Static Compilation Analysis for Host-Accelerator Communication Optimization

Mehdi Amini[1,2], Fabien Coelho[2], François Irigoin[2], and Ronan Keryell[1]

[1] HPC Project, Meudon, France
name.surname@hpc-project.com
[2] MINES ParisTech/CRI, Fontainebleau, France
name.surname@mines-paristech.fr

Abstract. We present an automatic, static program transformation that schedules and generates efficient memory transfers between a computer host and its hardware accelerator, addressing a well-known performance bottleneck. Our automatic approach uses two simple heuristics: to perform transfers to the accelerator as early as possible and to delay transfers back from the accelerator as late as possible. We implemented this transformation as a middle-end compilation pass in the PIPS/PAR4ALL compiler. In the generated code, redundant communications due to data reuse between kernel executions are avoided. Instructions that initiate transfers are scheduled effectively at compile-time. We present experimental results obtained with the Polybench 2.0, some Rodinia benchmarks, and with a real numerical simulation. We obtain an average speedup of 4 to 5 when compared to a naïve parallelization using a modern GPU with PAR4ALL, HMPP, and PGI, and 3.5 when compared to an OPENMP version using a 12-core multiprocessor.

Keywords: Automatic parallelization, communication optimization, source-to-source compilation, heterogeneous parallel architecture, GPU.

1 Introduction

Hybrid computers based on hardware accelerators are growing as a preferred method to improve performance of massively parallel software. In 2008, *Roadrunner*, using *PowerXCell 8i* accelerators, headed the TOP500 ranking. The June 2011 version of this list and of the Green500 list both include in the top 5 three hybrid supercomputers employing NVIDIA GPUs. These highly parallel hardware accelerators allow potentially better performance-price and performance-watts ratios when compared to classical multi-core CPUs. The same evolution is evident in the embedded and mobile world (NVIDIA Tegra, etc.). In all, the near future of high performance computing appears heterogeneous.

The disadvantage of heterogeneity is the complexity of its programming model: the code executing on the accelerator cannot directly access the host memory and *vice-versa* for the CPU. Explicit communications are used to exchange data, via slow IO buses. For example, PCI bus offers 8 GB/s. This is generally thought to be *the* most important bottleneck for hybrid systems [7,9,10].

Though some recent architectures avoid explicit copy instructions, the low performance PCI bus is still a limitation.

We propose with PAR4ALL [15] an open source initiative to unify efforts concerning compilers for parallel architectures. It supports the automatic integrated compilation of applications for hybrid architectures. Its basic compilation scheme generates parallel and hybrid code that is correct, but lacks efficiency due to redundant communications between the host and the accelerator.

Much work has been done regarding communication optimization for distributed computers. Examples include message fusion in the context of SPDD (*Single Program Distributed Data*) [12] and data flow analysis based on array regions to eliminate redundant communications and to overlap the remaining communications with compute operations [13].

We apply similar methods to offload computation in the context of a host-accelerator relationship and to integrate in a parallelizing compiler a transformation that optimizes CPU-GPU communications at compile time. In this paper we briefly present existing approaches addressing the issue of writing software for accelerators (§ 2). We identify practical cases for numerical simulations that can benefit from hardware accelerators. We show the limit of automatic transformation without a specific optimization for communication (§ 3). We present a new data flow analysis designed to optimize the static generation of memory transfers between host and accelerator (§ 4). Then, using a 12-core Xeon multiprocessor machine with a NVIDIA Tesla GPU C2050, we evaluate our solution on well known benchmarks [22,6]. Finally, we show that our approach scales well with a real numerical cosmological simulation (§ 5).

2 Automatic or Semi-automatic Transformations for Hardware Accelerators

Targeting hardware accelerators is hard work for a software developer when done fully manually. At the highest level of abstraction and programmer convenience, there are APIs and C-like programming languages such as CUDA, OPENCL. At lower levels there are assembly and hardware description languages like VHDL. NVIDIA CUDA is a proprietary C-extension with some C++ features, limited to NVIDIA GPUs. The OPENCL standard includes an API that presents an abstraction of the target architecture. However, manufacturers can propose proprietary extensions. In practice, OPENCL still leads to a code tuned for a particular accelerator or architecture. Devices like FPGAs are generally configured with languages like VHDL. Tools like the Altera C-to-VHDL compiler `c2h` attempt to raise the level of abstraction and convenience in device programming.

2.1 Semi-automatic Approach

Recent compilers comprise an incremental way for converting software toward accelerators. For instance, the PGI Accelerator [23] requires the use of directives. The programmer must select the pieces of source that are to be executed on the accelerator, providing optional directives that act as hints for data allocations and transfers. The compiler generates all code automatically.

HMPP [5], the CAPS compiler, works in a similar way: the user inserts directives to describe the parameters required for code generation. Specialization and optimization possibilities are greater, but with the same drawback as OpenCL extensions: the resulting code is tied to a specific architecture.

