Streaming-Oriented Parallelization of Domain-Independent Irregular Kernels *

J. Lobeiras, M. Amor, M. Arenaz, and B.B. Fraguela

Computer Architecture Group, University of A Coruña, Spain
{jlobeiras,margamor,manuel.arenaz,basilio.fraguela}@udc.es

Abstract. Current parallelizing and optimizing compilers use techniques for the recognition of computational kernels to improve the quality of the target code. Domain-independent kernels characterize the computations carried out in an application, independently of the implementation details of a given programming language. This paper presents streaming-oriented parallelizing transformations for irregular assignment and irregular reduction kernels. The advantage of these code transformations is that they enable the parallelization of many algorithms with little effort without a depth knowledge of the particular application. The experimental results show the efficiency on current GPUs, although the main goal of the proposed techniques is not performance, but assist the programmer in the parallelization for a better productivity.

Key words: Stream programming, domain-independent kernels, automatic parallelization, hardware accelerators, GPGPU

1 Introduction

The development and maintenance of applications that make efficient use of modern hardware architectures is a complex and time consuming task even for experienced programmers. Parallel application lifecycle costs are highly dependent on the hardware advances, especially in domains that change as fast as the GPUs (*Graphics Processing Units*). The use of GPUs for general purpose computation (or *GPGPU*) is becoming more relevant because of the increasing computational power and low cost of last-generation GPUs. However, from a programmability standpoint, CPUs have many advantages over GPUs due to the existence of standard programming languages like *C++* or *Java*, very powerful tools for software development and debugging, and well-known parallel programming *APIs* like *OpenMP*. Nowadays, GPU programming is more complicated as it requires using special languages (like *OpenCL* [9], *NVIDIA's CUDA* [12] or *ATI's Brook+* [1]) which often expose hardware features or limitations that restrict the flexibility of GPU programs.

* This work was supported by the Xunta de Galicia under projects INCITE08PXIB105161PR and 08TIC001206PR, and the Ministry of Science and Innovation, cofunded by the FEDER funds of the European Union, under the grant TIN2007-67536-C03-02. The authors are members of the HiPEAC network.

Recently, tools for the parallelization of sequential codes for modern GPUs are beginning to emerge. An *OpenMP*-like semiautomatic approach targeting regular codes as well as read-only irregular computations has been proposed [11]. A step forward towards automatic parallelization for these platforms is a *C*-to-*CUDA* parallel code generator for sequential affine (regular) programs based on the polyhedral model [6]. Despite these advances, the automatic parallelization of irregular applications for GPUs remains a great challenge.

Parallelizing compilers for multiprocessors address irregular applications by recognizing domain-independent computational kernels [5] (e.g. inductions, scalar reductions, irregular reductions and array recurrences) and by applying appropriate parallelizing transformations [8, 3]. The main contribution of this paper is the proposal of streaming-oriented parallelizing transformations for the well-known domain-independent kernels called irregular assignment and irregular reduction. Our strategies combine inspector-executor techniques, loop versioning and loop unrolling. A performance analysis using the *Brook+* language for GPU programming is also presented. *Brook+* is well suited for the streaming model that we are going to use in this work.

This paper is structured as follows. Section 2 describes the domain-independent irregular kernels and Section 3 presents the parallelizing transformations for a stream programming model. Section 4 describes our tests and shows the experimental results on a GPU. Finally, section 5 summarizes the main conclusions and future work.

2 Domain-Independent Irregular Kernels

Multiple definitions of computational kernel have been proposed in the literature in the context of automatic program analysis. In this work we use the *domain-independent concept-level computational kernels* recognized by the *XARK* compiler framework [5], which proved to be a useful tool for automatic parallelization of procedural and object-oriented programming languages [4], as well as for data locality optimization [2].

Domain-independent kernels (or simply *kernels* from now on) characterize the computations carried out in a program with independence of the programming language. These kernels do not take into account domain-specific problem solvers. Well-known examples are irregular assignment and irregular reduction, which will be described next.

