Locality-Aware Automatic Parallelization for GPGPU with OpenHMPP Directives

José M. Andión · **Manuel Arenaz** ·
François Bodin · **Gabriel Rodríguez** ·
Juan Touriño

Abstract The use of GPUs for general purpose computation has increased dramatically in the past years due to the rising demands of computing power and their tremendous computing capacity at low cost. Hence, new programming models have been developed to integrate these accelerators with high-level programming languages, giving place to heterogeneous computing systems. Unfortunately, this heterogeneity is also exposed to the programmer complicating its exploitation. This paper presents a new technique to automatically rewrite sequential programs into a parallel counterpart targeting GPU-based heterogeneous systems. The original source code is analyzed through domain-independent computational kernels, which hide the complexity of the implementation details by presenting a non-statement-based, high-level, hierarchical representation of the application. Next, a locality-aware technique based on standard compiler transformations is applied to the original code through OpenHMPP directives. Two representative case studies from scientific applications have been selected: the three-dimensional discrete convolution and the simple-precision general matrix multiplication. The effectiveness of our technique is corroborated by a performance evaluation on NVIDIA GPUs.

Keywords heterogeneous systems · GPGPU · locality · automatic parallelization · OpenHMPP · domain-independent kernel

1 Introduction

The use of GPUs for general purpose computation (GPGPU) has increased dramatically in the past years [29] mainly due to two reasons. On the one hand, the hard-

J.M. Andión · M. Arenaz · G. Rodríguez · J. Touriño
Dep. de Electrónica e Sistemas, Universidade da Coruña, Campus de Elviña, 15071 A Coruña, Spain
E-mail: {jandion,manuel.arenaz,grodriguez,juan}@udc.es

F. Bodin
Institut de Recherche en Informatique et Systèmes Aléatoires, Campus de Beaulieu, 35042 Rennes, France
E-mail: bodin@irisa.fr

ware industry has not been able to satisfy the rising demands of computing power while preserving the sequential programming model. Computer users expect their applications to run faster with each new generation of microprocessors, but this is not the case in the multicore era. A sequential program will only run on one of the cores, which will not become faster. Thus, the software community has been forced to develop and use parallel programming tools. On the other hand, GPUs offer a tremendous computing capacity at low cost due to the economic pressure of the video game industry. The transition from fixed-function to programmable shaders has made these computational resources useful for general purpose programming [19]. First approaches (OpenGL [31], Cg [23]) forced programs to look like graphics applications that drew triangles and polygons, limiting the accessibility of GPUs. Therefore, new programming models have been developed to integrate GPUs with high-level programming languages, giving place to heterogeneous computing systems.

The main drawback of these systems is that heterogeneity is exposed to the developer. Programming is hard, and parallel architectures make it harder because they require additional tasks to parallelize and tune for optimum performance. With most tools for GPU programming, developers have to deal with many low-level characteristics and limitations. Writing a GPU application by hand consumes a huge quantity of time, even to experienced programmers, and it is an error-prone activity. Exploiting locality in GPUs is key to achieving good performance, and it is more challenging than in CPUs. Moreover, the effect of a code transformation on the execution time is often unpredictable even for GPU experts. Nowadays, several directive-based approaches have appeared to program GPU-based heterogeneous systems (OpenMP 4.0 [28], OpenACC [32], OpenHMPP [27], PGI Accelerator [36], OpenMPC [20], hiCUDA [13]). Directives combine portability and good performance at the same time [21]. Thus, we believe that a directive-based approach is a suitable choice for the automatic parallelization of sequential applications on GPUs.

The main contribution of this paper is three-fold:

1. A new technique to automatically rewrite sequential programs into a parallel counterpart targeting GPU-based heterogeneous systems. This locality-aware technique exploits the GPU hardware architecture through OpenHMPP directives.
2. The successful application of this technique to two representative case studies extracted from compute-intensive scientific applications: the three-dimensional discrete convolution (CONV3D), and the simple-precision general matrix multiplication (SGEMM).
3. The performance evaluation of our technique corroborating its effectiveness.

The remainder of the paper is organized as follows. Section 2 briefly introduces GPGPU, describes the CUDA programming model [26] and highlights the GPU hardware features that impact on performance. Section 3 reviews the OpenHMPP standard and the additional functionality supported by CAPS Compilers [22] that is relevant for this work. Section 4 gives a summary on the background of compilation techniques used in this paper, namely the KIR [1] and the chains of recurrences [37]. Section 5 introduces the new locality-aware optimization technique for GPUs. Section 6 details the operation of our approach with the CONV3D and SGEMM case studies. Section 7 presents the performance evaluation. Section 8 discusses related work and, finally, Sect. 9 concludes the paper and presents future work.

2 GPGPU with the CUDA Programming Model

GPUs were designed to show images on displays. Certain stages of the graphics pipeline perform floating-point operations on independent data, such as transforming the positions of triangle vertices or generating pixel colors. Therefore, GPUs execute thousands of concurrent threads in an SIMD fashion requiring high-bandwidth memory access. This design goal is achieved because GPUs devote more transistors than CPUs to data processing instead of data caching and control flow.

In this paper we have used NVIDIA GPUs. This company developed CUDA [26], which enables the use of C as GPU programming language. The programmer defines C functions, called *CUDA kernels*, that specify the operation of a single GPU thread. Three ideas are behind CUDA. First, lightweight parallel threads are organized into a hierarchy: a *grid* of blocks, a *block* of threads. Blocks may execute in parallel allowing easy scalability. Second, CUDA defines a *shared memory* between the threads of a block to enable fast data interchange. And third, the execution of the threads of a block can be synchronized with a barrier. In addition, CUDA exposes a complex memory hierarchy that has to be explicitly managed by the programmer.

The hardware implementation of CUDA consists of an array of *Streaming Multiprocessors (SMs)*, where each SM executes the threads of a block in groups of 32 called *warps*. The threads of a warp execute one common instruction at a time. The *compute capability* of an NVIDIA GPU defines its core architecture (Tesla, Fermi, Kepler), supported features (e.g., double-precision floating-point operations), technical specifications (e.g., the maximum dimensions of the hierarchy of threads) and architectural specifications (e.g., the number of warp schedulers).

In summary, CUDA exposes the GPU hardware architecture through programming features that the GPGPU developer must handle to generate efficient code:

1. Threadification, i.e., the policy that guides the creation of GPU threads and what code they will execute. Each thread has a unique identifier that is commonly used to access data stored in the GPU memories, in a similar way to loop indices.
2. Thread grouping, so that threads are dispatched in warps to SMs.

The CUDA C Best Practices Guide [25] also prioritizes some strategies to improve the performance of the GPU code:

3. Minimization of CPU-GPU data transfers.
4. Coalesced accesses to global memory, i.e., several memory accesses from different threads are handled by a unique transaction to the global memory.
5. Maximum usage of registers and shared memory to avoid redundant accesses to global memory (the biggest but slowest one).
6. Avoidance of thread divergence, i.e., threads within the same warp following different execution paths.
7. Sufficient occupancy, i.e., sufficient number of active threads per SM.
8. The number of threads per block must be a multiple of 32.

The most relevant programming features in points (1)–(8) have been considered in the design of our locality-aware technique to tune the performance of the automatically generated GPU parallel code. The next section describes the support provided by OpenHMPP for those programming features.

3 OpenHMPP Directives and CAPS Compilers

CAPS Entreprise offers a complete suite of software tools to develop high performance parallel applications targeting heterogeneous systems based on manycore accelerators. The most relevant ones are CAPS Compilers [22], which generate CUDA [26] and OpenCL [30] code from a sequential application annotated with compiler directives. Directive-based approaches (as the well-known OpenMP [28]) try to reduce the programming effort and provide more readable codes. In this way, these approaches ease the interaction between application-domain experts and programmers. The sequential and the parallel versions coexist in the same file offering an incremental way to migrate applications. The developed codes are independent from the hardware platform and new hardware accelerators supported by the translator are automatically exploited. In addition, reasonable performance is achieved compared to hand-written GPU codes [21]. Thus, we consider that compiler directives offer a convenient instrument for the automatic parallelization of sequential applications on GPU-based heterogeneous systems.

Among the numerous proposals of compiler directives to exploit these systems, three standardization efforts have emerged throughout the last years: OpenHMPP [27], OpenACC [32] and, finally, OpenMP 4.0 [28]. All of them follow a similar approach regarding the interaction between the host and the accelerator: they present a Remote Procedure Call (RPC) paradigm that offloads a region of code from the CPU to be executed on the GPU. The address spaces of the host and the accelerator are considered to be disjoint, but data transfers are automatically inserted when needed. However, the programmer is allowed to explicitly manage these transfers in order to improve the performance (for instance, overlapping them with computations through asynchronous calls or specifying only portions of arrays to be copied).

