HAL
open science

An Hybrid Data Transfer Optimization Technique for GPGPU

Eric Petit, François Bodin, Romain Dolbeau

To cite this version:

Eric Petit, François Bodin, Romain Dolbeau. An Hybrid Data Transfer Optimization Technique for GPGPU. Compilers for Parallel Computers workshop (CPC2007), 2007, Lisbon, Portugal. hal-00418717

HAL Id: hal-00418717

https://hal.science/hal-00418717v1

Submitted on 21 Sep 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

An Hybrid Data Transfer Optimization Technique for GPGPU*

Eric Petit, Francois Bodin† and Romain Dolbeau‡
eric.petit, francois.bodin@irisa.fr, romain.dolbeau@caps-entreprise.com

25/04/2007

Abstract

Graphical Processing Units (GPU) can provide tremendous computing power. Current NVidia and ATI hardware display a peak performance of hundreds of gigaflops. However, because of the data transfer speed between CPU and GPU is limited, those devices are difficult to use to accelerate numerical applications. In this paper we propose a software hybrid technique for automatically optimizing data transfer based on static and dynamic information on data accesses.

1 Introduction

Because of their high potential computing power, the use of graphical processing units (GPUs) looks very attractive to speed up programs. Furthermore new programming environments such as Cuda [5], RapidMind [6], PeakStream [7] or CTM [10] have made the use of GPUs for general purpose programming easier and more efficient (but not portable). These devices achieve high performance with highly parallel microarchitecture and fast internal memories. This is illustrated in Figure 1. Data transfers are implemented using the PCI express bus or in more coupled systems via the Hypertransport channel [8].

Currently, the use of GPUs as hardware accelerators to speedup applications is limited by the communication involved between the main memory and the GPU memory. The communication overhead is so high that it is not worthwhile to remotely execute a computation intensive kernels on a GPU. The issue of optimizing the communication is usually left to the programmer in an error prone process. The programmer must decide when to prefetch, update or download the remote data while ensuring that the GPU data are up to date when the remote procedure call is performed on the GPU.

In this paper, we study an automatic technique for software advance data uploads insertion for kernel that are remotely executed on a GPU. The proposed technique relies on an hybrid approach that mixes speculative data collected via the thread analysis environment Astex [1] and usual static program analysis. We apply this technique to C programs and deal with pointer aliasing issues.

This technique is implemented using an API denoted Heterogeneous extension to OpenMp [2] (HOMP). HOMP allows to specify remote data upload and remote procedure call to GPU implemented functions in a portable fashion, keeping the user program independent from the hardware accelerator used. We assume that the computation intensive kernels remotely executed on the GPU has been provided via specialized code generation [11, 1] or hand programmed using the GPU specific programming environment.

The paper is organized as follow. In Section 2 we present the programming interface based on pragma HOMP. This API described, via pragma, which functions can be executed on the GPU. These functions exist in two versions. The regular C versions with the pragma and a GPU versions that can be remotely called via HOMP runtime. Starting from the application program enhanced with pragma we use Astex, described in Section 3, a speculative thread extraction environment to compute speculative data on the

*This work is partially funded by the SARC, FAME2 and PARA projects
†Irisa, campus universitaire de Beaulieu, 35042 Rennes, France
‡Caps entreprise, Immeuble Gallium, 80, avenue des Buttes de Coesmes, 35700 Rennes, France

remote functions parameters. Using these speculated data, we show in Section 4 how they can be used to automatically optimize data transfers. In the next section, we report preliminary results obtained using this technique.