Jcuda [24] offers a simpler interface to target CUDA from Java. Data transfers are automatically generated for each call. Arguments can be declared as IN, OUT, or INOUT to avoid useless transfers, but no piece of data can be kept in the GPU memory between two kernel launches. There have also been several initiatives to automate transformations for OpenMP annotated source code to CUDA [20,21]. The GPU programming model and the host accelerator paradigm greatly restrict the potential of this approach, since OpenMP is designed for shared memory computer. Recent work [14,19] adds extensions to OpenMP that account for CUDA specificity. But, these lead again to specialized source code.

These approaches offer either very limited automatic optimization of host-accelerator communications or none at all. OpenMPC [19] includes an interprocedural liveness analysis to remove some useless memory transfers, but it does not optimize their insertion. Recently, new directives were added to the PGI [23] accelerator compiler to precisely control data movements. These make programs easier to write, but the developer is still responsible for designing and writing communications code.

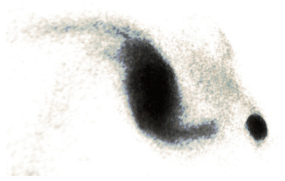
2.2 Fully Automatic Approach: Par4All

Par4All [15] is an open-source initiative that aims to develop a parallelizing compiler based on source-to-source transformations. The current development version (1.2.1) generates OpenMP from C and Fortran source code for shared memory architecture and CUDA or OpenCL for hardware accelerators including NVIDIA and ATI GPUs. The automatic transformation process in Par4All is heavily based on PIPS compiler framework phases [17,1]. The latter uses a linear algebra library [2] to analyze, transform and parallelize programs using polyhedral representations [11]. Parallel loop nests are outlined (*i.e.* extracted) in new functions tagged as eligible on the GPU. They are called *kernels* in the hybrid programming terminology. Convex array region analyses [8] are used to characterize the data used and defined by each kernel.

Par4All often parallelizes computations with no resulting speedup because communication times dominate. The new transformation presented here obtains better speedups by improving communication efficiency.

3 *Stars-PM* Simulation

Small benchmarks like the Polybench suite [22] are limited to a few kernels, sometimes surrounded with a time step loop. Thus they are not representative of a whole application when evaluating a global optimization. To address this issue, we do not limit our experiments to the Polybench benchmarks, but we also include *Stars-PM*, a particle mesh cosmological N-body code. The sequential version was written in C at *Observatoire Astronomique de Strasbourg* and was rewritten and optimized by hand using CUDA to target GPUs [3].



(a) A satellite triggers a bar and spiral arms in a galactic disc.

```
int main(int argc, char *argv[]) {
  /* Read data from a file */
  init_data(argv[1]);
  /* Main temporal loop */
  for(t = 0; t < T; t += DT)
    iteration(...);
  /* Output results to a file */
  write_data(argv[2]);
}
```

(b) Simplified scheme for simulations.

```
void iteration(coord pos[NP][NP][NP],
               coord vel[NP][NP][NP],
               float dens[NP][NP][NP],
               int data[NP][NP][NP],
               int histo[NP][NP][NP]) {

  /* Cut the tri-dimensionnal space
   * in a regular mesh */
  discretisation(pos, data);
  /* Compute density on the grid */
  histogram(data, histo);
  /* Compute potential on the mesh
   * in the Fourier space */
  potential(histo, dens);
  /* For each dimension, compute the
   * force and then update the speed */
  forcex(dens, force);
  updatevel(vel, force, data, 0, dt);
  forcey(dens, force);
  updatevel(vel, force, data, 1, dt);
  forcez(dens, force);
  updatevel(vel, force, data, 2, dt);
  /* Move particles */
  updatepos(pos, vel);
}
```

(c) Time step in the cosmological simulation.

Fig. 1. Outline of the *Stars-PM* cosmological simulation code

This simulation is a model of gravitational interactions between particles in space. It represents three-dimensional space with a discrete grid. Initial conditions are read from a file. A sequential loop iterates over successive time steps. Results are computed from the final grid state and stored in an output file. This general organization is shown in the simplified code shown in Fig. 1b. It is a common technique in numerical simulations. The processing for a time step is illustrated Fig. 1c.

3.1 Transformation Process

The automatic transformation process in PAR4ALL is based on parallel loop nest detection. Loop nests are then outlined to obtain kernel functions.

