

# Partitioning Programs for Automatically Exploiting GPU\*

Eric Petit and Sebastien Matz and Francois Bodin  
epetit, smatz,bodin@irisa.fr  
IRISA-INRIA-University of Rennes 1  
Campus de Beaulieu, 35042 Rennes, FRANCE

30/09/2006

## Abstract

In this paper we explore the use of ASTEX, a C language partitioning tool, to detect parts of code that can potentially be speeded up by using Graphical Processing Units (GPU).

## 1 Introduction

Because of their high potential computing power the use of graphical processing units looks very attractive to speed up programs [10, 12, 4]. However, because of their idiosyncrasies they are difficult to program. Furthermore data transfers between the main memory to the GPU may strongly impact on the resulting performance. Until recently, most previous works have been considering porting algorithm on GPU by hand [3, 1, 7]. Other studies have been focusing on provided better programming tools for GPU [2, 9, 8]. However to our knowledge no works have been addressing partitioning C programs for GPU. A first step to automatically exploit GPU in the context of general programming is to be able to focus the effort on pieces of code that fit GPU constraints.

In this paper we report work-in-progress that proposes an automatic approach to detect parts of codes that can make advantage of GPU. Data transfers and locality are the main issues we take into account. Figure 1 summarizes the approach we have been taken. In blue, ASTEX implements a dynamic analysis that computes a *speculative thread* that contains a kernel that have the potential to bring speedup. This is not a simple hotspot detection since parallelism, data locality and data transfers have to be taken into account. The yellow part in Figure 1 is the generation of the code of the kernel for the GPU. This part is currently being developed and is not discussed in this paper. Our work differs from [6] which proposes to automatically seeks parallelisable code fragments and replaces them with code for a graphics co-processor by the underlying parallel model. Our approach is based on a speculative threads model.

In the remainder of the paper, Section 2, we give a brief overview of the GPU programming constraints that have to be taken into account. We illustrate GPU usage with a linear computation kernel. In Section 3 we describe ASTEX a tool that is able to compute the static and dynamic properties of code sequence to evaluate their fitness for GPU usage.

## 2 Graphical Processing Units Model

To achieve very high performance (e.g. 50 GigaFlops for an Nvidia 7900GT) GPUs rely on a parallel and pipeline structure. Figure 2 shows the general architecture of a GPU. In red are the two programmable units. The vertex processor that is usually in charge of computing the coordinates of vertices. A fixed

---

\*This work is partially funded by the PARA and the FAME2 ANR projects <http://www.agence-nationale-recherche.fr/> and SARC european project.

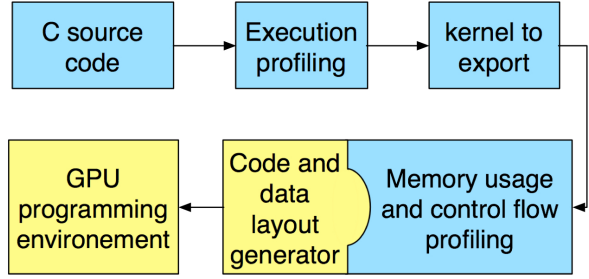


Figure 1: Pipeline from CPU implementation to GPU implementation.