2.1 Irregular Assignment

An *assignment kernel* consists in storing a value in a memory address. Within a program, this address can be accessed by a scalar variable, a memory pointer or an indexed variable, typically an array. Thus, an *irregular assignment* (see Algorithm 1) may be represented by a loop that computes a sentence $A(f(i)) = e(i)$, where A is the output array, f is an indirection array that introduces an unpredictable access pattern at compile-time, and e is an expression. Neither the

Algorithm 1: Irregular assignment
```
1: A(...) = ...
2: for i = 1 to Asize do
3:     A(f(i)) = e(i)
4: end for
```

Algorithm 2: Scalar reduction
```
1: v = ...
2: for i = 1 to n do
3:     v = v ⊕ e(i)
4: end for
```

Algorithm 3: Irregular reduction
```
1: A(...) = ...
2: for i = 1 to Asize do
3:     A(f(i)) = A(f(i)) ⊕ e(i)
4: end for
```

right-hand side expression $e(i)$ nor any function call within it contain occurrences of A. As a result, unless f is a permutation, output data dependencies will appear at run-time. This kernel can be found in application fields such as computer graphics, finite element applications or sparse matrix computations.

2.2 Irregular Reduction

The distinguishing characteristic of the *reduction kernel* is that the value stored in a memory address is computed using its previous value. The most popular one is the *scalar reduction* (see Algorithm 2), $v = v \oplus e(i)$, where the reduction variable v is a scalar, $\oplus$ is the reduction operator and $e(i)$ is a loop-variant expression whose value is not dependent on v. Scalar reductions appear in financial applications or statistical methods to obtain information of a sample, like the mean value. They are so common that programming languages usually provide some built-in support. An *irregular reduction*, $A(f(i)) = A(f(i)) \oplus e(i)$, is characterized by the use of an indirection array f that selects the locations of an array A to be updated (see Algorithm 3). Note that in this kernel loop-carried output and true data dependencies may appear at run-time. Irregular reductions are very common in many complex scientific applications and adaptive algorithms.

3 Parallelizing Transformations for the Streaming Model

In this section we describe streaming-oriented parallelizing transformations for irregular assignments and irregular reductions targeting current GPUs.

3.1 Irregular Assignment

In the literature, parallel irregular assignments for multiprocessors follow two main approaches. First, loop-partitioning oriented techniques [10] split the iteration space among processors and privatize the output array A. However, this technique is not of practical use on GPUs because memory requirements will grow proportionally to the number of threads (one copy of A for each thread), which limits its scalability and performance. Second, data-partitioning oriented techniques [3] split the iteration space and the output array A, reordering the loop iterations in order to balance the workload among the processors. Hereafter, we propose a data-partitioning oriented parallelizing transformation based on the inspector-executor model tuned for a streaming model.

The inspector-executor technique analyzes the contents of the indirection array f at runtime to determine which set of loop iterations must be assigned

```
Algorithm 4: Irr. Assignment Inspector
1: ins_table(1.. Asize) = 0
2: for i = 1 to Asize do
3:     ins_table(f(i)) = i
4: end for
```

```
Algorithm 5: Irr. Assignment Executor
1: for i = 1 to Asize do
2:     if ins_table(i) > 0 then
3:         A(i) = e(ins_table(i))
4:     end if
5: end for
```

```
Algorithm 6: Irr. Reduction Inspector
1: contention(1..Asize) = 0
2: max_cont = 0
3: for i = 1 to Asize do
4:     contention(f(i))++
5:     if contention(f(i)) > max_cont then
6:         max_cont = max_cont + 1
7:     end if
8: end for
9: ins_table(1..Asize, 1..max_cont) = 0
10: for i = 1 to Asize do
11:     dest = f(i)
12:     j = 1
13:     while ins_table(dest, j) > 0 do
14:         j = j + 1
15:     end while
16:     ins_table(dest, j) = i
17: end for
```

```
Algorithm 7: Irr. Reduction Executor
1: if max_cont > 4 then
2:     for i = 1 to Asize do
3:         j = 0
4:         while ins_table(i, j) > 0 do
5:             A(i) = A(i) ⊕ e(ins_table(i, j))
6:             j = j + 1
7:         end while
8:     end for
9: else
10:     for i = 1 to Asize do
11:         if ins_table(i, 0) > 0 then
12:             A(i) = A(i) ⊕ e(ins_table(i, 1))
13:         end if
            ...
20:         if ins_table(i, 3) > 0 then
21:             A(i) = A(i) ⊕ e(ins_table(i, 3))
22:         end if
23:     end for
24: end if
```

to each processor to avoid write conflicts. As shown in Algorithm 4, the inspector generates a table ins_table to store the last loop iteration that writes to each element of A. Algorithm 5 shows the executor, which uses ins_table to determine whether each element of the output array A remains unchanged ($ins_table(i) = 0$) or will be updated ($ins_table(i) > 0$) in iteration $ins_table(i)$.