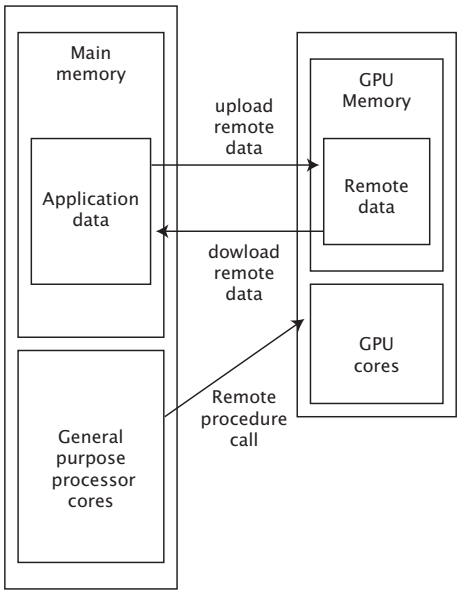

Figure 1: Use of GPU as an hardware accelerator for general purpose applications.

2 HOMP API

Homp [2] is an extension to OpenMP [9] pragma to handle hardware accelerators. It allows using OpenMP to exploit general purpose multicores while simultaneously using GPUs or other hardware accelerators. The use of pragma helps to keep the program portable. The Homp pragma are translated into calls to the Homp runtime that deals with resource allocation, data transfers and the remote procedure calls on the GPU. This runtime is compatible with OpenMP runtime.

Each kernel to be executed on the hardware accelerator is implemented as a codelet. A codelet is a **pure function** (i.e. The function always evaluates the same result value given the same argument value(s). It has no side effects, no I/O). It should also be suitable for the target hardware device capabilities (e.g. single precision float data for GPU, etc.). Because argument values must be transfered to the hardware accelerator there are **constraints on the codelet arguments**:

1. Arguments are scalars, arrays or pointers.

2. Array and pointer arguments are followed by an integer argument that gives the size of each dimension of the array or pointer argument. For instance, an argument `A[][]` is followed by two integer arguments that give the size of the first and second dimensions of A.

The `codelet` directive declares a function (mark 1 in Figure 2) that is to be performed by an available hardware accelerator. If there are more than one hardware target, multiple `codelet` directives can be added. The implementation of the codelet for the GPU is automatically or by hand produced. Data transfers and synchronizations instructions does not have to be inserted close to the call to a codelet. A `callsite` directive specifies where to use the hardware implementation of the codelet in a program point (mark 2 in Figure 2).

