, 100);
    return( a_SSA1^D);
}

a^D = cuda_with {
          ( lb <= iv < ub) {
              assigns;
          }:res;
      }:genarray( shp, def);
tmp^D = a^D;
a_SSA1^D = CUDA_single_thread_kernel( tmp^D, j);
a_SSA0^D = a_SSA1^D;
a_tmp^D = a^D;
b^D = cuda_with {
          ( lb <= iv < ub) {
              val = idx_sel(a_tmp^D, i);
              .. = ..val..;
              .. = ..a_SSA0^D..;
          }:res;
      }:genarray( shp, def);
```

Figure 14. Program after applying reduce memory transfer optimisation.

Benchmark	Size	Baseline	Memopt	Expar
Gauss-Jordan	1Kx1K	0.03	0.12	1.33
	2Kx2K	0.03	0.13	2.13
	3Kx3K	0.03	0.13	2.3
	4Kx4K	0.03	0.12	2.2
Insertion Sort	32K	0.03	0.03	0.68
	64K	0.03	0.03	1.16
	128K	0.03	0.03	1.72
	256K	0.03	0.03	2.22
KP1	1Kx1K	0.17	0.17	3.51
	2Kx2K	0.12	0.12	8.52
	3Kx3K	0.12	0.12	10.4
	4Kx4K	0.12	0.13	11.8

Table 3. Gflops/sec of three benchmarks before and after applying expar

space in a reconfigurable manner. Although looks similar, their work differs from us in several ways. Our work benefits from micro-threading mechanisms of CUDA whereas their work relies on POSIX threads. Furthermore, due to the nature of GPU-based applications, we aggressively optimise for redundant memory transfers between the host and the device. Since SMP-based systems do not suffer from such transfers, their compiler do not perform such optimisations. These differences will become magnified as we deploy additional optimisations to exploit different aspects of GPUs.

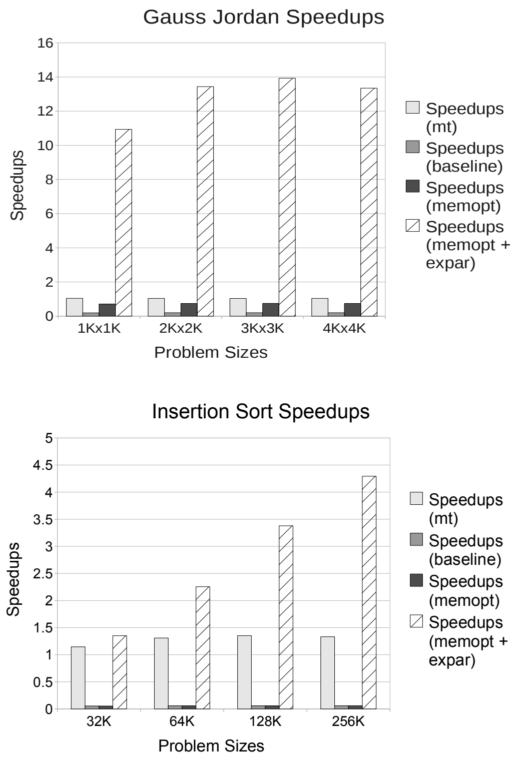


Figure 15. Speedups of Gauss-Jordan and Insertion Sort after expanding parallel region.

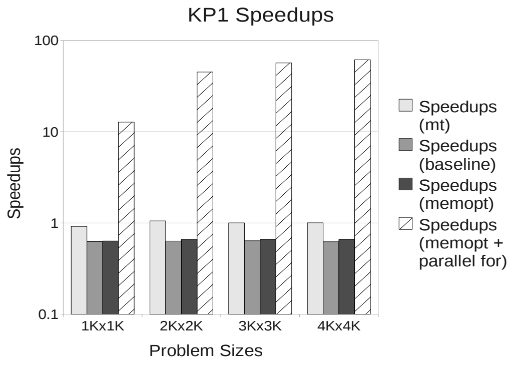


Figure 16. Speedups of KP1 when for-loops are executed in parallel.

Apart from this, mapping high-level code to GPGPUs is an active area of research and several frameworks are evolving in this direction. Microsoft's Accelerator Framework [5] is an example in that direction. Directive-based approaches offer a high-level alternative, whereby the underlying compiler is hinted using compiler directives for GPU-specific optimisations. Commercial compilers from PGI [13] and *hi*CUDA [9] aim to provide abstractions at the device level so that the separation of application logic from the device logic becomes easier. The directives in the language offer better expressibility and minimise the mechanical steps to manage device-specifics, such as memory allocation. They offer different types of directives to manage the data and computation. The former one enables the specification of the data movements between different memories of the device whilst the latter enables different partitioning strategies for threads. However, we believe that these abstractions are far from freeing the application programmers from device-specific programming.

Mapping domain-specific languages to GPGPUs is another approach where the mapping problem is simplified by the design of the domain-specific language under consideration, for example [6]. However, such approaches may not render a generic solution as ours.