The simplified code for function `discretization(pos, data)` is provided before and after the transformation in Figs. 2 and 3 respectively. The loop nest is detected as parallel and selected to be transformed into a kernel. The loop body is outlined in a new function that will be executed by the GPU, and the loop nest is replaced by a call to a kernel launch function. PIPS performs several array regions analyses: $\mathcal{W}$ (resp. $\mathcal{R}$) is the region of an array written (resp. read) by a statement or a sequence of statements, for example a loop or a function. PIPS

```
void discretization(coord pos[NP][NP][NP],
                    int data[NP][NP][NP]){
  int i, j, k;
  float x, y, z;
  for (i = 0; i < NP; i++)
    for (j = 0; j < NP; j++)
      for (k = 0; k < NP; k++) {
        x = pos[i][j][k].x;
        y = pos[i][j][k].y;
        z = pos[i][j][k].z;
        data[i][j][k] = (int)(x/DX)*NP*NP
                      + (int)(y/DX)*NP
                      + (int)(z/DX);
      }
}
```

Fig. 2. Sequential source code for function `discretization`

also computes $\mathcal{IN}$ and $\mathcal{OUT}$ [8] regions. These are conservative over-estimates of the respective array areas used ($\mathcal{IN}$) and defined ($\mathcal{OUT}$) by the kernel. $\mathcal{IN}$ regions must be copied from host to GPU before kernel execution. $\mathcal{OUT}$ regions must be copied back afterward.

Looking at function `discretization` (Fig. 2) we observe that the `pos` array is used in the kernel, whereas `data` array is written. Two transfers are generated (Fig. 3). One ensures that data is moved to the GPU before kernel execution and the other copies the result back to the host memory after kernel execution.

```
void discretization(coord pos[NP][NP][NP],int data[NP][NP][NP]) {
 // Pointers to memory on accelerator:
 coord (*pos0)[NP][NP][NP] = (coord (*)[NP][NP][NP]) 0;
 int (*data0)[NP][NP][NP] = (int (*)[NP][NP][NP]) 0;
 // Allocating buffers on the GPU and copy in
 P4A_accel_malloc((void **) &data0, sizeof(int)*NP*NP*NP);
 P4A_accel_malloc((void **) &pos0, sizeof(coord)*NP*NP*NP);
 P4A_copy_to_accel(sizeof(coord)*NP*NP*NP, pos, *pos0);
 P4A_call_accel_kernel_2d(discretization_kernel,NP,NP,*pos0,*data0);
 // Copy out and GPU buffers deallocation
 P4A_copy_from_accel(sizeof(int)*NP*NP*NP, data, *data0);
 P4A_accel_free(data0);
 P4A_accel_free(pos0);
}
// The kernel corresponding to sequential loop body
P4A_accel_kernel discretization_kernel( coord *pos, int *data ) {
   int k; float x, y, z;
   int i = P4A_vp_1; //  P4A_vp_* are mapped from CUDA BlockIdx.*
   int j = P4A_vp_0; //  and ThreadIdx.* to loop indices
   // Iteration clamping to avoid GPU iteration overrun:
   if (i<=NP&&j<=NP)
      for(k = 0; k < NP; k += 1) {
         x = (*(pos+k+NP*NP*i+NP*j)).x;
         y = (*(pos+k+NP*NP*i+NP*j)).y;
         z = (*(pos+k+NP*NP*i+NP*j)).z;
         *(data+k+NP*NP*i+NP*j) = (int)(x/DX)*NP*NP
                                + (int)(y/DX)*NP
                                + (int)(z/DX);
      }
}
```

Fig. 3. Code for function `discretization` after automatic GPU code generation

3.2 Limit of This Approach

Data exchanges between host and accelerator are executed as DMA transfers between RAM memories across the PCI-express bus, which currently offers a theoretical bandwidth of 8 GB/s. This is really small compared to the GPU memory bandwidth which is close to 150 GB/s. This low bandwidth can annihilate all gain obtained when offloading computations in kernels, unless they are really compute intensive.

With our hardware (see § 5), we measure up to 5.6 GB/s from the host to the GPU, and 6.2 GB/s back. This is obtained for a few tens of MB, but decreases dramatically for smaller blocks. Moreover this bandwidth is reduced by more than a half when the transfered memory areas are not *pinned*—physically contiguous and not subject to paging by the virtual memory manager. Figure 5 illustrates this behavior.

In our tests, using as reference a cube with 128 cells per edge and as many particles as cells, for a function like `discretization`, one copy to the GPU for particle positions is a block of 25 MB. One copy back for the particle-to-cell association is a 8 MB block. The communication time for these two copies is about 5 ms. Recent GPUs offer ECC hardware memory error checking that more than doubles time needed for the same copies to 12 ms. Each buffer allocation and deallocation requires 10 ms. In comparison, kernel execution requires only 0.37 ms on the GPU, but 37 ms on the CPU. We note that the memory transfers and buffer allocations represent the largest potential for obtaining high speedups, which motivates our work.

3.3 Observations

In each time step, function `iteration` (Fig. 1c) uses data defined by a previous one. The parallelized code performs many transfers from the GPU followed immediately by the opposite transfer.

Our simulation (Fig. 1b) exemplifies the common pattern of data dependencies between loop iterations, where the current iteration uses data defined during previous ones. There is clear advantage in allowing such data to remain on the GPU, with copies back to the host only as needed for checkpoints and final results.