Nevertheless, there exist some differences between the functionality offered by the standards. GPUs commonly have software-managed caches (e.g., the shared memory in the CUDA programming model) whose exploitation is key to achieving good performance. Only OpenACC and OpenHMPP provide a mechanism to explicitly handle this memory, while OpenMP relies on the implementer. Another significant difference exists when specifying parallelism. OpenHMPP exposes a set of threads where each thread executes a loop iteration. OpenACC presents three levels of parallelism: the programmer can launch a set of *gangs* executing in parallel, where each gang may support multiple *workers*, each with vector or SIMD operations. OpenMP presents a set of threads that are organized in *teams* and can run loop iterations or explicit tasks. This standard can exploit SIMD operations too.

In this work, we have selected OpenHMPP (formerly known as HMPP [8]) and the extension HMPPCG (HMPP Codelet Generator) because these sets of directives provide unique functionality to transform loop nests, which allow the fine tuning of the generated GPU code, and both their compiler and their runtime are much more mature. However, these loop transformations can be performed without directives and we will be able to use OpenACC when a complete implementation is developed. Regarding the recently approved OpenMP 4.0, the explicit management of the complete memory hierarchy by the programmer has not been considered, but our work can help to exploit locality in the implementations of the standard.

OpenHMPP supports the programming features mentioned in points (1)–(8) of Sect. 2 in the following way:

1. The `gridify` directive performs threadification on loops and thread grouping as follows. For simple loops, it generates consecutive GPU threads for consecutive loop iterations, one thread per iteration. For loop nests, it implements a 2D threadification process with the two outermost loops in the nest; consecutive GPU threads are created for consecutive iterations of the second loop.
2. The `advancedload` and `delegatedstore` directives, with the `asynchronous` clause, allow the overlapping between CPU-GPU data transfers and computations. In addition, it is possible to specify only portions of arrays to be transferred.
3. The `permute`, `unroll`, `fuse`, `tile`. . . directives perform standard compiler transformations on loops. These directives are used to fine tune the performance of the generated GPU code.
4. The `gridify` directive also enables the allocation of program variables on the different GPU memories (for instance, the `shared` clause for the shared memory).

Our technique will use these OpenHMPP mechanisms to automatically generate GPU code. Next, the compilation foundations of our approach are summarized.

4 Background of Compilation Techniques

Despite great advances in compiler technology during the last decades, current compilers usually fail to parallelize even simple sequential programs. For the successful automatic parallelization of applications, two complex problems have to be addressed: first, the detection of parallelism to determine what parts in the original source code can be executed concurrently; and second, the generation of efficient parallel code taking into account the underlying hardware architecture.

This section covers parallelism detection and is organized as follows. Section 4.1 briefly describes the KIR [1], a novel compiler Intermediate Representation (IR). After that, Sect. 4.2 introduces the chains of recurrences, an algebraic formalism for the characterization of memory access patterns.

4.1 The KIR: an IR for the Detection of Parallelism

Compilers typically address the automatic detection of parallelism by running classical dependence analyses on standard statement-based IRs (e.g., Abstract Syntax Trees —ASTs—, Data Dependence Graph —DDG—, Control Flow Graph —CFG—, Dominator Tree —DT—). Such IRs are well suited for code generation, but not for the detection of parallelism. Previous work presented the KIR [1], a new compiler IR that eases the automatic detection of parallelism in sequential codes. Our IR hides the complexity of the implementation details presenting a non-statement-based, high-level, hierarchical representation of the application.

The KIR is based on the concept of domain-independent computational kernel (from now on, *diKernel*). This new IR consists of a set of diKernels and dependence relationships between them representing DDG edges that cross diKernel boundaries.

In order to capture the order in which diKernels are executed, flow dependences between diKernels are identified using the CFG, the DDG, the DT and the production and use of values throughout the program. Finally, the KIR also comprises a hierarchy of execution scopes (based on the hierarchy of loops) that reflects the computational stages of the sequential program and groups diKernels into these stages.

Multiple definitions of the term *computational kernel* have been proposed in the literature in the context of automatic program analysis. The diKernels do not represent domain-specific problem solvers. Instead, they characterize the computations carried out in a program from the point of view of the automatic detection of parallelism. A detailed description of the collection of diKernels can be consulted in [5]. The diKernels that appear in this paper are:

- **scalar assignment** $v = e$, which stores the value of the expression e in the memory address specified by the scalar variable v. The value e is not dependent on v, that is, neither e nor any function call within it contain occurrences of v.
- **scalar reduction** $v = v \oplus e(i)$ where the variable v is a scalar, $\oplus$ is an associative and commutative operator, i is an affine expression of the enclosing loop indices, and $e(i)$ is an expression that may depend on i but it must not depend on v.
- **regular reduction** $A[i] = A[i] \oplus e(i)$ where $A[i]$ represents an entry of the array A, i is an affine expression of the enclosing loop indices, $\oplus$ is an associative and commutative operator, and $e(i)$ is an expression that may depend on i but it must not depend on A.

For illustrative purposes, consider the source code of the 3D convolution operator shown in Fig. 1a. The corresponding KIR is shown in Fig. 1b. The loops are perfectly nested, thus they are represented by a unique execution scope $ES\_for_{i,j,k}$. One diKernel is created for each temporary variable, which stores the calculations in each 3D axis: $K{<}tempx_7{>}$, $K{<}tempy_{14}{>}$ and $K{<}tempz_{21}{>}$. Note that the subindices refer to the line number in the source code (e.g., the term $tempx_7$ refers to the statement in lines 7–13 of Fig. 1a). Their contribution to the final result $K{<}output_{28}{>}$ is symbolized by diKernel-level flow dependences ($\Rightarrow$). Scalars $tempx$, $tempy$ and $tempz$ are assigned new values in each $for_{i,j,k}$ iteration, thus $K{<}tempx_7{>}$, $K{<}tempy_{14}{>}$ and $K{<}tempz_{21}{>}$ are scalar assignments. In contrast, the value stored in $output[i][j][k]$ depends on the previous one and thus $K{<}output_{28}{>}$ is a regular reduction. The diKernels that represent loop indices are not shown because they are already represented in the notation of the execution scope and the types of the remaining diKernels.

4.2 Chains of Recurrences in the KIR

Chains of recurrences (from now on, *chrecs*) are an algebraic formalism to represent closed-form functions which have been successfully used to expedite function evaluation at a number of points in a regular interval [37]. Given a constant ϕ, a function g defined over the natural numbers and zero ($\mathbb{N} \cup \{0\}$), and the operator $+$, the chrec $\{\phi, +, g\}$ is defined as a function:

$$\{\phi, +, g\}(i) = \phi + \sum_{j=0}^{i-1} g(j) \text{ with } i \in \mathbb{N} \cup \{0\}$$

```
1  int i, j, k, sizex, sizey, sizez;
2  float coefx,coefy,coefz,*input,*output;
3
4  for (i = 0; i < sizex; i++) {
5   for (j = 0; j < sizey; j++) {
6    for (k = 0; k < sizez; k++) {
7     float tempx = input[i][j][k]+coefx*
8       (
9        input[i-1][j][k]+input[i+1][j][k]+
10       input[i-2][j][k]+input[i+2][j][k]+
11       input[i-3][j][k]+input[i+3][j][k]+
12       input[i-4][j][k]+input[i+4][j][k]
13      );
14    float tempy = input[i][j][k]+coefy*
15      (
16       input[i][j-1][k]+input[i][j+1][k]+
17       input[i][j-2][k]+input[i][j+2][k]+
18       input[i][j-3][k]+input[i][j+3][k]+
19       input[i][j-4][k]+input[i][j+4][k]
20      );
21    float tempz = input[i][j][k]+coefz*
22      (
23       input[i][j][k-1]+input[i][j][k+1]+
24       input[i][j][k-2]+input[i][j][k+2]+
25       input[i][j][k-3]+input[i][j][k+3]+
26       input[i][j][k-4]+input[i][j][k+4]
27      );
28    output[i][j][k] =
29      output[i][j][k]+tempx+tempy+tempz;
30   }
31  }
32 }
```

(a) Source code.
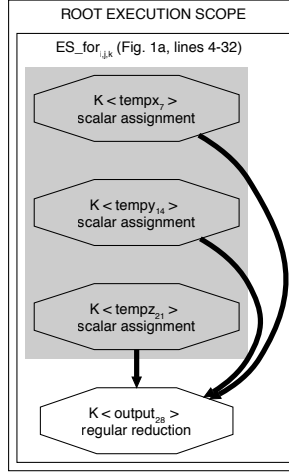
(b) KIR.

Fig. 1: The 3D discrete convolution operator (CONV3D).