Finally, some performance issues are briefly discussed. First, a given access pattern is often reused during the execution of an adaptive irregular application (this is called *reusability*). In this case, the extra cost of the inspector is amortized over several calls of the executor. Second, our inspector minimizes the cost of the executor by performing run-time dead code elimination, which removes any loop iterations that compute values of A overwritten in higher iterations.

3.2 Irregular Reduction

Techniques based on loop-partitioning and data-partitioning have also been proposed for irregular reduction parallelization in multiprocessors. In the scope of GPUs, we propose an inspector-executor technique that uses loop versioning and loop unrolling to efficiently exploit the available resources. The inspector code is shown in Algorithm 6. The goal is to create a table ins_table that stores all the iterations writing to a given element of A. First, the indirection array f is analyzed (lines 1-8) to compute the degree of contention, that is, the maximum number of writes to the same element of the output array A. Then, the degree of contention max_cont is used to statically allocate memory for ins_table in the GPU. Note that, in contrast to irregular assignments, all the iterations that contribute to an element need to be stored. Next, the executor presented

in Algorithm 7 is called to compute the parallel reduction. A set of conflict-free iterations can be assigned to each processor using *ins_table*. In the CPU each thread will compute a portion of the iterations, while on the GPU each thread will be assigned the reduction of a single location of A. This adaptation is only beneficial for streaming architectures because they heavily depend on multithreading techniques to hide memory access latencies. The GPU can also benefit from the use of both loop versioning and loop unrolling (see lines 11-24), storing the information on one or more *SIMD* short vectors (like *float4* or *int4*) which can be fetched in a single memory access.

4 Performance evaluation on a GPU using Brook+

Our test platform is composed by a *Phenom II X4 940* processor running at 3.0 *GHz*, 4 *GB DDR2 800 CL5* memory, a *790X* chipset based motherboard and a *Radeon 4850* GPU. The software setup is *WinXP x64* operating system, using *MS Visual C++ 2005* compiler (x64, release profile) and *Catalyst 9.12* driver.

As the programming language we use *Brook+ 1.4* [1], a *C* extension for *AMD* GPUs that exposes a stream programming model [7], designed to encourage and exploit a high degree of parallelism without significant compiler effort. In this paradigm the same function is applied to a set of inputs in parallel, producing another set of outputs, but there should be no overlapping between the input and the output data to prevent race conditions. The data inputs and outputs of a *streaming kernel* are called *streams* and each thread can only write to a certain location of the output stream, otherwise the performance is greatly reduced.

4.1 Benchmark suite

We designed several benchmarks to analyze the performance of the GPU using our streaming-oriented parallelization strategies. In the irregular assignment test *Asig_Irr*, the data of a matrix is updated using an indirection array whose values were generated using a uniform random distribution. As the number of indirections is equal to the size of the input, it is very likely that several iterations will try to update the same output address. To simulate a moderate computational load, the right-hand side of the assignment adds 100 integer numbers.