4 Optimization Algorithm

We propose a new analysis for the compiler middle-end to support efficient host-GPU data copying. The host and the accelerator have separated memory spaces, this analysis annotates internally the source program with information about where up-to-date copies of data lie—in host and/or GPU memory. This allows additional transformation to statically determine good places to insert asynchronous transfers with a simple strategy: Launch transfers from host to GPU as early as possible and launch those from GPU back to host as late as possible, while still guaranteeing data integrity. Additionally, we avoid launching transfers inside loops wherever possible. We use a heuristic to place transfers as high as possible in the call graph and in the AST.[1]

4.1 Definitions

The analysis computes the following sets for each statement:

- $\mathcal{U}_A^>$ is the set of arrays known to be *u*sed next ($>$) to the *a*ccelerator;
- $\mathcal{D}_A^<$ is the set of arrays known to be lastly ($<$) *d*efined on the *a*ccelerator;
- $\mathcal{T}_{H\to A}$ is the set of arrays to transfer to the accelerator memory space immediately after the statement;
- $\mathcal{T}_{A\to H}$ is the set of arrays to transfer from the accelerator on the host immediately before the statement.

4.2 Intraprocedural Phase

The analysis begins with the $\mathcal{D}_A^<$ set in a forward pass. An array is defined on the GPU for a statement S iff this is also the case for its immediate predecessors in the control flow graph and if the array is not used or defined by the host, *i.e.* is not in the set $\mathcal{R}(I)$ or $\mathcal{W}(I)$ computed by PIPS:

$$\mathcal{D}_A^<(S) = \left(\bigcap_{S' \in \text{pred}(S)} \mathcal{D}_A^<(S') \right) - \mathcal{R}(S) - \mathcal{W}(S) \tag{1}$$

The initialization is done at the first kernel call site S_k with the arrays defined by the kernel k and used later ($\mathcal{OUT}(S_k)$). The following equation is used at each kernel call site:

$$\mathcal{D}_A^<(S_k) = \mathcal{OUT}(S_k) \bigcup \left(\bigcap_{S' \in \text{pred}(S_k)} \mathcal{D}_A^<(S') \right) \tag{2}$$

A backward pass is then performed in order to build $\mathcal{U}_A^>$. For a statement S, an array has its next use on the accelerator iff this is also the case for all statements immediately following in the control flow graph, and if it is not defined by S.

$$\mathcal{U}_A^>(S) = \left(\bigcup_{S' \in \text{succ}(S)} \mathcal{U}_A^>(S') \right) - \mathcal{W}(S) \tag{3}$$

Just as $\mathcal{D}_A^<$, $\mathcal{U}_A^>$ is initially empty and is initialized at kernel call sites with the arrays necessary to run the kernel, $\mathcal{IN}(S_k)$, and the arrays defined by the kernel, $\mathcal{W}(S_k)$. These defined arrays have to be transferred to the GPU if we cannot established that they are entirely written by the kernel. Otherwise, we might overwrite still-valid data when copying back the array from the GPU after kernel execution:

[1] PIPS uses a hierarchical control flow graph [17,1] to preserve as much as possible of the AST. However, to simplify the presentation of the analyses, we assume that a CFG is available.

$$\mathcal{U}_A^{>}(S_k) = \mathcal{IN}(S_k) \bigcup \mathcal{W}(S_k) \bigcup \left(\bigcup_{S' \in \text{succ}(S_k)} \mathcal{U}_A^{>}(S') \right) \quad (4)$$

An array must be transferred from the accelerator to the host after a statement S iff its last definition is in a kernel and if it is not the case for at least one of the immediately following statements:

$$\mathcal{T}_{A \to H}(S) = \mathcal{D}_A^{<}(S) - \bigcap_{S' \in \text{succ}(S)} \mathcal{D}_A^{<}(S') \quad (5)$$

This set is used to generate a copy operation at the latest possible location.

An array must be transferred from the host to the accelerator if its next use is on the accelerator. To perform the communication *at the earliest*, we place its launch immediately after the statement that defines it, *i.e.* the statement whose $\mathcal{W}(S)$ set contains it.

$$\mathcal{T}_{H \to A}(S) = \mathcal{W}(S) \bigcap \left(\bigcup_{S' \in \text{succ}(S)} \mathcal{U}_A^{>}(S') \right) \quad (6)$$

4.3 Interprocedural Extension

Kernel calls are potentially localized deep in the call graph. Consequently, a reuse between kernels requires interprocedural analysis. Function `iteration` (Fig. 1c) illustrates this situation, each step corresponds to one or more kernel executions.