The chrecs, which are provided by the KIR, have demonstrated to be a powerful representation of the complex loops and the memory accesses that appear in full-scale real applications [2]. For example, the loop index of for_i in Fig. 1a takes integer values in the interval $[0, sizex - 1]$. The chrec $\{0, +, 1\}$ provides a closed-form function to compute the value of i at each for_i iteration and thus to determine the memory access pattern i in the first dimension of $input[i][j][k]$ (see line 7 of Fig. 1a).

The algebraic properties of chrecs provide rules for carrying out arithmetic operations with them [37]. For instance, the addition of a chrec and a constant c is given by $\{\phi, +, g\} + c = \{\phi + c, +, g\}$. This rule enables the representation of the access pattern in the first dimension of $input[i - 1][j][k]$ (see line 9 of Fig. 1a) as $\{0, +, 1\} - 1 = \{-1, +, 1\}$. Hence, chrecs can be computed to completely describe the access pattern for n-dimensional arrays. For illustrative purposes, the first two accesses to *input* are modeled as:

$$CHREC\_input_1 = [\{0, +, 1\}][\{0, +, 1\}][\{0, +, 1\}]$$

$$CHREC\_input_2 = [\{-1, +, 1\}][\{0, +, 1\}][\{0, +, 1\}]$$

Note that a chrec is computed for each one of the dimensions of the array.

In this paper, we introduce the term *instantiated chrecs* to refer to the chrecs that represent the memory accesses performed by each GPU thread after loop threadification (see point (1) of Sect. 2). Hence, we fix the value of the index of the threadified

loop that the thread executes. For instance, assuming that for_i is threadified, the instantiated chrecs for $input[i][j][k]$ (see line 7 of Fig. 1a) for the GPU thread $T0$ are:

$$CHREC\_input_1^{T0} = [\{0,+,0\}][\{0,+,1\}][\{0,+,1\}]$$

From now on, the notation of the chrecs with the form $\{\phi,+,0\}$ (i.e., $g = 0$) will be simplified to $\{\phi\}$. In the previous example, the chrec $\{0,+,0\}$ will be written as $\{0\}$ representing that thread $T0$ always executes $input[i][j][k]$ with $i = 0$.

5 Locality-Aware Automatic Generation of Efficient GPGPU Code

The generation of parallel code is a complex problem that a parallelizing compiler has to address. Previous work [1] presented an OpenMP-based hardware-independent approach targeting multicore processors that consists of two steps:

1. Filtering out the *spurious* diKernel-level dependences, which are those that do not prevent the parallelization of the sequential application. The subgraphs of the KIR that represent the computations carried out on privatizable scalar variables [11] are *shaded* to be omitted in the discovery of parallelism. Regarding the example of Fig. 1, scalars *tempx*, *tempy* and *tempz* are loop temporaries as they are recomputed at the beginning of each $for_{i,j,k}$ iteration. Thus, they are privatizable and the corresponding parts of the KIR are shaded (see the shaded region in Fig. 1b).

2. The construction of an efficient parallelization strategy based on the application features captured by the KIR. The parallel code generation is based on the existence of parallelizing transformations for each type of diKernel. For instance, a scalar reduction can be parallelized in three phases with privatization support [34]. A regular reduction represents conflict-free loop iterations that can be transformed into forall parallel loops. Other examples are discussed in [4].

This OpenMP-based hardware-independent approach has demonstrated to be effective for multicore processors [1]. However, for peak performance on the GPU, the generated code must exploit its characteristic hardware architecture (in particular, the complex memory hierarchy). Hereafter, we introduce a new locality-aware code generation technique that extends the previous approach considering the most impacting programming features enumerated in points (1)–(8) of Sect. 2: loop threadification (1), thread grouping (2), coalesced access to global memory (4), and maximum usage of registers and shared memory (5). The minimization of CPU-GPU data transfers (3) will be addressed with a new automatic partitioning algorithm of the KIR, which will decide what parts of the computations of full-scale applications must be executed on the CPU or on the GPU. Therefore, we assume that program data fits into the GPU memory and, in our experiments (see Sect. 7), we have measured the execution times excluding CPU-GPU data transfers. This paper does not address the avoidance of thread divergence (6) as it is a problem related to the algorithm implemented by the given source code. In addition, maintaining sufficient occupancy (7) or determining the best block size (8) are programming features very close to the concrete GPU hardware that executes the code and their optimization needs runtime information, thus they are out of the scope of this paper.

Algorithm 1 Detection of whether an access to GPU global memory can be coalesced

---

1: **FUNCTION** ISCOALESCEDACCESS
Input: access $x_k[i_{k,1}][i_{k,2}]\dots[i_{k,n}]$ to an n-dimensional array x stored in row-major order
Input: loop nest $L = L_1, L_2, \dots, L_l$ where L_1 is the threadified loop
Output: returns whether the given access x_k can be coalesced after threadifying the loop nest L
2:     $CHRECS\_x_k \leftarrow [\{\phi_{k,1}, +, g_{k,1}\}][\{\phi_{k,2}, +, g_{k,2}\}]\dots[\{\phi_{k,n}, +, g_{k,n}\}]$
3:     $W \leftarrow$ warp of GPU threads $\{T0, T1, T2\dots\}$
4:     **for each** thread Ti in W **do**
5:         $CHRECS\_x_k^{Ti} \leftarrow [\{\phi_{k,1}^{Ti}, +, g_{k,1}^{Ti}\}][\{\phi_{k,2}^{Ti}, +, g_{k,2}^{Ti}\}]\dots[\{\phi_{k,n}^{Ti}, +, g_{k,n}^{Ti}\}]$
6:     **end for**
7:     **if** $(\exists d \in \{1, \dots, n-1\}, Tj \in W - \{T0\} : \{\phi_{k,d}^{Tj}, +, g_{k,d}^{Tj}\} \neq \{\phi_{k,d}^{T0}, +, g_{k,d}^{T0}\})$ **then**
8:         **return** false
9:     **end if**
10:    $CHRECS\_RANGE\_x_{k,n} \leftarrow \bigcup^{Ti} \{\phi_{k,n}^{Ti}, +, g_{k,n}^{Ti}\}$
11:    **if** $CHRECS\_RANGE\_x_{k,n}$ defines a contiguous range **then**
12:        **return** true
13:    **else**
14:        **return** $(\forall Tj \in W - \{T0\} : \{\phi_{k,n}^{Tj}, +, g_{k,n}^{Tj}\} = \{\phi_{k,n}^{T0}, +, g_{k,n}^{T0}\})$
15:    **end if**
16: **end FUNCTION**

5.1 Detection of Coalesced Accesses to the GPU Global Memory

According to the CUDA Best Practices Guide [25], coalescing is maximized (and thus memory requests are minimized) if the threads of a warp access consecutive memory locations. Algorithm 1 identifies coalesced accesses by taking into account loop threadification, thread grouping and chrecs. As mentioned in Sect. 4.2, for an access x_k to an array x in a loop nest L, the KIR provides the chrecs associated to each array dimension (see line 2 of Alg. 1). Next, chrecs are instantiated to represent the memory accesses performed by each GPU thread by fixing the value of the index of L_1 that the thread executes (lines 4–6). Assuming row-major storage, consecutive memory positions are given by consecutive accesses to the last dimension of the array x. Thus, the first $n-1$ chrecs must be the same (lines 7–9). Finally, if the union of the chrecs of the last dimension defines a contiguous range, then the accesses are coalesced (lines 10–12). If the chrecs of the last dimension are equal, then the same memory position is accessed and only one memory transaction is needed (line 14).

For illustrative purposes, Fig. 2a and 2c present the two possibilities to traverse a 2D array x: row-major traversal (denoted S1) and column-major traversal (S2). Arrays are stored in row-major order in C and thus S1 accesses array x row by row, exploiting locality and minimizing data cache misses on the CPU. Assume that only the outer loop of a nest is threadified on the GPU (contrary to the OpenHMPP default policy —see Sect. 3—). Hence, each GPU thread will access consecutive memory positions: $T0$ will access $x[0][0]$, $x[0][1]$, $x[0][2]\dots$ (see Fig. 2b). Therefore, for the iteration $j = 0$, the threads of the first warp ($T0$, $T1$, $T2\dots$) will access to the non-consecutive memory locations $x[0][0]$, $x[1][0]$, $x[2][0]\dots$ and these memory requests cannot be coalesced by the GPU memory controller. Algorithm 1 detects this non-coalesced access pattern as follows. The KIR provides (see line 2 of Alg. 1):

$$CHRECS\_x_k = [\{0, +, 1\}][\{0, +, 1\}]$$

```
1 | // only for_i is threadified
2 | for (i = 0; i <= N; i++) {
3 |    for (j = 0; j <= N; j++) {
4 |       ... x[i][j] ...
5 |    }
6 | }
```

(a) Source code S1.

```
1 | // only for_j is threadified
2 | for (j = 0; j <= N; j++) {
3 |    for (i = 0; i <= N; i++) {
4 |       ... x[i][j] ...
5 |    }
6 | }
```

(c) Source code S2.