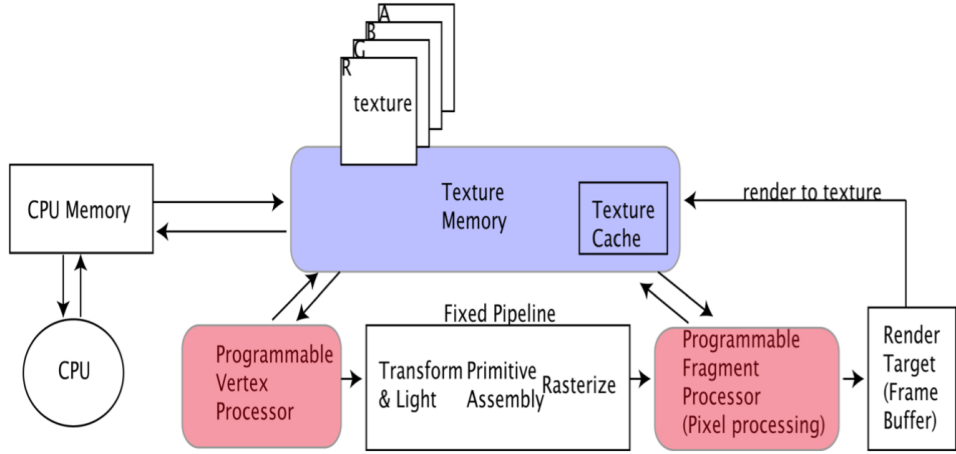


Figure 2: GPU architecture model.

pipeline converts set of vertices into image fragments. The fragment processor is in charge of computing fragments (coordinates and attributes values to display). This later concentrates most of the computing power of the GPU. It should be noted that all fragments are computed independently. The data are stored in 2D memories, texture. Access can address in parallel the 4 values, RGBA, of one fragment. Each value in memory is either an 8 bit, 16 bit or 32 bit floating point value. The processors have very limited capabilities for branches. The output values are written in a frame buffer. The content of the frame buffer can be loaded in a texture memory by a primitive called *render to texture*.

When implementing loop kernels on the GPU the vertex processor and the fixed pipeline are usually in charge of computing the loop iteration numbers. The fragment processor computes, using parallel units, the fragment values. Each individual computing unit of the fragment processor can compute in an SIMD fashion the 4 RGBA values. Data are usually loaded in the texture memory via a PCIx bus (a fast CPU-GPU PCI bus is about 4GB/s both ways).

To illustrate GPU programming issues let's consider the matrix-vector multiply like kernel given Figure 3. The arrays  $\mathbf{x}$  and  $\mathbf{y}$  are mapped in  $(N, N)$  2D texture while matrix  $\mathbf{M}$  is mapped in an  $(N^2, N^2)$  2D texture. This mapping is performed for a given value of  $N$  that must be known at GPU code generation time. For the NV QuadroFX 3450 GPU maximum value for  $N$  is 32. The shape and size of the texture used impact on the performance. The size of the two dimensions of a matrix should be equal and be a power of 2. Rectangular textures can be used but they usually incur performance penalty. To achieve maximum parallelism,  $\mathbf{y}[0]$ ,

```

static const float M[N*N][N*N] ;
void foo(float y[4][N*N], float x[4][N*N]){
  int i,j;
  for(i = 0 ; i < N*N ; i++) {
    for(j = 0 ; j < N*N ; j++) {
      y[0][i] += M[i][j] * x[0][j] ;
      y[1][i] += M[i][j] * x[1][j] ;
      y[2][i] += M[i][j] * x[2][j] ;
      y[3][i] += M[i][j] * x[3][j] ;
    }
  }
}

```

Figure 3: Kernel Example.

$y[1]$ ,  $y[2]$ ,  $y[3]$  are respectively mapped on the RGBA components. The same is done for array  $x$ . The outer loop is unrolled and jammed to generate more instruction level parallelism as well as more data locality. Specific optimizations, when generating the GPU code, can be performed to improve the usage of the GPU small cache memories [5]. After this transformation the inner loop is mapped on the GPU and the outer loop of the kernel is executed on the CPU. The matrix  $M$  is assumed to be loaded once on the GPU while arrays  $x$  and  $y$  are transferred back and forth at each iteration of the outer loop (i.e. index  $i$ ).

Table 1 shows the relative performance of the kernel example between a P4 3.6GHz CPU, using the GCC and Intel ICC compilers, and two NVidia GPU cards using the OpenGL compiler. Table 3 shows the Mflops obtained on each platform. The speed up is obtained for the highest problem size.

Table 2 shows the communication time from CPU to GPU (upload) and from GPU to CPU (download). Because of the constant overhead of the OpenGL loading and unloading primitives, communication can be a large part of the execution time. As a consequence, it is important to reduce data movements and, when possible, to prefetch data.

As illustrated by this example, a kernel code must have many properties to be mapped on the GPU and to provide speedup. We are currently making experiments to build a performance model for the GPU. In next section we show how kernels can be extracted from C programs and how all the properties relevant to GPU usage are obtained.