Our approach is to perform a backward analysis on the call graph. For each function f, *summary* sets $\overline{\mathcal{D}_A^{<}}(f)$ and $\overline{\mathcal{U}_A^{>}}(f)$ are computed. They summarize information about the formal parameters for the function. These sets can be viewed as contracts. They specify a data mapping that the call site must conform to. All arrays present in $\overline{\mathcal{U}_A^{>}}(f)$ must be transferred to the GPU before the call, and all arrays defined in $\overline{\mathcal{D}_A^{<}}(f)$ must be transferred back from the GPU before any use on the host. These sets are required in the computation of $\mathcal{D}_A^{<}$ and $\mathcal{U}_A^{>}$ when a call site is encountered. Indeed at a call site c for a function f, each argument of the call that corresponds to a formal parameter present in $\overline{\mathcal{U}_A^{>}}$ must be transferred to the GPU before the call, because we know that the first use in the called function occurs in a kernel. Similarly, an argument that is present in $\overline{\mathcal{D}_A^{<}}$ has been defined in a kernel during the call and not already transferred back when the call ends. This transfer can be scheduled later, but before any use on the host.

Equations 1 and 3 are modified for call site by adding a translation operator, $\text{trans}_{\text{f} \to \text{c}}$, between arguments and formal parameters:

$$\mathcal{D}_A^{<}(c) = \left(\text{trans}_{\text{f} \to \text{c}}(\overline{\mathcal{D}_A^{<}}(f)) \bigcup \left(\bigcap_{S' \in \text{pred}(c)} \mathcal{D}_A^{<}(S') \right) \right) - \mathcal{R}(c) - \mathcal{W}(c) \quad (7)$$

$$\mathcal{U}_A^{>}(c) = \left(\text{trans}_{\text{f} \to \text{c}}(\overline{\mathcal{U}_A^{>}}(f)) \bigcup \left(\bigcup_{S' \in \text{succ}(c)} \mathcal{U}_A^{>}(S') \right) \right) - \mathcal{W}(c) \quad (8)$$

On the code Fig. 4, we observe, comparing the result of the interprocedural optimized code with the very local approach (Fig. 3), that all communications and memory management (allocation/deallocation) have been eliminated from the main loop.

```
void discretization(coord pos[NP][NP][NP],
                    int data[NP][NP][NP]) {
  //generated variable
  coord *pos0 = P4A_runtime_resolve(pos,NP*NP*NP*sizeof(coord));
  int *data0 = P4A_runtime_resolve(pos,NP*NP*NP*sizeof(int));
  // Call kernel
  P4A_call_accel_kernel_2d(discretization_kernel,
                           NP, NP, *pos0, *data0);
}
int main(int argc, char *argv[]) {
  /* Read data from input files */
  init_data(argv[1], ....);
  P4A_runtime_copy_to_accel(pos, ...*sizeof(...));
  /* Main temporal moop */
  for(t = 0; t < T; t+=DT)
    iteration(...);
  /* Output results to a file */
  P4A_runtime_copy_from_accel(pos, ...*sizeof(...));
  write_data(argv[2],....);
}
```

Fig. 4. Simplified code for functions `discretization` and `main` after interprocedural communication optimization

4.4 Runtime Library

Our compiler PAR4ALL includes a lightweight runtime library that allows to abstract from the target (OPENCL and CUDA). It also supports common functions such as memory allocation at kernel call and memory transfer sites. It maintains a hash table that maps host addresses to GPU addresses. This allows flexibility in the handling of the memory allocations. Using it, the user call sites and his function signatures can be preserved.

The memory management in the runtime does not free the GPU buffers immediately after they have been used, but preserves them as long as there is enough memory on the GPU. When a kernel execution requires more memory than is available, the runtime frees some buffers. The policy used for selecting a buffer to free can be the same as for cache and virtual memory management, for instance LRU or LFU.

Notice in Fig. 4 the calls to the runtime that retrieve addresses in the GPU memory space for arrays `pos` and `data`.

5 Experiments

We ran several experiments using the Polybench Suite, Rodinia, and *Stars-PM*. The Rodinia's tests had to be partially rewritten to match PAR4ALL coding guidelines. Specifically, we performed a one-for-one replacement of C89 array definitions and accesses with C99 variable length array constructs. This was necessary for the automatic parallelization process of Par4All to succeed and provide good input to our communication optimization later in the compilation chain. All benchmarks presented here, including *Stars-PM*, are available with the current development version of Par4All [15].

5.1 Metric

The first question is: what should we measure? While speedups in terms of CPU and wall clock time are most important to users if not to administrators, many parameters impact the results obtained. Let us consider for example the popular *hotspot* benchmark [16]. Its execution time depends on two parameters: the matrix size and the number of time steps. In the general context of GPU the matrix size should be large enough to fully load the GPU. In the context of this paper the time step parameter is at least as important since we move data transfers out of the time step loop. Figure 6 shows how *hotspot* is affected by the number of time step iterations and approaches an asymptote, acceleration ranges from 1.4 to 14. The single speedup metric is not enough to properly evaluate our scheme.

We believe that a more objective measurement for evaluating our approach is the number of communications removed and the comparison with a scheme written by an expert programmer. Focusing on the speedup would also emphasize the parallelizer capabilities.