		$T0$ $(i{=}0)$	$T1$ $(i{=}1)$	$T2$ $(i{=}2)$
$j{=}0$		$x[0][0]$	$x[1][0]$	$x[2][0]$
$j{=}1$		$x[0][1]$	$x[1][1]$	$x[2][1]$
$j{=}2$		$x[0][2]$	$x[1][2]$	$x[2][2]$
...		...	...	...
chrecs $1^{st}dim$		$\{0\}$	$\{1\}$	$\{2\}$
$2^{nd}dim$		$\{0,+,1\}$	$\{0,+,1\}$	$\{0,+,1\}$

(b) Non-coalesced accesses.

		$T0$ $(j{=}0)$	$T1$ $(j{=}1)$	$T2$ $(j{=}2)$
$i{=}0$		$x[0][0]$	$x[0][1]$	$x[0][2]$
$i{=}1$		$x[1][0]$	$x[1][1]$	$x[1][2]$
$i{=}2$		$x[2][0]$	$x[2][1]$	$x[2][2]$
...		...	...	...
chrecs $1^{st}dim$		$\{0,+,1\}$	$\{0,+,1\}$	$\{0,+,1\}$
$2^{nd}dim$		$\{0\}$	$\{1\}$	$\{2\}$

(d) Coalesced accesses.

Fig. 2: Examples of access patterns to the GPU global memory.

Next, chrecs are instantiated (lines 4–6):

$$CHRECS\_x_k^{T0} = [\{0\}][\{0,+,1\}],\ CHRECS\_x_k^{T1} = [\{1\}][\{0,+,1\}]\dots$$

They are different for the first dimension, thus the threads cannot access consecutive memory positions (lines 7–9).

In contrast, j drives the access to the last dimension of array x in S2 (see Fig. 2c). This code will run poorly on the CPU in the common situation when the array x is bigger than the cache memory. However, on the GPU, $T0$ will access to $x[0][0]$, $x[1][0]$, $x[2][0]\dots$ (see Fig. 2d). Hence, for the iteration $i = 0$, the threads of the first warp ($T0$, $T1$, $T2\dots$) will access the consecutive memory locations $x[0][0]$, $x[0][1]$, $x[0][2]\dots$ and these memory requests can be coalesced. Algorithm 1 detects this coalesced access pattern as follows. The KIR provides (see line 2 of Alg. 1):

$$CHRECS\_x_k = [\{0,+,1\}][\{0,+,1\}]$$

Next, chrecs are instantiated (lines 4–6):

$$CHRECS\_x_k^{T0} = [\{0,+,1\}][\{0\}],\ CHRECS\_x_k^{T1} = [\{0,+,1\}][\{1\}]\dots$$

They are the same for the first dimension, thus the threads may access consecutive memory positions (lines 7–9). The union of the last chrecs $\{0\} \cup \{1\}\dots$ defines a contiguous range and therefore the performed accesses maximize coalescing and correctly exploit the GPU global memory locality (lines 10–12).

Algorithm 1 is invoked for all the array accesses enclosed in the loop nests of the program. If the index of the threadified loop does not drive the access to the last dimension of the array, a general strategy to try to exploit coalescing is to permute the loops of the nest as will be seen in Sect. 6.

5.2 Maximization of the Usage of Registers and Shared Memory

As mentioned in point (5) of Sect. 2, the GPU global memory is the biggest but slowest one. Both registers and shared memory are faster, but they have much less capacity. Therefore, this complex memory hierarchy should be managed with even more care than the traditional CPU memory hierarchy to obtain good performance.

Algorithm 2 presents a technique to detect reused data within a GPU thread. First, it collects all the accesses to an n-dimensional array x in a loop nest L (see line 2 of Alg. 2). Next, the KIR provides the chrecs associated to each access in each array dimension (line 3). For each thread, the chrecs are instantiated by fixing the value of the index of L_1 that the thread executes (line 5). If the intersection of the instantiated chrecs for the GPU thread is not empty, then some data are accessed several times and they can be stored in the GPU registers if they are not modified by another thread (lines 6–9). The shared memory can be used for the same purpose.

However, the GPU shared memory has been specifically designed to share data between the threads of a block. Algorithm 3 presents a technique that takes into account all the accesses to an n-dimensional array x in a loop nest L (see line 2 of Alg. 3). The KIR provides the chrecs associated to each access in each array dimension (line 3). For each thread, the chrecs are instantiated by fixing the value of the index of L_1 that the thread executes (lines 5–7). If the intersection of the instantiated chrecs associated to all the accesses is not empty, then some data are accessed several times and can be stored in the shared memory (lines 8–11).

Another general technique to improve performance is loop tiling. It consists of partitioning the loop iterations into blocks to ensure that data being used stay in the faster levels of the memory hierarchy. As explained in Sect. 3, OpenHMPP implements loop threadification and thread grouping with the two outermost loops in a nest; consecutive GPU threads are created for consecutive iterations of the second loop. Therefore, the common $m \times n$ tiling breaks coalescing because the step of L_2 is different from one and thus consecutive threads will not access consecutive memory locations. Algorithm 4 presents a technique for loop tiling that preserves coalescing under OpenHMPP and also considers the promotion of the enclosed scalar variables. Instead of creating a thread for each access x_k, a bigger portion of data to compute (Δ) is given to each thread. Hence, the algorithm increments the step of L_1 to $i = i + \Delta$ (see line 2 of Alg. 4). Scalar variables inside L are promoted to arrays of size Δ, and their corresponding reads and writes are transformed into loops preserving dependences (lines 3–6). The optimization of the size of Δ depends on runtime information about the GPU hardware, thus it has been set by hand in this paper. This technique can be complemented with loop unrolling and loop interchange. Typically, GPU compilers make better optimizations if the program is coded with several instructions using scalar variables. In this way, the GPU compiler is able to store them in registers.

6 Case Studies

This section details the operation of the locality-aware automatic parallelization technique introduced in Sect. 5. We have selected two representative case studies ex-

---

Algorithm 2 Usage of registers to store reused data within a GPU thread

---

1: **PROCEDURE** STOREREUSEDDATAINREGISTERS
Input: n-dimensional array $x[s_1][s_2]\ldots[s_n]$
Input: loop nest $L = L_1, L_2, \ldots, L_l$ where L_1 is the threadified loop
Output: a modified program that exploits reused data to maximize the usage of the GPU registers
2:     collect accesses $x_k[i_{k,1}][i_{k,2}]\ldots[i_{k,n}]$ with $k \in \{1,\ldots,m\}$
3:     $CHRECS\_x_k \leftarrow [\{\phi_{k,1},+,g_{k,1}\}][\{\phi_{k,2},+,g_{k,2}\}]\ldots[\{\phi_{k,n},+,g_{k,n}\}]$
4:     **for each** thread Ti **do**
5:         $CHRECS\_x_k^{Ti} \leftarrow [\{\phi_{k,1}^{Ti},+,g_{k,1}^{Ti}\}][\{\phi_{k,2}^{Ti},+,g_{k,2}^{Ti}\}]\ldots[\{\phi_{k,n}^{Ti},+,g_{k,n}^{Ti}\}]$
6:         $REUSED\_DATA\_x^{Ti} \leftarrow \bigcap_{k=1}^{m} CHRECS\_x_k^{Ti}$
7:         **if** $(REUSED\_DATA\_x^{Ti} \neq \emptyset)$ **then**
8:             store reused data between the accesses made by Ti in registers if data are private
9:         **end if**
10:     **end for**
11: **end PROCEDURE**

Algorithm 3 Usage of the GPU shared memory for data shared between the threads of a block

---

1: **PROCEDURE** STORESHAREDDATAINSHAREDMEMORY
Input: n-dimensional array $x[s_1][s_2]\ldots[s_n]$
Input: loop nest $L = L_1, L_2, \ldots, L_l$ where L_1 is the threadified loop
Output: a modified program using the GPU shared memory to share data between the threads of a block
2:     collect accesses $x_k[i_{k,1}][i_{k,2}]\ldots[i_{k,n}]$ with $k \in \{1,\ldots,m\}$
3:     $CHRECS\_x_k \leftarrow [\{\phi_{k,1},+,g_{k,1}\}][\{\phi_{k,2},+,g_{k,2}\}]\ldots[\{\phi_{k,n},+,g_{k,n}\}]$
4:     **for each** block B **do**
5:         **for each** thread Ti in B **do**
6:             $CHRECS\_x_k^{Ti} \leftarrow [\{\phi_{k,1}^{Ti},+,g_{k,1}^{Ti}\}][\{\phi_{k,2}^{Ti},+,g_{k,2}^{Ti}\}]\ldots[\{\phi_{k,n}^{Ti},+,g_{k,n}^{Ti}\}]$
7:         **end for**
8:         $SHDATA\_x \leftarrow \bigcap^{Ti} CHRECS\_x_k^{Ti}$ with $k \in \{1,\ldots,m\}$
9:         **if** $(SHDATA\_x \neq \emptyset)$ **then**
10:             store data shared between the threads of block B in the shared memory
11:         **end if**
12:     **end for**
13: **end PROCEDURE**