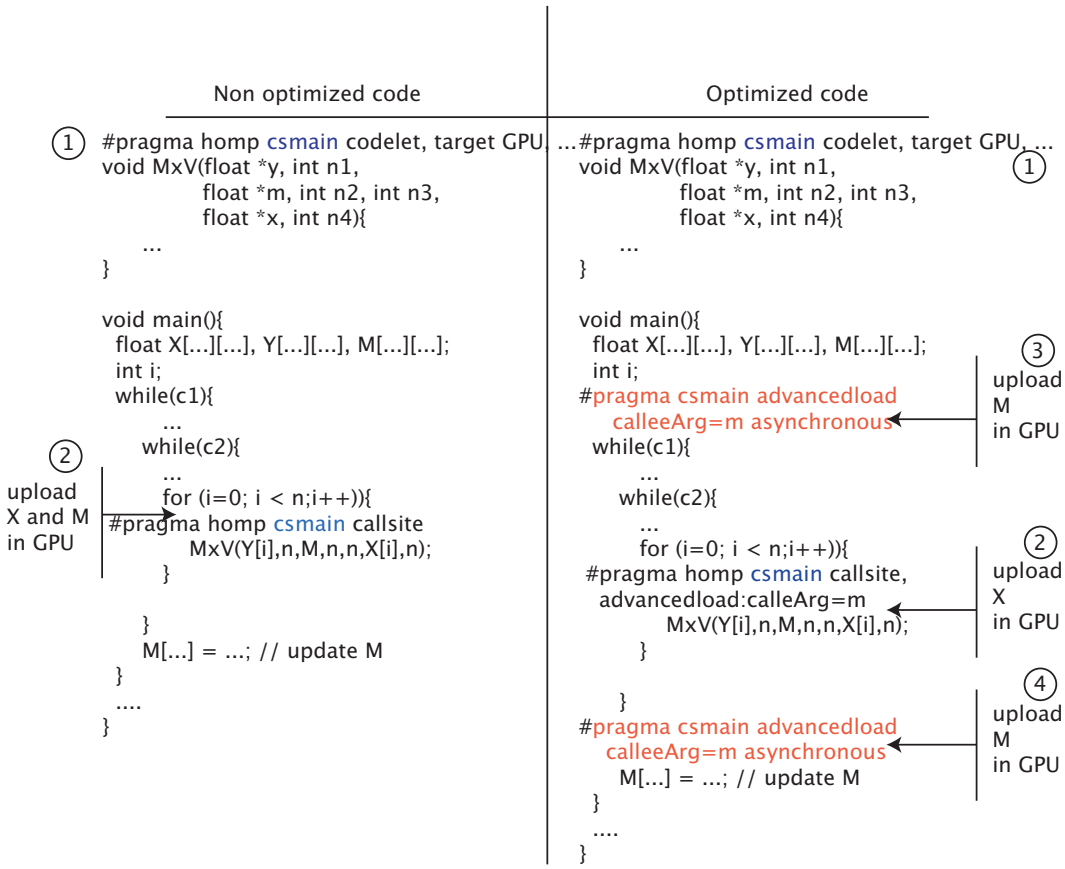

Figure 2: HOMP pragma example.

A label (`csmain` in blue in Figure 2)) is given by the `codelet` and the `callsite` directives to identify which directives apply to which call site.

The `advancedload` directive shown on the marks 3 and 4 of Figure 2 is used to specify that the remote data can be uploaded at the program point where the directive is inserted. If an advancedload is added the data is not uploaded when reaching the codelet call. In this example, the matrix `M` is uploaded on the GPU only when leaving the inner loops which are assumed not to modify matrix `M`.

In a similar way to the advancedload, getting data back from the hardware accelerator can be optimized. The delegated-store directive aims at this purpose. It is placed into the code where the data transfer is completed for one of the outputs of the codelet. In this paper we do not address the issue for delegated-store and assume that the result is needed as soon as the codelet execution returns. However, the technique presented in this paper can also be used to delay the synchronization with the end of the codelet execution. The reader interested in more details about HOMP can refer to [2].

In the remainder of the paper, we present a technique to automatically insert the advancedload directives for all input parameters of a codelet.

3 Computing Speculative Data using ASTEX

ASTEX is a tool that computes "speculative threads" in C programs, based on execution traces. ASTEX

does successive steps composed of instrumentation, execution, and profile analysis. In this study, we use the memory access model built by ASTEX. This model has two main components:

1. Abstract memory sets describing data objects allocated by the program.
2. Memory access descriptions indicating to which abstract memory sets an access was made, and the range of acceded data for each abstract memory set.

These data are collected on a program by instrumenting the source code and running the application for some input data. As a consequence, when optimizing the code we have to treat those data as speculative [1]. In the remainder of this section we define these two components. The reader interested in more details can refer to [1].

3.1 ASTEX Program Working Set and Access Description

A memory block b is defined by a memory address and a size denoted by a couple *(@,S)*. A block is created in the program whenever a new object is allocated.

To identify blocks in the program we use *abstract memory sets* (AMS) that are defined by a *program creation point*, a *content type*, *a free program point* and a set of memory blocks. There is no overlap between memory blocks and a block can only belong to one AMS. The *content type* can be *value* or *address* or both.

For global variables the *program creation point* is defined as the beginning of the main program, the *free program point* is the end of the program. For dynamically allocated memory, the malloc statement is the *program creation point*. If a unique free statement can be identified, it is the *free program point* otherwise the end of the program is used.

For stack variables the *program creation point* and *a free program point* are respectively the function entry and exit points for a given call site. The blocks corresponding to stack variables are also stacked in the abstract memory sets.

The set of *abstract memory sets* available at a program point during the execution defines the *program working set*. An *abstract memory set* which all memory blocks have been freed is removed from the *program working set*.