We propose to count the number of memory transfers generated for each version of the code. When the communication occurs in a loop this metric is parametrized by the surrounding loop iteration space. For instance many benchmarks are parametrized with a number of time steps t, thus if 3 memory transfers are present in the time step loop, and 2 are outside of the loop, the number of communication will be expressed as $3 \times t + 2$. Sometimes a loop that iterate over a matrix dimension cannot be parallelized, either intrinsically or because of limited capacity of the compiler. Memory transfers in such loop have a huge performance impact. In this case we use n to emphasize the difference with the time step iteration space.

5.2 Measurements

Figure 7 shows our results on 20 benchmarks from Polybench suite, 3 from Rodinia, and the application *Stars-PM* (§ 3). Measurements where performed on a machine with two Xeon Westmere X5670 (12 cores at 2.93 GHz) and a NVIDIA GPU Tesla C2050. The OpenMP versions used for the experiments are generated automatically by the parallelizer and are not manually optimized.

Kernels are exactly the same for the two automatically generated versions using Par4All. We used the NVIDIA CUDA SDK 4.0, and GCC 4.4.5 with `-O3`.

For Polybench, we forced Par4All to ignore array initializations because they are both easily parallelized and occur before any of the timed algorithms. Our normal optimizer configuration would thus have "optimized away" the initial data transfer to the GPU within the timed code. The measurements in Fig. 7 includes all communications.

Some of the Polybench test cases use default sizes that are too small to amortize even the initial data transfer to the accelerator. Following Jablin *et al.* [18], we adapted the input size to match capabilities of the GPU.

For the cosmological simulation, the communication optimization speeds up execution by a factor of 14 compared to the version without our optimization, and 52 compared to the sequential code.

We include results for HMPP and PGI. Only basic directives were added by hand. We didn't use more advanced options of the directives, thus the compiler doesn't have any hints on how to optimize communications.

The geometric mean over all test cases shows that this optimization improves by a 4 to 5 factor over PAR4ALL, PGI and HMPP naïve versions.

When counting the number of memory transfers, the optimized code performs very close to a hand written mapping. One noticeable exception is *gramschmidt*. Communications cannot be moved out of any loop due to data dependencies introduced by some sequential code. Some aggressive transformations in the compiler might help by accepting a slower generated code but allowing data to stay on the GPU. This would still be valuable if the slowdown is significantly lower than the communication overhead. The difficulty for the compiler is to evaluate the slowdown and to attempt parallelization only if optimized communications lead to a net performance increase. This is beyond our current capabilities.

5.3 Comparison with Respect to a Fully Dynamic Approach

While our aim has been to resolve the issue of communication optimization at compile time, other approaches are addressing it entirely at runtime. This is the case for the *StarPU* [4] library.

We took an example included with the PAR4ALL distribution and rewrote it using *StarPU* in order to evaluate the overhead of the dynamic management of communication compared to our static scheme. This example performs 400 iterations of a simple *Jacobi* scheme on a 500×500 pixels picture loaded from a file and stores the result in another file.

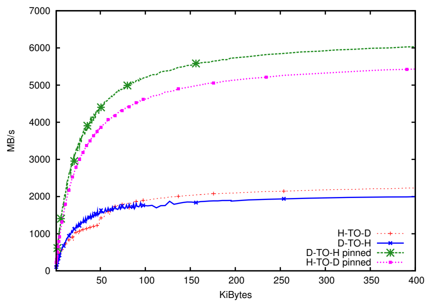


Fig. 5. Bandwidth for memory transfers over the PCI-express bus by data size

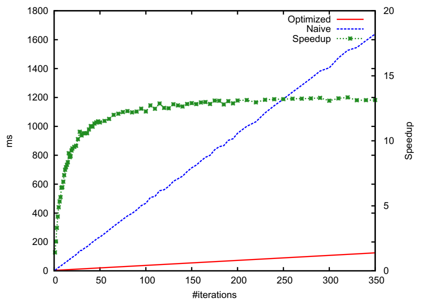
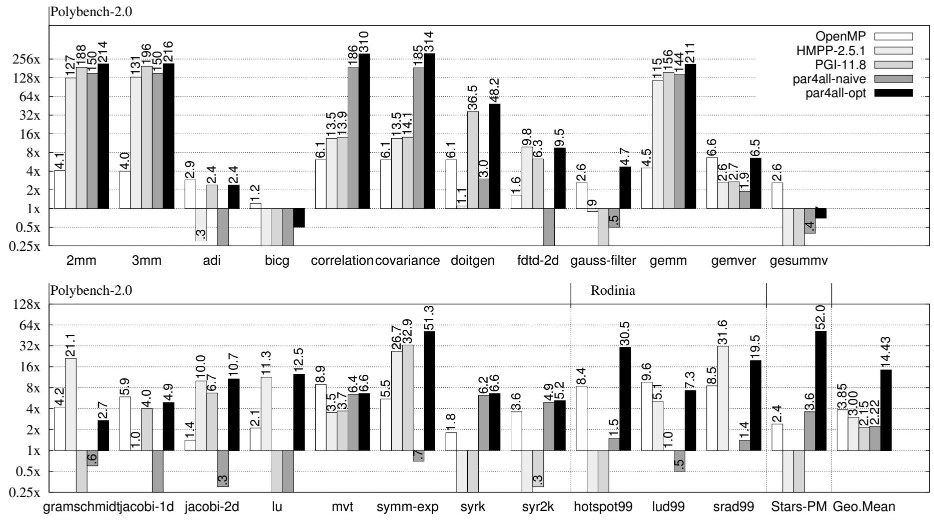