N	CPU GCC -O3	CPU ICC -O3	NV QuadroFX 3450 GPU	NV 7900GT GPU
8	1	0.97	0.11	0.12
16	1	0.95	0.53	0.63
32	1	1.09	2.41	3.53

Table 1: Relative speed-up compare to GCC

N	Upload time	Download time	percentage of comm. in the execution
8	0.035	0.092	30 %
16	0.041	0.092	10 %
32	0.080	0.103	3 %

Table 2: Time in millisecond for memory transfer for 8 vectors of size  $N*N$  and communication versus execution ratio in percentage.

N	CPU (GCC -O3)	NV QuadroFX 3450 GPU	NV 7900GT GPU
8	1396	159	171
16	1401	751	884
32	1130	2725	3993

Table 3: Mflops obtained on each processor including memory transfer on 32bit FP data.

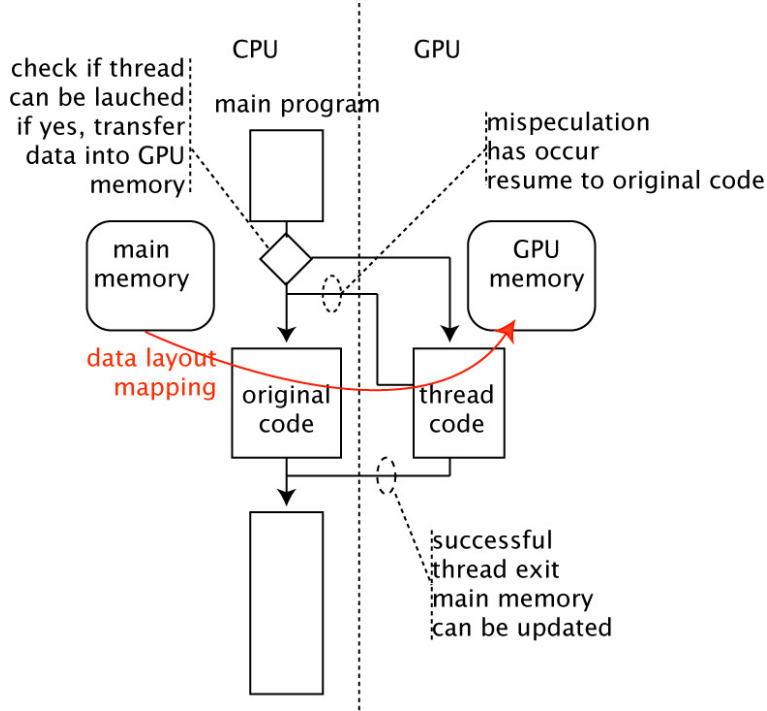


Figure 4: ASTEX thread model.

### 3 GPU Kernel Detection with ASTEX

ASTEX is a tool to compute "speculative threads" in C programs based on execution traces. The thread model is shown in Figure 4. In this model, speculation is performed on control flow and data layout:

1. At control flow level: The thread is assumed to correspond to a set of paths in the execution of the application program. If the execution of the thread leaves the assumed set of paths, then the speculation fails.
2. At data layout level: The data structures used in the application must be mapped onto the local memory of the GPU. If a data accessed by the thread is out of the speculated memory space, the thread returns with an error code. This is one of the key points of this study.

ASTEX build an executable C version of the threads so they can be tested against multiple data sets before deciding to porting them on a GPU. This is useful to check if speculation is allright with various input data set. ASTEX does successive steps composed of instrumentation, execution, and profile analysis. As shown in figure 5 the ASTEX process is divided in three main steps.

1. The first one finds the computation intensive parts. This step is based on trace analysis. The hot-paths are the most frequently taken sub-path in the control flow graph during program execution.