Algorithm 4 Increase the computational load of a GPU thread

1: **PROCEDURE** INCREASELOAD
Input: access $x_k[i_{k,1}][i_{k,2}]\ldots[i_{k,n}]$ to an n-dimensional array x stored in row-major order
Input: loop nest $L = L_1, L_2, \ldots, L_l$ where both L_1, L_2 are threadified
Input: amount of data Δ to be processed by a GPU thread
Output: a modified program after applying loop tiling under the OpenHMPP programming model
2:     increment the step of the outer loop L_1 to Δ
3:     **for each** scalar variable s in L **do**
4:         promote s to an array $s[\Delta]$
5:         transform reads and writes to s into loops of Δ iterations
6:     **end for**
7: **end PROCEDURE**

GPU Features	*conv3d-cpu*	*conv3d-hmpp1*	*conv3d-hmpp2*	*conv3d-hmpp3*	*sgemm-cpu*	*sgemm-mkl*	*sgemm-hmpp1*	*sgemm-hmpp2*	*sgemm-hmpp3*	*sgemm-hmpp4*	*sgemm-cublas*
Coalescing	-	√	√	√	-	-	√	√	√	√	-
Registers	-		√	√	-	-			√	√	-
Shared Memory	-			√	-	-				√	-

Table 1: GPU features exploited with each variant of CONV3D and SGEMM.

tracted from compute-intensive scientific applications. First, Sect. 6.1 presents the study of the three-dimensional discrete convolution (CONV3D). With this case study we cover stencil codes, which are commonly found in computer simulations, image processing and finite element methods. Next, Sect. 6.2 addresses the simple-precision general matrix multiplication (SGEMM), which is one of the most important linear algebra routines commonly used in engineering, physics or economics.

6.1 Case Study: CONV3D

The three-dimensional discrete convolution operator can be generally written as:

$$output[i][j][k] = \sum_{n_1,n_2,n_3} coef[i][j][k] \cdot input[i-n_1][j-n_2][k-n_3]$$

with *input* being the input 3D-function data, *coef* the filter, and *output* the convoluted data. Consider the implementation shown in Fig. 1a (from now on, denoted as variant *conv3d-cpu*). Three nested loops for_i, for_j and for_k traverse *output* (see lines 4–6). For each element $output[i][j][k]$, four elements in each sense of the three directions of the coordinate axis are taken to perform the convolution with the scalar values *coefx*, *coefy* and *coefz*, respectively. Thus, the temporary variable *tempx* (lines 7–13) stores the weighted sum of nine values of *input* along the *x*-axis, *coefx* being the weight. Similarly, temporaries *tempy* and *tempz* are along the *y*-axis and *z*-axis. Finally, these contributions are accumulated in $output[i][j][k]$ (lines 28–29).

The corresponding KIR, depicted in Fig. 1b, was described in Sect. 4.1. Only the regular reduction $K<output_{28}>$ determines if CONV3D is parallelizable (note that the remaining parts of the KIR are shaded because they represent privatizable temporaries). As the regular reduction diKernel represents conflict-free loop iterations, it can be converted into a forall parallel loop. On the CPU, it can be parallelized using the OpenMP `parallel for` directive.

Table 1 summarizes the GPU features addressed by our locality-aware automatic parallelization technique to generate the same optimal variant as the one written by an expert in GPU programming. The first optimized variant is *conv3d-hmpp1*, which exploits coalescing through loop interchange as follows. A basic OpenHMPP variant could be generated by simply isolating the source code between lines 4–32 of Fig. 1a.

However, Alg. 1 detects that this is not the correct approach due to the non-coalesced accesses. The chrecs associated to the first access to *input* (see line 7 of Fig.1a) are:

$$CHREC\_input_1 = [\{0,+,1\}][\{0,+,1\}][\{0,+,1\}]$$

As explained in Sect. 3, the default OpenHMPP loop threadification policy creates GPU threads for the two outermost loops (*for$_i$* and *for$_j$*). Hence, the instantiated chrecs would be:

$$CHREC\_input_1^{T0} = [\{0\}][\{0\}][\{0,+,1\}], \; CHREC\_input_1^{T1} = [\{0\}][\{1\}][\{0,+,1\}]\ldots$$

These accesses cannot be coalesced by the memory controller (see lines 7–9 of Alg. 1). However, if the loop nest is permuted to *for$_j$*, *for$_k$*, *for$_i$*, the chrecs will be:

$$CHREC\_input_1^{T0} = [\{0,+,1\}][\{0\}][\{0\}], \; CHREC\_input_1^{T1} = [\{0,+,1\}][\{0\}][\{1\}]\ldots$$

Thus,

$$CHREC\_RANGE\_input_{1,3} = \{0\} \cup \{1\}\ldots$$

defines a contiguous range satisfying the condition in line 11 of Alg. 1.

The second optimized variant is *conv3d-hmpp2*. Note that each GPU thread along the threadified *for$_{j,k}$* executes the entire innermost *for$_i$*. Hence, each thread will repeat reads to the array *input* in the *x*-axis in consecutive iterations of *for$_i$* (see lines 7–13 of Fig. 1a). Old values can be stored in local registers reducing the needs of memory bandwidth. Algorithm 2 detects this situation as follows. For illustrative purposes, the chrecs for the first three accesses to array *input* are (see line 3 of Alg. 2):

$$CHREC\_input_1 = [\{0,+,1\}][\{0,+,1\}][\{0,+,1\}]$$
$$CHREC\_input_2 = [\{-1,+,1\}][\{0,+,1\}][\{0,+,1\}]$$
$$CHREC\_input_3 = [\{1,+,1\}][\{0,+,1\}][\{0,+,1\}]$$

For *T0*, the instantiated chrecs are (line 5):

$$CHREC\_input_1^{T0} = [\{0,+,1\}][\{0\}][\{0\}]$$
$$CHREC\_input_2^{T0} = [\{-1,+,1\}][\{0\}][\{0\}]$$
$$CHREC\_input_3^{T0} = [\{1,+,1\}][\{0\}][\{0\}]$$

Thus,

$$\bigcap_{k=1}^{3} CHRECS\_input_k^{T0} = [\{1,+,1\}][\{0\}][\{0\}] \neq \emptyset$$

and, as *input* is only read, copies of already accessed values can be kept in registers for subsequent uses (lines 6–9).

The variant *conv3d-hmpp3* exploits, in addition, the shared memory. Contiguous threads repeat accesses to some positions in the *y,z*-plane of the array *input*. Hence, those values can be stored in the shared memory and be interchanged among the threads of a block. Table 2 focuses on the chrecs corresponding to the first two threads, *T0* and *T1*, and the accesses performed in lines 21–27 of Fig. 1a. Algorithm 3 computes the intersections of all the instantiated chrecs (see lines 5–8 of Alg. 3). As can be observed, the intersection is not empty and some values are stored in the GPU shared memory (lines 9–11). Therefore, the number of accesses to the GPU global memory is reduced significantly.