In the irregular reduction test *Red_Irr*, a matrix is updated by adding a value to those matrix locations specified by an indirection array generated using an uniform random distribution, thus again it is highly probable that more than one reduction will be performed on many of the matrix locations. The number of reductions for a given location can be easily estimated by a binomial distribution $B(N, 1/N)$. As in the previous case, in order to simulate some computational load, the reduction function will add 100 integers. Figure 1 shows an implementation using *Brook+* of the executor method for the irregular reduction kernel. It presents a general version for any degree of contention (*gpu_executor*), as well as a specialized version for degrees of contention less or equal to (*gpu_executor_f4*) that uses *float4* data type.

```
1:  // Function to execute, Brook+ GPU version
2:  kernel float
3:  gpu_fun (int pos<>, float val<>, float red<>)
4:  {
5:     return red + (float)pos * val;
6:  }

7:  // Executor code, Brook+ GPU general version
8:  kernel void
9:  gpu_executor(float src[][], int ins[][],
10:          float dst_R<>, out float dst_W<>,
11:          int max_cnt, int dimX)
12: {
13:    // 2D texture coordinates to 1D position
14:    int2 ins2D, pos2D = instance().xy;
15:    int i, pos = ADR_2Dto1D(ins, dimX);

16:    // Writes the previous value in the output array
17:    dst_W = dst_R;

18:    // Reads the inspector table
19:    for(i = 0; i < max_cnt; i++) {
20:       int p = ins[pos2D.y][max_cnt * pos2D.x + i];

21:       // Calls the reduction function when needed
22:       if(p < 0) break;
23:       ins2D = ADR_1Dto2D(p, DIMX);
24:       dst_W = gpu_fun(p,
                    src[ins2D.y][ins2D.x], dst_W);
25:    }
26: }
```

```
27: // Executor code, Brook+ GPU float4 version
28: kernel void
29: gpu_executor_f4(float src[][],
30:              int4 ins<>, float dst_R<>,
31:              out float dst_W<>,
32:              int dimX)
33: {
34:    // Obtains the locations to read from ins
35:    int2 ins2Da = ADR_I1(ins.x, dimX);
36:    int2 ins2Db = ADR_I1(ins.y, dimX);
37:    int2 ins2Dc = ADR_I1(ins.z, dimX);
38:    int2 ins2Dd = ADR_I1(ins.w, dimX);

39:    // Writes the previous value in the output array
40:    dst_W = dst_R;

41:    // Calls the reduction function when needed
42:    if(ins.x >= 0) dst_W =
          gpu_fun(ins.x, src[ins2Da.y] [ins2Da.x],
                dst_W);
43:    if(ins.y >= 0) dst_W =
          gpu_fun(ins.y, src[ins2Db.y] [ins2Db.x],
                dst_W);
44:    if(ins.z >= 0) dst_W =
          gpu_fun(ins.z, src[ins2Dc.y] [ins2Dc.x],
                dst_W);
45:    if(ins.w>= 0) dst_W =
          gpu_fun(ins.w, src[ins2Dd.y] [ins2Dd.x],
                dst_W);
46: }
```

Fig. 1: Brook+ versions of the executor for the irregular reduction kernel

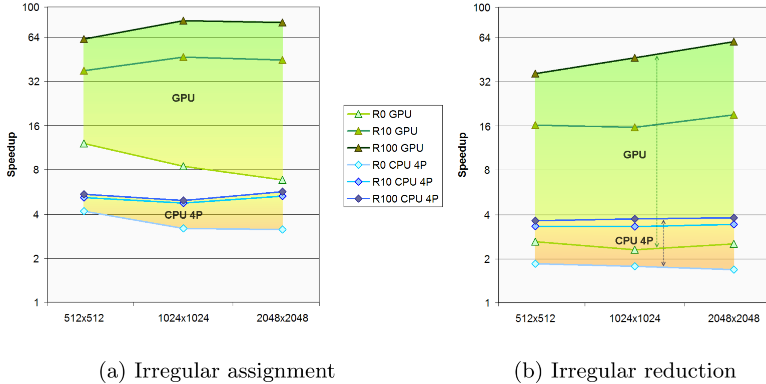
Table 1: Execution time (in sec.) and speedup for the 2048×2048 problem size

Benchmark		CPU 1P (Original)		CPU 2P (OMP)		CPU 4P (OMP)		GPU (Brook+)	
Asig_Irr	R0	71.44	–	37.20	(1.9x)	22.94	(3.1x)	10.59	(6.7x)
	R100	71.38	–	25.83	(2.8x)	12.70	(5.6x)	0.91	(78.4x)
Red_Irr	R0	75.09	–	65.69	(1.1x)	44.81	(1.7x)	29.98	(2.5x)
	R100	75.12	–	41.09	(1.8x)	19.97	(3.8x)	1.28	(58.7x)

4.2 Performance analysis

Here we analyze the performance of the proposed parallelization techniques on a GPU and on a multi-core CPU using *OpenMP*. The tests were run in single precision for matrices of sizes 512×512, 1024×1024 and 2048×2048, repeating each test 100 times to obtain meaningful times for the smaller problems. The computational cost of the tests tends to be deliberately low to study a worst case GPU scenario. Table 1 summarizes the execution times obtained in the tests using a 2048×2048 problem size as well as the respective speedups enclosed in parentheses. The time measured for the GPU includes the inspector and the data transfer of the analysis table between the CPU and the GPU. The execution of each inspector requires about 0.10 sec. for the *Asig_Irr* test, while the *Red_Irr* test requires about 0.25 sec. due to the additional memory and complexity.

Figure 2 shows the speedup of the two kernels for several problem sizes and for several reusability degrees ($R0$ if the inspector is not reused, $R10$ if it is reused 10 times, and $R100$ if it is reused 100 times). Under the same conditions of problem



(a) Irregular assignment (b) Irregular reduction

Fig. 2: Performance analysis of domain-independent irregular kernels

size and reusability, GPU performance is always better than the CPU, but for a good performance there should be some reusability in both cases. Otherwise, the execution time of the additional analysis required by the inspector stage is proportionately high. In the GPU, even a small reusability degree is able to compensate for the memory transfer times.