3.2 ASTEX Memory Access Description

An *access* to memory, denoted *(@,R|W, s)*, is defined by the address (@), the access mode (read or write, $R|W$), and the size of the element (s). An *abstract reference* (AR), for a program statement is constructed using the real accesses (obtained by instrumentation of the source code) . For each access, the AMS and the corresponding memory block are determined. According to the block, the minimum and maximum offsets are computed for the set. There is a unique *abstract reference* for each memory access expression in the program. The abstract reference is defined by a tuple *(id, {(abstract memory set, block, offset min, offset max, R|W) .... })* where the *id* is the identifier of the expression.

In the Section 4, we show how, using these speculative data, we insert the `advance load directive`.

4 Data Transfer Optimization

Data transfer optimization aims at prefetching data as soon as possible and at avoiding to load data on the GPU that are already up to date. Furthermore, updating multiple times the remote data before the codelet is called must be avoided, to save memory bandwidth. This is a difficult task since data modifications (taking into account pointer aliasing) and control flow must considered.

The technique we propose relies on an hybrid, pure software, scheme that mixes static analysis of data accesses and speculative information provided by ASTEX. The proposed scheme mixes two policies:

[1] We assume here that the input data have a low impact on code behavior

LATE upload policy: This policy is used when the parameter upload has a strong chance to be redundant (follows at runtime by another one).

ASAP upload policy: This policy aims at moving the upload point at the last more likely modification of the remote data. It is used only if it does not increase the number of executed uploads.

To illustrate these policies, let us consider the example in Figure 3. We assume here, for the sake of simplicity that the data structure corresponding to parameter `M` is unique[2]. The choices between LATE and ASAP policy for each program point from 1 to 5 in this example are the following:

1: This is a sure modification of parameter `M` on an infrequent path. To avoid multiple uploading of the data in **3**, a LATE upload policy is chosen. It should be noted that when possible the upload is performed asynchronously. The call to `doUpload()` is either implemented using HOMP pragma or by direct call to HOMP runtime.

2: In this case, there may be a modification of `M` (this could not be determined statically and ASTEX did not provide speculative information about it). Because this is an unfrequent path (uploading the parameter just before doing the remote procedure call won't have a large impact on performance) and this upload may be redundant, because of upload 4, a LATE upload policy is chosen. Furthermore, the LATE policy implementation reduces the runtime overhead. The `change(M)` condition is build using the ASTEX AMS (See Section 3.1).

3: This is a sure modification of parameter `M` on a frequent path and there is a no further upload on one of the path from 3 to the remote call, except 4 but with a very low probability as it is speculated `M` not acceded. Indeed we can perform the upload ASAP.

4: In this case, we have speculative information indicating that `M` is not modified here. But because this could not be proved statically, a LATE upload is inserted.

5: This is similar to case **3**.

The proposed algorithm works in three steps. The first step inserts uploads according to a classification of the memory accesses. The second step removed redundant uploads or moves them out of the loops. The last step converts some ASAP upload to LATE upload to limit potential runtime overhead taking into account paths probability in the program.

S	Static Status	D	Dynamic Status
1	Access to the parameter	1	access to the parameter
2	No access to the parameter	2	Did not see an access to the parameter
3	undetermined	3	no information

Table 1: Static and dynamic status, not all association are possible, (1,2) and (2,1) are impossible

The classification of the memory accesses is performed according to the static and dynamic information (see Section 3) collected on memory accesses for a given codelet parameter. Each access is characterized with a pair: $(Stat_status, Dyn_status)$. The values for $Stat_status$ and Dyn_status are listed in Table 1. The static data are collected using usual data flow analysis technique [13] For each possible pair, the first step of the proposed technique performs the following actions:

1. (1, 1 or 3): Static information indicates that the parameter is modified, an ASAP upload is inserted after the data access.

2. (2, 2 or 3): This access does not modify the parameter in any case. No action is performed.

[2]If this not the case, the call is duplicated and guards are inserted.