Fig. 6. Execution times and speedups for versions of hotspot on GPU, with different iterations counts



	2mm	3mm	adi	atax	bicg	correlation	covariance	doitgen	fdtd-2d	gauss-filter	gemm	gemver	gesummv	gramschmidt	jacobi-1d	jacobi-2d	lu	mvt	symm-exp	syrk	syr2k	hotspot99	lud99	srad99	Stars-PM
Naïve	8	12	15t +9nt	6+ 4n	6+ 4n	18	15	5	13t	7	4	17	7	9n	6t	6t	4n	8	6n	5	6	5t	4n	24t	25t
Opt.	5	6	4	3	5	2	2	3	6	4	4	10	3	4n +1	3	3	2	7	4	3	4	3	2	t	3
Hand	4	5	4	3	5	2	2	3	6	4	4	10	3	3	3	3	2	7	4	3	4	3	2	t	3

Fig. 7. Speedup relative to naïve sequential version for an OpenMP version, a version with basic PGI and HMPP directives, a naïve CUDA version, and an optimized CUDA version, all automatically generated from the naïve sequential code. Tables show the number of memory transfers.

We were able to confirm that *StarPU* removed all spurious communications, just as our static scheme does. The manual rewrite using *StarPU* and a GPU with CUDA offers a 4.7 speedup over the sequential version. This is nearly 3 times slower than our optimized scheme, which provides a 12.8 acceleration.

Although *StarPU* is a library that has capabilities ranging far beyond the issue of optimizing communications, the overhead we measured confirmed that our static approach is relevant.

6 Related Work

Among the compilers that we evaluated § 2.1, none implements such an automatic optimization. While Lee *et al.* address this issue [20, §.4.2.3], their work is limited to liveness of data and thus quite similar to our unoptimized scheme.

Our proposal is independent of the parallelizing scheme involved, and is applicable to systems that transform OpenMP in CUDA or OpenCL like OMPCuda [21] or OpenMP to GPU [20]. It's also relevant for directives-based compiler, such as Jcuda and *hi*CUDA [14]. It would also complete the work done on OpenMPC [19] by not only removing useless communications but moving them up in the call graph. Finally it would free the programmer of the task of adding directives to manage data movements in HMPP [5] and PGI Accelerator [23].

In a recent paper [18], Jablin *et al.* introduce CGCM, a system targeting exactly the same issue. CGCM, just like our scheme is focused on transferring full allocation units. While our granularity is the array, CGCM is coarser and considers a structure of arrays as a single allocation unit. While our decision process is fully static, CGCM takes decisions dynamically. It relies on a complete runtime to handle general pointers to the middle of any heap-allocated structure, which we do not support at this time. We obtain similar overall results, and used the same input sizes. Jablin *et al.* measured a less-than 8 geometric mean speedup vs. ours of more than 14. However, a direct comparison of our measurement is hazardous. We used GCC while Jablin *et al.* used *Clang*, and we made our measurements on a Xeon Westmere while he uses an older Core2Quad Kentsfield. He optimzed the whole program and measured wall clock time while we prevent optimization accross initialization functions and excluded them from our measures. Finally, he used a LLVM PTX backend for GPU code generation, while we used NVIDIA NVCC compilation chain. The overhead introduced by the runtime system in CGCM is thus impossible to evaluate.

7 Conclusion

With the increasing use of hardware accelerators, automatic or semi-automatic transformations assisted by directives take on an ever greater importance.

We have shown that the communication impact is critical when targeting hardware accelerators for massively parallel code like numerical simulations. Optimizing data movements is thus a key to high performance.

We introduced an optimization scheme that addresses this issue, and we implemented it in PIPS and Par4All.

We have experimented and validated our approach on 20 benchmarks of the Polybench 2.0 suite, 3 from Rodinia, and on a real numerical simulation code. The geometric mean for our optimization is over 14, while a naïve parallelization using CUDA achieves 2.22, and the OpenMP loop parallelization provides 3.9. While some benchmarks are not representative of a whole application, we measured on a real simulation an acceleration of 12 compared to a naïve parallelization and 8 compared to an OpenMP version on two 6-core processors. We found that our scheme performs very close to a hand written mapping.