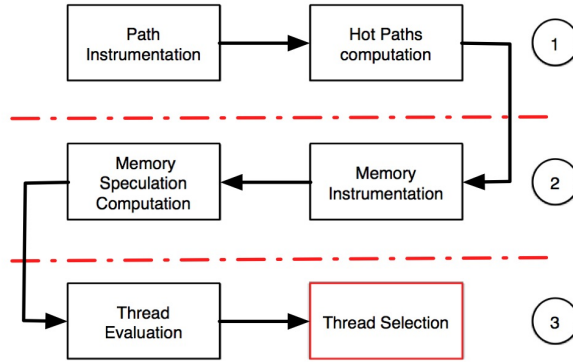


Figure 5: ASTEX partition pipeline steps.

2. The second step performs the memory instrumentation in order to get a precise characterization of memory transfers and accesses. After exercising the instrumented program against many input data, we compute a speculative memory usage model.
3. The last step, according to execution results, evaluates the characteristics of the potential threads.

To select the threads for GPU, static and dynamic criteria are used. Static criteria include instructions, data types used, control flow and data accesses shape. Speculation on the control flow helps to remove control flow from the kernel. For instance, if a thread only consider one of the branch of a conditional statement, the conditional can be removed from the thread. The size of the data to be mapped in the GPU texture memory is also known. If the data size, for a given execution of the partitioned program is different than the one speculated, the GPU thread is not launch, the original code is then used instead. Parallelism analysis is obtained by combining classical data dependence and the computed speculative data space. This is particularly useful since aliasing in C program usually prohibits accurate program analysis.

Once a thread has been selected to be executed on the GPU the speculation checks that should be executed by the thread are analyzed and moved at the entry of the thread. This is performed prior giving the thread code to the code generator. Moving speculation checks at the beginning of the thread is usually possible thanks to the properties of the codes that fits GPU constraints (there is no data dependencies on the speculation tests). For instance, in the case of the example in Figure 3, before launching the threads, aliasing between the memory region of  $y$ ,  $x$  and  $M$  is checked. The last step is to include the communications in the main program. More details on ASTEX can be found in [11].

## 4 Conclusion

In this paper we have sketched the approach we are taken to achieve automatic partitioning of code to exploit graphical processing unit. ASTEX thread model is well suited for this task since speculation helps to deal with the many, static and dynamic, constraints of GPU. Current work is twofold. First, we are experimenting with NVIDIA to get an accurate model of the GPU behavior to help the selection. In parallel we develop a specific code generator from C to OpenGLS.

## References

- [1] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

- [2] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.
- [3] Benjamin Bustos, Oliver Deussen, Stefan Hiller, and Daniel Keim. A graphics hardware accelerated algorithm for nearest neighbor search. In Vassil N. Alexandrov, Geert Dick van Albada, Peter M.A. Sloot, and Jack Dongarra, editors, *Computational Science – ICCS 2006*, volume 3994 of *LNCS*, pages 196–199. Springer, 2006.
- [4] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. Gpu cluster for high performance computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47, Washington, DC, USA, 2004. IEEE Computer Society.
- [5] Naga K. Govindaraju, Scott Larsen, Jim Gray, and Dinesh Manocha. A memory model for scientific algorithms on graphics processors. In *Proceedings of Supercomputing'06*, 2006.
- [6] Cornwall J.L.T., Beckmann O., and Kelly P.H.J. Automatically translating a general purpose c++ image processing library for gpus. In *20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.
- [7] E. Scott Larsen and David McAllister. Fast matrix multiplies using graphics hardware. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 55–55, New York, NY, USA, 2001. ACM Press.
- [8] Philipp Lucas, Nicolas Fritz, and Reinhard Wilhelm. The development of the data-parallel gpu programming language CGIS. In Vassil N. Alexandrov, Geert Dick van Albada and Peter M.A. Sloot, and Jack Dongarra, editors, *Computational Science – ICCS 2006*, volume 3994 of *LNCS*, pages 200–203. Springer, 2006.
- [9] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.
- [10] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.
- [11] Eric Petit and Francois Bodin. Extracting speculative threads using traces for system on a chip. Technical Report 1789, IRISA, 2005.
- [12] Chris J. Thompson, Sahngyun Hahn, and Mark Oskin. Using modern graphics architectures for general-purpose computing: a framework and analysis. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 306–317, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.