	T0			T1		
	$1^{st}\,dim$	$2^{nd}\,dim$	$3^{rd}\,dim$	$1^{st}\,dim$	$2^{nd}\,dim$	$3^{rd}\,dim$
$CHRECS\_input_{19}$	$\{0,+,1\}$	$\{0\}$	$\{0\}$	$\{0,+,1\}$	$\{0\}$	$\{1\}$
$CHRECS\_input_{20}$	$\{0,+,1\}$	$\{0\}$	$\{-1\}$	$\{0,+,1\}$	$\{0\}$	$\{0\}$
$CHRECS\_input_{21}$	$\{0,+,1\}$	$\{0\}$	$\{1\}$	$\{0,+,1\}$	$\{0\}$	$\{2\}$
$CHRECS\_input_{22}$	$\{0,+,1\}$	$\{0\}$	$\{-2\}$	$\{0,+,1\}$	$\{0\}$	$\{-1\}$
$CHRECS\_input_{23}$	$\{0,+,1\}$	$\{0\}$	$\{2\}$	$\{0,+,1\}$	$\{0\}$	$\{3\}$
$CHRECS\_input_{24}$	$\{0,+,1\}$	$\{0\}$	$\{-3\}$	$\{0,+,1\}$	$\{0\}$	$\{-2\}$
$CHRECS\_input_{25}$	$\{0,+,1\}$	$\{0\}$	$\{3\}$	$\{0,+,1\}$	$\{0\}$	$\{4\}$
$CHRECS\_input_{26}$	$\{0,+,1\}$	$\{0\}$	$\{-4\}$	$\{0,+,1\}$	$\{0\}$	$\{-3\}$
$CHRECS\_input_{27}$	$\{0,+,1\}$	$\{0\}$	$\{4\}$	$\{0,+,1\}$	$\{0\}$	$\{5\}$

Table 2: Chrecs for the accesses in lines 21–27 of Fig. 1a (CONV3D).

```
1  int i, j, l, m, n, k;
2  float A[m][k], B[k][n], C[m][n];
3  float alpha, beta, prod;
4
5  for (i = 0; i < m; i++) {
6    for (j = 0; j < n; j++) {
7      prod = 0;
8      for (l = 0; l < k; l++) {
9        prod += A[i][l] * B[l][j];
10     }
11     C[i][j] = alpha * prod + beta * C[i][j];
12   }
13 }
```


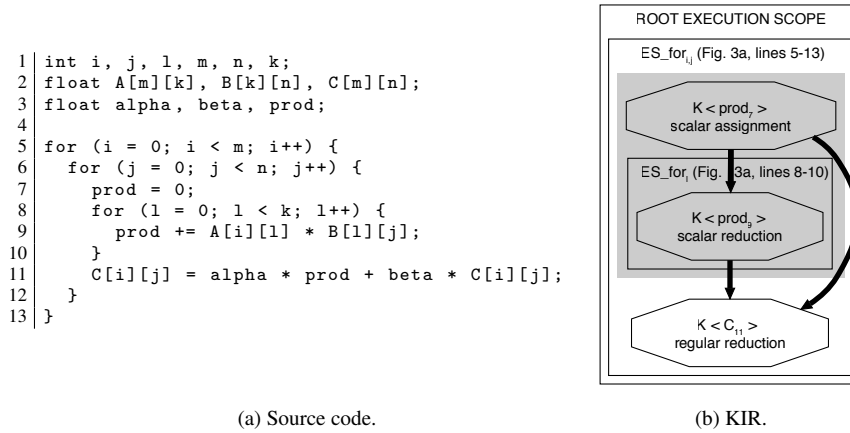(a) Source code.   (b) KIR.

Fig. 3: The simple-precision general matrix multiplication (SGEMM).

6.2 Case Study: SGEMM

The simple-precision general matrix multiplication from the BLAS library [7] performs the matrix operation:

$$C = \alpha \cdot A \times B + \beta \cdot C$$

where A, B, C are $m \times k$, $k \times n$ and $m \times n$ matrices, respectively, and α, β are the scale factors for $A \times B$ and C. Figure 3a shows an implementation with two nested loops for_i and for_j that traverse the matrix C row by row (see lines 5–6). Each matrix position $C[i][j]$ is computed with the dot product between the ith row of matrix A and the jth column of B. The dot product is temporarily stored in the scalar variable $prod$ (lines 7–10).

The KIR shown in Fig. 3b captures the semantics of Fig. 3a as follows. Loops for_i and for_j are perfectly nested, thus a unique execution scope $ES\_for_{i,j}$ is created. $K<prod_7>$ represents the initialization of the temporary variable $prod$ at line 7.

	not instantiated		T0		T1	
	$1^{st}dim$	$2^{nd}dim$	$1^{st}dim$	$2^{nd}dim$	$1^{st}dim$	$2^{nd}dim$
CHRECS_A	$\{0,+,1\}$	$\{0,+,1\}$	$\{0\}$	$\{0,+,1\}$	$\{0\}$	$\{0,+,1\}$
CHRECS_B	$\{0,+,1\}$	$\{0,+,1\}$	$\{0,+,1\}$	$\{0\}$	$\{0,+,1\}$	$\{1\}$
CHRECS_C	$\{0,+,1\}$	$\{0,+,1\}$	$\{0\}$	$\{0\}$	$\{0\}$	$\{1\}$

Table 3: Chrecs for the accesses to arrays A, B and C in SGEMM.

The computation of the dot product is contained in for_l. Hence, the scalar reduction $K{<}prod_9{>}$ is attached to ES_for_l. Finally, $K{<}C_{11}{>}$ is a regular reduction that updates the previous value stored in $C[i][j]$. As *prod* is a privatizable scalar variable, the parts of the KIR referring to its computations are shaded to be omitted in the discovery of parallelism. Thus, only $K{<}C_{11}{>}$ needs to be considered to decide if the source code is parallelizable. As mentioned in Sect. 4.1, a regular reduction diKernel represents conflict-free loop iterations and it is therefore parallelizable.

From the point of view of the locality, the challenge of SGEMM is to handle the tradeoff between opposite array traversals efficiently: row-major for C and A, and column-major for B. On the CPU, the general solution is to apply loop tiling: matrices are computed in small tiles to keep data in the cache memory. This approach can be also applied on the GPU using the shared memory as cache and being aware of coalescing.

The first variant of SGEMM is the sequential code shown in Fig. 3a (*sgemm-cpu*). In addition, we have selected the `cblas_sgemm` function of the non-clustered, threaded part of the Intel MKL library [15] to build the *sgemm-mkl* variant.

The first OpenHMPP variant is *sgemm-hmpp1*. It is trivially built by offloading to the GPU the same code as *sgemm-cpu*. Table 3 shows the chrecs for this variant, which are analyzed by Alg. 1 as follows. Regarding A, all the threads of a warp have the same chrecs and thus access the same memory position (see line 14 of Alg. 1). Regarding B, coalescing is maximized because the chrecs of the first dimension are the same while the chrecs of the second one define a contiguous range (lines 10–12). Finally, the same situation holds for C and thus accesses are coalesced.

The second OpenHMPP variant is *sgemm-hmpp2*. Algorithm 4 transforms the source code of Fig. 3a as follows. The scalar variable *prod* is promoted to an array $prod[\Delta]$, and thus a new loop for_t is created to enclose all its definitions and uses (see lines 3–6 of Alg. 4). The step of the outer for_i is incremented by Δ, and uses of the loop index i inside for_t are replaced by $i+t$.

The third OpenHMPP variant is *sgemm-hmpp3*. For the reasons mentioned in the last paragraph of Sect. 5.2, our technique first performs loop fission in the new for_t giving place to for_{t1} (*prod* initialization), for_{t2} (dot product between the row of A and the column of B), and for_{t3} (computation with the old value of C). Next, `fullunroll` directives are inserted in for_{t1} and for_{t3}. In order to fully unroll for_{t2}, it is first interchanged with for_l. This way, the GPU compiler is able to store $prod[\Delta]$ in registers.

The fourth OpenHMPP variant is *sgemm-hmpp4*. Algorithm 2 presented a method to store reused data in registers. In this case, as the number of registers is finite and

the previous transformation in *sgemm-hmpp3* increased register pressure, we have used the shared memory to store slices of *B*.

Finally, the last variant is *sgemm-cublas*, the implementation provided by the NVIDIA CUBLAS library [24]. CUBLAS has been designed assuming a column-major order, thus a transformation is needed before and after calling the library.

7 Performance Evaluation

Two NVIDIA-based heterogeneous systems were used to carry out our experiments. The first one is `nova`, the CAPS Compute Lab, based on Tesla S1070 (compute capability 1.3 —Tesla architecture—). The GPU contains 30 multiprocessors with 8 cores each, for a total of 240 CUDA cores at 1.30 GHz. The total amount of global memory is 4 GB at 800 MHz. Each block (of up to 512 threads) can access 16 KB of shared memory and 16384 registers. The accelerator is connected to a host system consisting of 2 Intel Xeon X5560 quad-core processors at 2.80 GHz and 12 GB of memory.

The second system is `pluton`, the cluster of the Computer Architecture Group at the University of A Coruña, based on Tesla S2050 (compute capability 2.0 —Fermi architecture—). The GPU contains 14 multiprocessors with 32 cores each, for a total of 448 CUDA cores at 1.15 GHz. The total amount of global memory is 3 GB at 1546 MHz with ECC disabled. Each block (of up to 1024 threads) can access 48 KB of shared memory, 16 KB of L1 cache and 32768 registers. The amount of L2 cache is 768 KB. The accelerator is connected to a host system consisting of 2 Intel Xeon X5650 six-core processors at 2.66 GHz and 12 GB of memory.