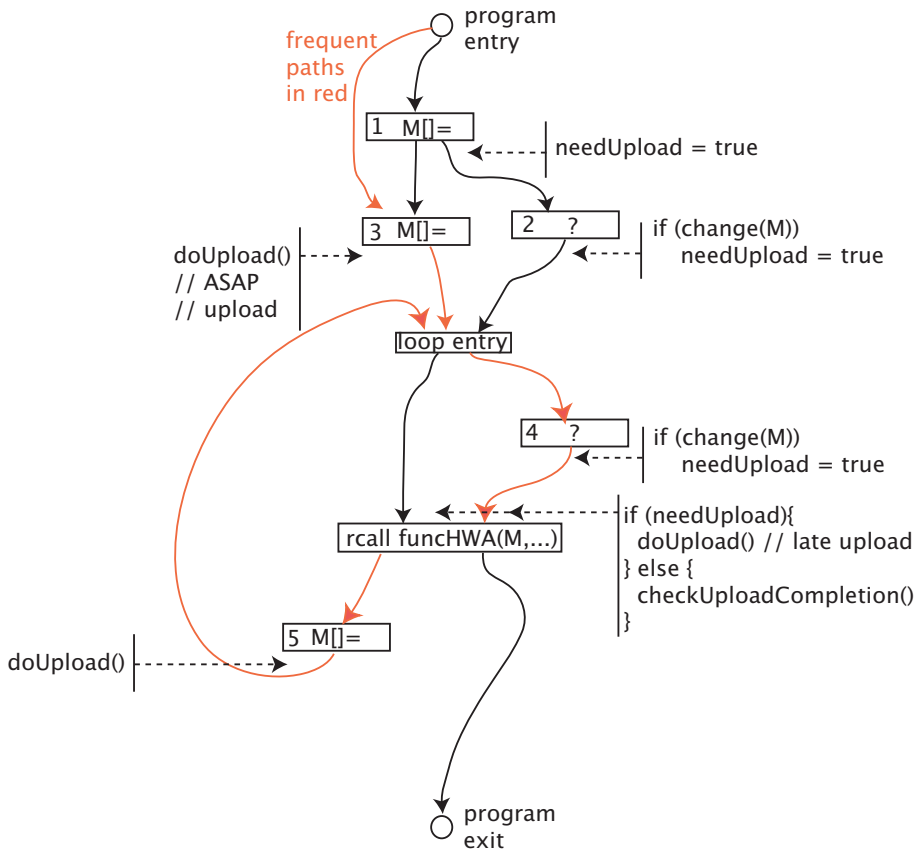

Figure 3: Upload strategy example.

3. (3,1): An ASAP upload with a dynamic check for the parameter is inserted, based on the speculative data.

4. (3,2): The dynamic data does not report a change of the codelet parameter. A LATE upload of the data with a dynamic check for the parameter access is inserted.

5. (3,3): We don't know anything about this access, a LATE upload of the data with a dynamic check for the parameter access is inserted.

6. (1,2) or (2,1): impossible case.

Once this initial step is performed, as illustrated in the left column of Figure 4, the second step moves uploads outside loops and removes LATE and ASAP uploads that are redundant due to existing non conditional ASAP uploads on the paths between the uploads to remove and the remote codelet call. This is illustrated in the middle column of Figure 4. ASAP upload X and Y are moved at the exit of the loops where they were.

The last step converts ASAP upload to LATE upload if there is a strong probability to encounter an ASAP upload on the paths to the remote codelet call. The probability of encountering an ASAP upload is obtained using profiling data such as presented in [12]. If the probability to have another upload at runtime before reaching the codelet call is above a threshold (or instance 80%), the ASAP upload is converted to a LATE upload. It should be noted that in these cases, the ASAP upload is not totally redundant since it