In the irregular assignment (Figure 2(a)) the optimal GPU performance is obtained for a 1024×1024 input. If there is no reusability, both architectures lose speedup as the problem size increases. The *OpenMP* implementation has superlinearity for the irregular assignment kernel because with the increase in the number of cores, the problem fits better in their caches. Also note that our inspector implementation is performing runtime dead code elimination, so the parallel execution can avoid computing some of the iterations.

Figure 2(b) shows the speedups for the irregular reductions. Although the speedups are not as remarkable as in the case of the irregular assignment, the parallelization is still beneficial. The reason behind this lower *speedup* is the additional bandwidth required by the inspector table in the GPU. Observe that in this case, the bigger the problem size, the more speedup the GPU is able to achieve over the CPU. In the GPU, every thread within a wavefront must execute the same code, so a certain degree of computing power will be wasted if the degree of contention is uneven. In problems where the contention has a large variance, the lookup table could be stored in a sparse matrix format like CRS (compressed row storage), however, accessing the data in the executor would require an additional indirection level, which according to our experiments lowers the efficiency on the GPU.

5 Conclusions and future work

This paper proposes streaming-oriented parallelizing transformations for two widely-used domain-independent computational kernels: the irregular assign-

ment and the irregular reduction. The strategy hinges on the inspector-executor model to split the iteration space and the output array with irregular access pattern. It also takes advantage of loop versioning and loop unrolling to exploit the hardware of the GPU. The paper proposes a performance evaluation on a GPU using *Brook+* and an *OpenMP*-based multi-core implementation. The results show good performance even in codes with low arithmetic intensity and irregular memory access patterns. Due to the complexity of GPU programming, peak performance is not the goal of this work. Rather our contribution is centered on maximizing the programmer productivity thanks to the described parallelization techniques. As future work we intend to study the parallelization of other less common kernels and port our work to other languages like *OpenCL* or *CUDA*.

References

1. AMD. *AMD Stream Computing User Guide*, 2009. v1.4.0a.
2. D. Andrade, M. Arenaz, B.B. Fraguela, J. Touriño, and R. Doallo. Automated and Accurate Cache Behavior Analysis for Codes with Irregular Access Patterns. *Concurrency and Computation: Practice and Experience*, 19(18):2407–2423, 2007.
3. M. Arenaz, J. Touriño, and R. Doallo. An Inspector-Executor Algorithm for Irregular Assignment Parallelization. In *Proc. of the 2nd International Symposium on Parallel and Distributed Processing and Applications (ISPA)*, pages 4–15, Hong Kong, China, 2004.
4. M. Arenaz, J. Touriño, and R. Doallo. Compiler Support for Parallel Code Generation through Kernel Recognition. In *Proc. of the 18th IEEE International Parallel and Distributed Processing Symposium*, page 79b, Santa Fe, New Mexico, 2004.
5. M. Arenaz, J. Touriño, and R. Doallo. XARK: An eXtensible Framework for Automatic Recognition of computational Kernels. *ACM Transactions on Programming Languages and Systems*, 30(6):1–56, 2008.
6. M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In *Proc. of 19th Internat. Conference on Compiler Construction (CC)*, volume 6011 of *Lecture Notes in Computer Science*, pages 244–263, Paphos, Cyprus, 2010.
7. J. Gummaraju and M. Rosenblum. Stream Programming on General-Purpose Processors. In *Proc. of the 38th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 343–354, Barcelona, Spain, 2005.
8. E. Gutiérrez, O.G. Plata, and E.L. Zapata. Data partitioning-based Parallel Irregular Reductions. *Concurrency and Computation: Practice and Experience*, 16(2–3):155–172, 2004.
9. Khronos OpenCL Working Group. *The OpenCL Specification*, 2009. v1.0.48.
10. K. Knobe and V. Sarkar. Array SSA Form and Its Use in Parallelization. In *Proc. of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 107–120, 1998.
11. S. Lee, S. Min, and R. Eigenmann. OpenMP to GPGPU: a Compiler Framework for Automatic Translation and Optimizations. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 101–110, Raleigh, NC, USA, 2009.
12. NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture*, 2009. v2.3.1.