7.1 Performance Evaluation of CONV3D

We have run the 216 experiments corresponding to all matrix sizes for *sizex*, *sizey* and *sizez* values in 128, 256, 384, 512, 640 and 768. In each experiment, we have measured GFLOPS for all CONV3D variants. As can be viewed in Table 4, our experiments revealed that the obtained GFLOPS did not show a significant variation with the dimensions of the tested matrices. This is due to the fact that the limiting factor in the performance of this test case is the memory access bandwidth. For all tested sizes, even the smallest ones, more than a 50 % GPU occupancy is achieved (which is a good value for this sort of codes [35]).

Figure 4 depicts the performance evaluation of the CPU and the GPU-accelerated variants on our experimental platforms. The offloading of the computations on the GPU, with a loop interchange (*conv3d-hmpp1*), gets a speedup of 5.43x on `nova` and 15.27x on `pluton`. Note the big step in performance improvement between *conv3d-hmpp2* and *conv3d-hmpp3* due to the use of the shared memory: 3.35x on `nova` and 1.62x on `pluton`. The improvement is less impressive on `pluton` because of the cache memories present in the Fermi cards that partially cover the functionality exploited by our locality-aware automatic technique.

7.2 Performance Evaluation of SGEMM

We have run the 6859 experiments corresponding to all matrix sizes for m, n, and k values in 128, 256, 384, 512, 640, 768, 896, 1024, 1152, 1280, 1408, 1536, 1664, 1792, 1920, 2048, 4096, 6144 and 8192. In each experiment, we have measured GFLOPS for all SGEMM variants.

Table 5 and Fig. 5 present the performance evaluation of the CPU and the GPU-accelerated variants. On average, *sgemm-mkl* is better than *sgemm-hmpp1*: 5.26x on `nova` and 1.96x on `pluton`. However, for most of the combinations of $m, n, k < 2048$, *sgemm-hmpp1* is better than *sgemm-mkl* (up to 31.50x for $m = 256$, $n = 128$ and $k = 512$ on `pluton`). Hence, in contrast to CONV3D, the performance of SGEMM varies significantly for different matrix sizes (as can be observed in the minimum, average and maximum columns of Table 5) and the simple use of the GPU does not always improve the best CPU variant. For the majority of the tested sizes, *sgemm-hmpp2* slightly improves *sgemm-hmpp1* on `nova`, but not on `pluton`. This is due to the fact that accesses to $prod[\Delta]$ in *sgemm-hmpp2* read and write from the GPU memory and not from the registers. The performance improvement of *sgemm-hmpp3* with respect to *sgemm-hmpp1* is bigger (1.79x on `nova` and 2.03x on `pluton`) because the transformation allows the GPU compiler to store $prod[\Delta]$ in registers. However, the biggest improvement factor is the usage of the shared memory, as can be observed in the *sgemm-hmpp4* results.

The best variant on `nova` for the majority of cases is *sgemm-cublas*. However, it is only 10 % better than *sgemm-hmpp4* on average. In fact, *sgemm-hmpp4* is the best for $k < 1024$, and *sgemm-mkl* is the best for $m, n = 128$ with $512 \leq k \leq 1792$. Regarding average performance on `pluton`, *sgemm-cublas* is clearly the best, being 36 % faster than *sgemm-hmpp4*. Variant *sgemm-cublas* is only bested by *sgemm-hmpp4* for $m, n \in \{128, 256\}$, and by *sgemm-mkl* for $m, n = 128$ with $k > 1152$. Nevertheless, we have demonstrated that a variant automatically generated by applying basic loop transformations to the original sequential code is competitive with the highly optimized NVIDIA's CUBLAS implementation.

8 Related Work

In this paper, we have introduced a new technique to tune the performance of automatically generated GPU parallel code exploiting locality through standard loop transformations. This technique has been successfully applied to two representative case studies, namely CONV3D and SGEMM. There exist in the literature previous works about the optimization of the execution on the GPU of these case studies (for instance, [10] and [38] for CONV3D, [18] for SGEMM) that are based on templates or domain-specific languages. In contrast, our approach is devoted to be general.

Similar efforts to automatically generate code for GPUs from a sequential program are being developed. Parallware for OpenACC [3], in alpha state, also employs technology based on diKernels. However, unlike our approach, it does not support locality exploitation.

GFLOPS	nova			pluton		
	min	avg	max	min	avg	max
conv3d-cpu	1.42	2.54	2.72	-	-	-
conv3d-hmpp1	13.42	13.78	14.03	31.40	38.74	46.95
conv3d-hmpp2	18.76	19.80	20.28	51.08	67.47	78.79
conv3d-hmpp3	41.50	66.32	70.60	97.02	109.48	118.75

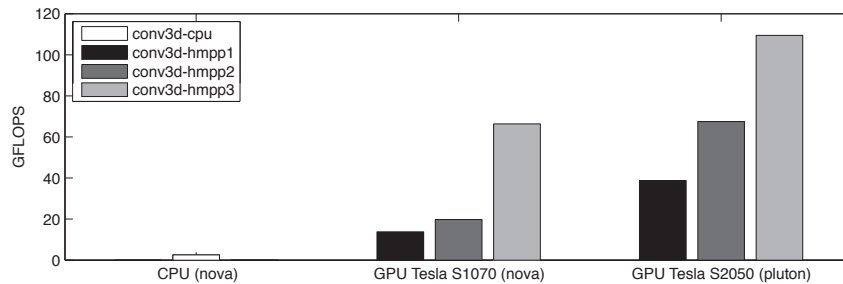Table 4: Minimum, average and maximum GFLOPS of CONV3D variants.

Fig. 4: Average GFLOPS of CONV3D variants.

GFLOPS	nova			pluton		
	min	avg	max	min	avg	max
sgemm-cpu	0.13	0.51	1.40	-	-	-
sgemm-mkl	1.43	114.99	183.70	-	-	-
sgemm-hmpp1	8.10	21.85	27.41	15.41	58.77	79.22
sgemm-hmpp2	3.33	21.80	27.69	3.72	51.57	78.83
sgemm-hmpp3	6.93	39.19	64.81	12.04	119.56	134.74
sgemm-hmpp4	7.12	295.45	354.46	9.20	357.38	420.63
sgemm-cublas	71.30	325.91	370.12	38.78	486.41	650.16

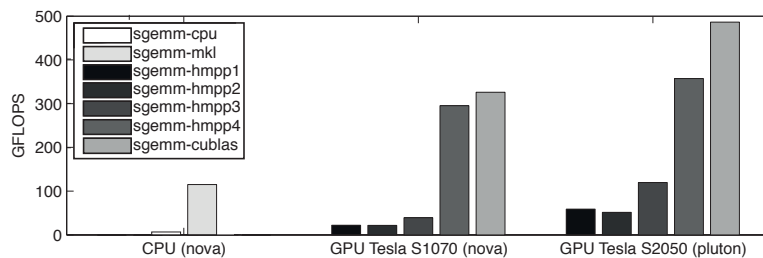Table 5: Minimum, average and maximum GFLOPS of SGEMM variants.

Fig. 5: Average GFLOPS of SGEMM variants.

There also exist other active approaches based on the polyhedral model. C-to-CUDA [6], based on PLUTO [9], looks for a region as large as possible and transforms memory accesses to be coalesced (using the shared memory if it is not possible). The shared memory is also employed to store the arrays that are reused in the same thread, but the reuse of data between the threads of a block is not considered.

PPCG [33] also searches for the largest possible region of code and the parallelization strategy evolved from PLUTO. It applies an elaborated policy for the use of the memory hierarchy grouping array references to copy parts of the global memory. Reused data are placed in registers. If there is any reuse or the original accesses were not coalesced, then it places the data in the shared memory.

Par4All [14] uses abstract interpretation for array regions, which also involves polyhedra. It treats each loop nest independently, generating a CUDA kernel for each one. Par4All does not consider the exploitation of reuse in the registers or the shared memory: all accesses are performed directly on the global memory. However, it performs powerful inter-procedural analysis on the input code.

Jablin et al. [16, 17] propose a framework that automatically generates pipeline parallelizations and provides software-only shared memory. The memory allocation system ensures that addresses of equivalent allocation units on the CPU and GPU are equal, relieving the runtime library of the burden of translation and communication optimization. The compiler inserts appropriate calls into the original program. The pipeline parallelization technique exploits the fact that GPUs have abundant parallel computing resources but communication between them can be very expensive. If the loaded values were constant, each of the threads could execute the load redundantly, reducing communication overhead at the expense of computational efficiency.

In summary, most approaches partially exploit the GPU memory hierarchy and generate low-level, difficult to understand, CUDA code. In contrast, our proposal based on OpenHMPP directives provides understandable and portable code easing the interaction between programmers and application-domain experts. Additionally, with the inclusion of auto-tuning techniques [12], OpenHMPP has demonstrated to be able to obtain even better performance than hand-coded CUDA/OpenCL codes.