We plan to improve the cache management in the runtime. We can go further than classic cache management algorithms because, unlike hardware cache, our runtime is software managed and can be dynamically controlled. Data flow analyses provide knowledge on the potential future execution of the program.

This can be used in metrics to choose the next buffer to free from the cache. Cheap computations unlikely to be used again should be chosen first. Costly results that are certain to be used should be freed last.

The execution times measured with multi-core processors show that attention should be paid to work sharing between hosts and accelerators rather than keeping the host idle during the completion of a kernel. Multi-core and multi-GPU configurations are another track to explore, with new requirements to determine optimal array region transfers and computation localization.

Acknowledgments. We are grateful to Béatrice Creusillet, Pierre Jouvelot, and Eugene Ressler for their numerous comments and suggestions which helped us improve our presentation, to Dominique Aubert who let us use his cosmological simulation, and to Thomas Jablin for his collaboration and comments.

References

1. Amini, M., Ancourt, C., Coelho, F., Creusillet, B., Guelton, S., Irigoin, F., Jouvelot, P., Keryell, R., Villalon, P.: PIPS is not (just) polyhedral software. In: 1st International Workshop on Polyhedral Compilation Techniques, Impact (in Conjunction with CGO 2011) (April 2011)
2. Ancourt, C., Coelho, F., Irigoin, F., Keryell, R.: A linear algebra framework for static High Performance Fortran code distribution. Scientific Programming 6(1), 3–27 (1997)
3. Aubert, D., Amini, M., David, R.: A Particle-Mesh Integrator for Galactic Dynamics Powered by GPGPUs. In: Allen, G., Nabrzyski, J., Seidel, E., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2009, Part I. LNCS, vol. 5544, pp. 874–883. Springer, Heidelberg (2009)
4. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience 23, 187–198 (2011); Special Issue: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 863–874. Springer, Heidelberg (2009)
5. Bodin, F., Bihan, S.: Heterogeneous multicore parallel programming for graphics processing units. Sci. Program. 17, 325–336 (2009)
6. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: IEEE International Symposium on Workload Characterization (2009)
7. Chen, Y., Cui, X., Mei, H.: Large-scale FFT on GPU clusters. In: 24th ACM International Conference on Supercomputing, ICS 2010 (2010)
8. Creusillet, B., Irigoin, F.: Interprocedural array region analyses. Int. J. Parallel Program. 24(6), 513–546 (1996)
9. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (2008)
10. Fang, W., He, B., Luo, Q.: Database compression on graphics processors. Proc. VLDB Endow. 3, 670–680 (2010)

11. Feautrier, P.: Parametric integer programming. RAIRO Recherche Opérationnelle 22 (1988)
12. Gerndt, H.M., Zima, H.P.: Optimizing Communication in SUPERB. In: Burkhart, H. (ed.) CONPAR 1990 and VAPP 1990. LNCS, vol. 457, pp. 300–311. Springer, Heidelberg (1990)
13. Gong, C., Gupta, R., Melhem, R.: Compilation techniques for optimizing communication on distributed-memory systems. In: ICPP 1993 (1993)
14. Han, T.D., Abdelrahman, T.S.: hiCUDA: a high-level directive-based language for GPU programming. In: Proceedings of GPGPU-2. ACM (2009)
15. HPC Project. Par4All automatic parallelization, `http://www.par4all.org`
16. Huang, W., Ghosh, S., Velusamy, S., Sankaranarayanan, K., Skadron, K., Stan, M.R.: Hotspot: acompact thermal modeling methodology for early-stage VLSI design. IEEE Trans. Very Large Scale Integr. Syst. (May 2006)
17. Irigoin, F., Jouvelot, P., Triolet, R.: Semantical interprocedural parallelization: an overview of the PIPS project. In: ICS 1991, pp. 244–251 (1991)
18. Jablin, T.B., Prabhu, P., Jablin, J.A., Johnson, N.P., Beard, S.R., August, D.I.: Automatic CPU-GPU communication management and optimization. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 142–151. ACM, New York (2011)
19. Lee, S., Eigenmann, R.: OpenMPC: Extended OpenMP programming and tuning for GPUs. In: SC 2010, pp. 1–11 (2010)
20. Lee, S., Min, S.-J., Eigenmann, R.: OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In: PPoPP (2009)
21. Ohshima, S., Hirasawa, S., Honda, H.: OMPCUDA: OpenMP Execution Framework for CUDA Based on Omni OpenMP Compiler. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 161–173. Springer, Heidelberg (2010)
22. Pouchet, L.-N.: The Polyhedral Benchmark suite 2.0 (March 2011)
23. Wolfe, M.: Implementing the PGI accelerator model. In: GPGPU (2010)
24. Yan, Y., Grossman, M., Sarkar, V.: JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 887–899. Springer, Heidelberg (2009)