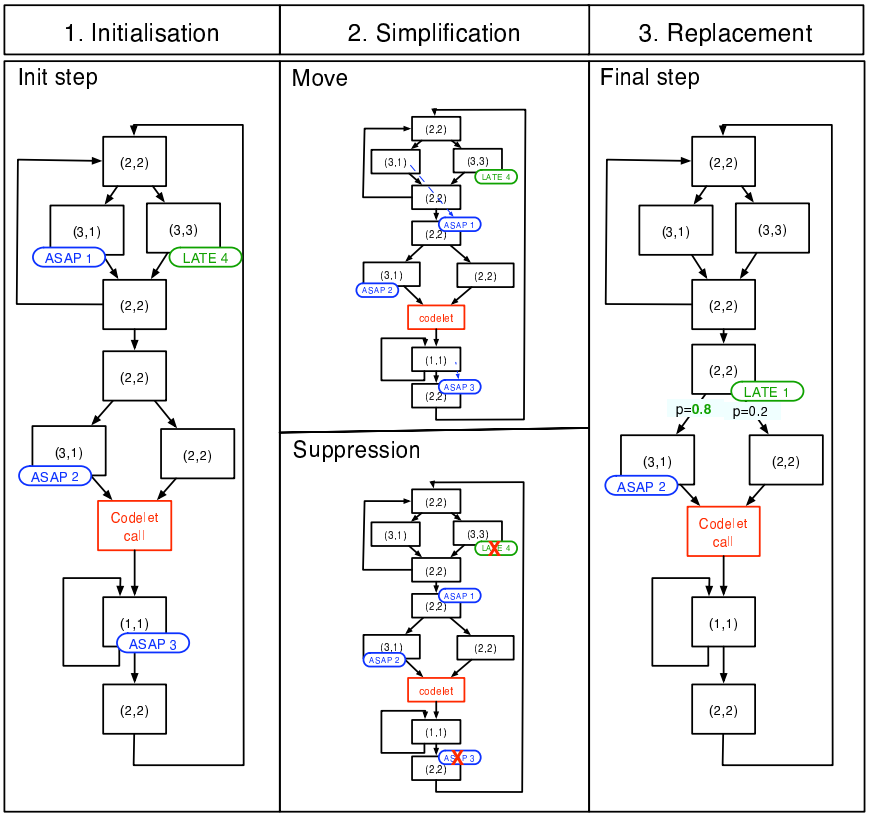

Figure 4: Upload point simplification and LATE upload insertion.

exists paths reaching the codelet call that may not upload the new value of the parameter. This step of the algorithm is illustrated in the right column of Figure 4.

5 Preliminary Experiments

To experiment the proposed strategy we use a simple benchmark based on a matrix times a vector kernel. The GPU used is an NVIDIA 8800 board. The host machine is a Pentium D, 2.80GHz machine. The connection between the GPU and the main processor is based on a PCIe bus. This benchmark is sketched in Figure 5. On this Figure, at mark 1, is the matrix times a vector kernel. The GPU kernel is implemented using CUDA [5]. At mark 2 is the main loop. At each iteration of this loop, the new vector is multiplied by the matrix. The matrix is updated when `cond1` or `cond2` is true. For each remote call to the kernel, the input and output vectors are respectively uploaded and downloaded. At mark 3, is the LATE upload of the matrix. At mark 5 is an ASAP upload of the matrix because `cond1` is frequently true. The update performed in the conditional guarded by `cond2` could not be resolved statically and no dynamic information is available, the algorithm inserts a LATE upload.

```
1  #pragma homp matvec codelet, output=outv, target GPU
   void matvec(int n, int m,float *inv, int N1,
                    float *inm, int N2,float *outv, int N3) {
     int i, j;
     ...
   }
   int main(int argc, char **argv) {
      ...
2     for (k = 0 ; k < iter ; k++) {
         ...
3        if (uploadmat) {
            #pragma homp matvec advancedload, calleeArg=inm
            uploadmat = 0;
         }
4        #pragma homp matvec callsite, advancedload:calleeArg=inm,
                                   asynchronous
         matvec(n, m, (inc+(k*n)), n, inm, n*m, (outv+(k*m)), m);
         if (cond2){
            for (i=0; i<m; i++){ // update the matrix
5            *pt= ...; pt++;
            }
            uploadmat = 1;
         }
         if (cond1) {
6           for (i=0; i<m; i++) { // update the matrix
                inm[i*n +m] = ...;
            }
            #pragma homp matvec advancedload, calleeArg=inm
            uploadmat = 0;
         }

      }
      ...
   }
```

Figure 5: Synthetic benchmark based on a matrix times a vector kernel.