In [11], Lee *et. al.* present a translator to map OPENMP programs to CUDA programs, where they target work sharing directives for parallelisation which are eventually mapped as kernel regions. They optimise the data accesses to the global memory using suitable loop transformations. Since kernel boundaries are demarcated by work-sharing constructs in OPENMP, this particular optimisation is not global. In [1], Baskaran *et. al.* describe a compiler framework where they target affine loop nests for parallelisation using polyhedral model. In particular, they optimise the loop nests to have effective data transfers between global and shared-memories of the device. However, the analysis does not extend beyond loop nests.

8. Conclusions and Further Work

One of the key challenges in using GPUs is that application programmers are expected to have expert knowledge on the GPU architecture. Different programming models improved this situation but it is still a challenge to program at the abstraction level. In this paper, we offered an implicit high-level approach: we generated CUDA code from a high-level functional array programming language, Single Assignment C (SAC). Our compiler targets the data parallel loops, WITH-loops in SAC for parallel execution. In addition to mapping the WITH-loops to CUDA kernels, we performed additional transformations for improving the performance even further. We found minimising redundant data transfers as a key optimisation technique. Using a suite of benchmarks we demonstrated that this optimisation can lead to significant performance gains. We also did a proof-of-concept demonstration on a subset of our benchmark suite to confirm the effectiveness of expanding parallel regions beyond WITH-loops.

The work we presented here is part of a larger effort in producing an auto-parallelising compiler framework for heterogeneous architectures. A number of issues to be addressed in reaching this goal are:

- The expanding parallel region transformation requires further development to include other looping constructs, such as `for` loops.
- The current implementation does not optimise kernels on their memory usage such as different types of memory (constant, cache or shared) or memory coalescing. A careful data re-use analysis is required to determine placement of data or to reorder the accesses.
- The compiler framework, SAC, is flexible enough for retargeting the code generation for different languages. Accelerator-based OPENMP like directives [9, 13] is an attractive route for retaining the abstractions while addressing heterogeneity.
- In our work, we fervently hoped that the data size does not exceed size of the device memory. However, several class of applications do demand processing on rather large amount of data. Such computations on large data sets will require advanced features such as support for streaming from our compiler.
- Finally, the parallelising framework targets only the WITH-loops for mapping the execution to kernels. We would like to extend this support towards traditional `for` loops.

References

[1] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In *ICS '08: Proceedings of the 22nd Annual International Conference on Supercomputing*, pages 225–234, New York, USA, 2008. ACM.

[2] M. Bernaschi, M. Fatica, S. Melchionna, S. Succi, and E. Kaxiras. A Flexible High-Performance Lattice Boltzmann GPU Code for the Simulations of Fluid Flows in Complex Geometries. *Concurrency and Compututation : Practice and Experience*, 22(1):1–14, 2010.

[3] R. Daniel, J. Carl, K. Alexei, S. Sven-Bodo, and V. S. Alexander. Numerical Simulations of Unsteady Shock Wave Interactions Using SaC and Fortran-90. In *Lecture Notes in Computer Science*, volume 5698, pages 445–456. Springer-Verlag, 2009.

[4] David B. Kirk, Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann, 2010.

[5] David Tarditi and Sidd Puri and Jose Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (12th ASPLOS'06)*, pages 325–335. ACM 2006, 2006.

[6] Fred V. Lionetti and Andrew D. McCulloch and Scott B. Baden. Source-to-Source Optimization of CUDA C for GPU Accelerated Cardiac Cell Modeling. In Pasqua D'Ambra, Mario Rosario Guarracino, and Domenico Talia, editors, *Proceedings of the 16th International Euro-Par Conference (Euro-Par 2010), Part I*, volume 6271 of *Lecture Notes in Computer Science*, pages 38–49. Springer-Verlag, 2010.

[7] Clemens Grelck. SAC – From High-Level Programming with Arrays to Efficient Parallel Execution. In *Proceedings of the 2nd International Workshop on High Level Parallel Programming and Applications (HLPP'03)*, Paris, 2003.

[8] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. Towards Compiling SaC to CUDA. In *Trends in Functional Programming*, volume 10, pages 33–48, Bristol, UK, 2010. Intellect.

[9] Tianyi David Han and Tarek S. Abdelrahman. hiCUDA: A High-Level Directive-Based Language for GPU Programming. In *GPGPU-2: Proceedings of 2nd Workshop on GPGPUs*, pages 52–61, New York, USA, 2009. ACM.

[10] Khronos Group. OpenCL 1.1, Last accessed November 22, 2010. `http://www.khronos.org/opencl/`.

[11] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 101–110, New York, USA, 2009. ACM.

[12] Sven-Bodo Scholz. Single Assignment C – Efficient Support for High-level Array Operations in a Functional Setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[13] Wolfe, Michael. Implementing the PGI Accelerator model. In *GPGPU '10: Proceedings of the 3rd Workshop on GPGPUs*, pages 43–50, New York, USA, 2010. ACM.