9 Conclusions and Future Work

This paper has introduced a new KIR-based locality-aware automatic parallelization technique that targets GPU-based heterogeneous systems. Our proposal is devoted to exploit locality in the complex GPU memory hierarchy in order to generate efficient code. It takes into account the most impacting GPU programming features: loop threadification, thread grouping, coalesced access to global memory, and maximum usage of registers and shared memory. We have successfully applied this technique to two representative case studies extracted from compute-intensive scientific applications (namely, CONV3D, the three-dimensional convolution, and SGEMM, the simple-precision general matrix multiplication). We have modeled the accesses to n-dimensional arrays with chains of recurrences. This algebraic formalism allowed us to analyze the interactions between the memory accesses performed by the GPU threads in a loop nest. The usage of OpenHMPP directives enabled a great under-

standability and portability of the generated GPU code. The performance evaluation on NVIDIA GPUs (with two different core architectures) has corroborated the effectiveness of our approach.

As future work, we will design and implement a new automatic partitioning algorithm of the KIR to handle the interactions between computations in full-scale applications. Auto-tuning approaches will be incorporated to select the variant with best performance on a given hardware architecture. We will also test our proposals with a larger benchmark suite and on other manycore accelerators.

References

1. Andión, J.M., Arenaz, M., Rodríguez, G., Touriño, J.: A Novel Compiler Support for Automatic Parallelization on Multicore Systems. Parallel Comput. **39**(9), 442–460 (2013)
2. Andrade, D., Arenaz, M., Fraguela, B.B., Touriño, J., Doallo, R.: Automated and Accurate Cache Behavior Analysis for Codes with Irregular Access Patterns. Concurr. Comput.: Pract. Exper. **19**(18), 2407–2423 (2007)
3. Appentra Solutions: Parallware for OpenACC. http://www.appentra.com/products/parallware/ (Accessed 31 January 2015)
4. Arenaz, M., Touriño, J., Doallo, R.: Compiler Support for Parallel Code Generation Through Kernel Recognition. In: Proc. of the 18th International Parallel and Distributed Processing Symposium (IPDPS), Santa Fe, NM, USA, p. 79b. IEEE (2004)
5. Arenaz, M., Touriño, J., Doallo, R.: XARK: An Extensible Framework for Automatic Recognition of Computational Kernels. ACM Trans. Program. Lang. Syst. **30**(6), 32:1–32:56 (2008)
6. Baskaran, M.M., Ramanujam, J., Sadayappan, P.: Automatic C-to-CUDA Code Generation for Affine Programs. In: Proc. of the 19th International Conference on Compiler Construction (CC), Paphos, Cyprus, *LNCS*, vol. 6011, pp. 244–263. Springer (2010)
7. BLAS: Basic Linear Algebra Subprograms. http://www.netlib.org/blas/ (Accessed 31 January 2015)
8. Bodin, F., Bihan, S.: Heterogeneous Multicore Parallel Programming for Graphics Processing Units. Scientific Programming **17**(4), 325–336 (2009)
9. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In: Proc. of the 29th Conference on Programming Language Design and Implementation (PLDI), Tucson, AZ, USA, pp. 101–113. ACM (2008)
10. Christen, M., Schenk, O., Burkhart, H.: Automatic Code Generation and Tuning for Stencil Kernels on Modern Shared Memory Architectures. Computer Science - R&D **26**(3-4), 205–210 (2011)
11. Eigenmann, R., Hoeflinger, J., Li, Z., Padua, D.A.: Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs. In: Proc. of the 4th International Workshop on Languages and Compilers for Parallel Computing (LCPC), Santa Clara, CA, USA, *LNCS*, vol. 589, pp. 65–83. Springer (1992)
12. Grauer-Gray, S., Xu, L., Searles, R., Ayalasomayajula, S., Cavazos, J.: Auto-Tuning a High-Level Language Targeted to GPU Codes. In: Proc. of Innovative Parallel Computing (InPar), San Jose, CA, USA, pp. 1–10. IEEE (2012)
13. Han, T.D., Abdelrahman, T.S.: hiCUDA: High-Level GPGPU Programming. IEEE Trans. Parallel Distrib. Syst. **22**(1), 78–90 (2011)
14. HPC Project: Par4All. http://www.par4all.org/ (Accessed 31 January 2015)
15. Intel Corporation: Intel Math Kernel Library. http://software.intel.com/intel-mkl/ (Accessed 31 January 2015)

16. Jablin, T.B., Jablin, J.A., Prabhu, P., Liu, F., August, D.I.: Dynamically Managed Data for CPU-GPU Architectures. In: Proc. of the 10th International Symposium on Code Generation and Optimization (CGO), San Jose, CA, USA, pp. 165–174. ACM (2012)

17. 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: Proc. of the 32nd Conference on Programming Language Design and Implementation (PLDI), San Jose, CA, USA, pp. 142–151. ACM (2011)

18. Kurzak, J., Tomov, S., Dongarra, J.: Autotuning GEMM Kernels for the Fermi GPU. IEEE Trans. Parallel Distrib. Syst. **23**(11), 2045–2057 (2012)

19. Larsen, E.S., McAllister, D.: Fast Matrix Multiplies using Graphics Hardware. In: Proc. of the 14th International Conference on High Performance Computing, Networking, Storage and Analysis (SC), Denver, CO, USA, p. 55. ACM (2001)

20. Lee, S., Eigenmann, R.: OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In: Proc. of the 23rd International Conference on High Performance Computing, Networking, Storage and Analysis (SC), New Orleans, LA, USA, pp. 1–11. IEEE (2010)

21. Lee, S., Vetter, J.S.: Early Evaluation of Directive-Based GPU Programming Models for Productive Exascale Computing. In: Proc. of the 25th International Conference on High Performance Computing, Networking, Storage and Analysis (SC), Salt Lake City, UT, USA, pp. 23:1–23:11. IEEE (2012)

22. Novatte Pte. Ltd.: CAPS Compilers. http://www.novatte.com/component/content/article/126-products/hpcclusters/301-caps-compilers-for-cuda-and-opencl/ (Accessed 31 January 2015)

23. NVIDIA Corporation: Cg Toolkit. http://developer.nvidia.com/Cg/ (Accessed 31 January 2015)

24. NVIDIA Corporation: CUBLAS Library. https://developer.nvidia.com/cublas/ (Accessed 31 January 2015)

25. NVIDIA Corporation: CUDA C Best Practices Guide. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/ (Accessed 31 January 2015)

26. NVIDIA Corporation: CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/ (Accessed 31 January 2015)

27. OpenHMPP Consortium: OpenHMPP Concepts & Directives. http://en.wikipedia.org/wiki/OpenHMPP (Accessed 31 January 2015)

28. OpenMP Architecture Review Board: OpenMP Application Program Interface (Version 4.0). http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf (Accessed 31 January 2015)

29. Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., Phillips, J.: GPU Computing. Proc. of the IEEE **96**(5), 879–899 (2008)

30. The Khronos Group Inc.: The OpenCL Specification (Version 2.0). http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf (Accessed 31 January 2015)

31. The Khronos Group Inc.: The OpenGL Shading Language (Version 4.50). https://www.opengl.org/registry/doc/GLSLangSpec.4.50.pdf (Accessed 31 January 2015)

32. The OpenACC Standards Group: The OpenACC Application Programming Interface (Version 2.0a). http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf (Accessed 31 January 2015)

33. Verdoolaege, S., Juega, J.C., Cohen, A., Gómez, J.I., Tenllado, C., Catthoor, F.: Polyhedral Parallel Code Generation for CUDA. ACM Trans. Archit. Code Optim. **9**(4), 54:1–54:23 (2013)

34. Viñas, M., Lobeiras, J., Fraguela, B.B., Arenaz, M., Amor, M., García, J.A., Castro, M.J., Doallo, R.: A Multi-GPU Shallow-Water Simulation with Transport of Contaminants. Concurr. Comput.: Pract. Exper. **25**(8), 1153–1169 (2013)

35. Volkov, V.: Better Performance at Lower Occupancy. In: Proc. of the 2010 GPU Technology Conference (GTC), San Jose, CA, USA. NVIDIA (2010)

36. Wolfe, M.: Implementing the PGI Accelerator Model. In: Proc. of the 3rd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU), Pittsburgh, PA, USA, pp. 43–50. ACM (2010)

37. Zima, E.: Simplification and Optimization of Transformations of Chains of Recurrences. In: Proc. of the 1995 International Symposium on Symbolic and Algebraic Computation (ISSAC), Montreal, Canada, pp. 42–50. ACM (1995)

38. Zhang, Y., Mueller, F.: Autogeneration and Autotuning of 3D Stencil Codes on Homogeneous and Heterogeneous GPU Clusters. IEEE Trans. Parallel Distrib. Syst. **24**(3), 417–427 (2013)