The performance of the code is given in Table 2. The matrix is a 4096x4096 single precision floating point array. The code reference code (Pentium only column) is compiled with the Intel compiler `icc -O2 -march=pentium4 -mcpu=pentium4 -vec-report3` (vectorization was effective). When modifications of the matrix happen every 10 iterations (frequency 0.1 for the conditional `cond1`) there is a very marginal performance gain (6%). The best speedup achieved is 1.9 when there is no modification of the matrix. When modifications occur every 50 iterations (frequency 0.02 for the conditional `cond1`) the speedup is 1.7. This is a good result since the matrix time a vector does not have a very high computing density

Matrix update frequency with cond1	Matrix update frequency with cond2	Pentium only	GPU+Pentium
0.1	0.01	7.86 sec.	7.36 sec.
0.1	0.005	7.86 sec.	7.18 sec.
0.02	0.01	7.86 sec.	4.73 sec.
0.02	0.005	7.86 sec.	4.53 sec.
0.	0.	7.86 sec.	3.81 sec.

Table 2: Performance, in seconds, of the synthetic benchmark for 512 iterations of the outer loop.

compared to a matrix time a matrix kernel.

6 Conclusion

In this paper, we have presented preliminary work for automatically optimizing data transfers when using a GPU as an hardware accelerator for numerical applications. This is a key issue for achieving performance gain. We use HOMP and Cuda as implementation layers for communication and exploitation of the GPU. The proposed hybrid scheme can handle C code without requiring complex and unreliable pointer analysis thanks to the use of speculative data. To our knowledge, very few previous works on the usage of GPU in the context of general programming have considered the communication overhead issues. Preliminary performance results show that it is possible to get speedup with the proposed strategy. However, the current speed of the PCIe bus used to transfer the data between the CPU and the accelerator is still very slow and limits the potential of the approach for most applications. Current trends in the multicore technology let us believe that communication costs will be decreasing making the use of hardware accelerators efficient on a large range of programs.

References

[1] Eric Petit and Francois Bodin. ASTEX *project web site*. www.irisa.fr/CAPS/projects/astex/index.htm.

[2] Romain Dolbeau and Francois Bodin. HOMP *description documentation*. http://www.caps-entreprise.com.

[3] J. D. Collins, H. Wang, D. M. Tullsen, C. Huges, Y. Lee, D. Lavery and J. P. Shen Speculative Precomputation: Long-range Prefetching of Delinquent Loads. *ISCA 01*, ACM SIGARCH Computer Architecture News, 2001.

[4] A. D. Brown and T.C. Mowry and O. Krieger, Compiler-Based I/O Prefetching for Out-of-Core Applications. ACM Trans. Comput. Syst., vol. 19, New York, NY, USA, 2001

[5] NVIDIA devellopers site, CUDA homepage, http://developer.nvidia.com/object/cuda.html

[6] RapidMind Corp. home page, http://www.rapidmind.net/

[7] Peakstream Corp. home page, http://www.peakstreaminc.com/

[8] Hypertransport consortium home page, http://www.hypertransport.org/

[9] OpenMP web site: http://www.openmp.org/

[10] ATI CTM Guide, Technical Reference Manual, http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf

[11] Caps entreprise white paper, Build High Performance Libraries with Outstanding Stability, http://www.caps-entreprise.com/documentation/capstuner_libraries.pdf

[12] Thomas Ball and James R. Larus, Optimally profiling and tracing programs, *POPL '92* Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Albuquerque, New Mexico, United States, 1992

[13] Steven S. Muchnick Advanced Compiler Design and Implementation, Morgan Kaufmann, 1997