Source-to-source code translator: OpenMP C to CUDA

Gabriel Noaje, Christophe Jaillet, Michaël Krajecki
CReSTIC - SysCom, University of Reims Champagne-Ardenne, Reims, France
Email: {gabriel.noaje, christophe.jaillet, michael.krajecki}@univ-reims.fr

Abstract—In recent years hardware accelerators have become a full part of the HPC domain as their peak performance has increased steadily. Although their programmability has greatly improved and different tools have been developed to smooth down their learning curve, porting legacy code to the new programming models could reveal itself a cumbersome and time-consuming process. However a large amount of code benefits from the multicore architectures using either a shared memory model or a distributed model. Our goal was to allow developers to benefit from the simplicity of OpenMP code and at the same time permitting their code to be executed on GPUs manycore architectures. Thus we propose a source-to-source compiler able to automatically transform an OpenMP C code into a CUDA code, while maintaining a human readable version of the code that can be further analyzed or optimized. Using the OMPi compiler as a base ground, we implemented the “pragma omp parallel for” transformation along with data visibility clauses. The generated code is fully NVIDIA CUDA compliant and can be compiled using the nvcc compiler. In this paper we present the entire transformation process, starting from the pragma split-up and kernel generation, passing through the data visibility clauses management and ending with the device memory management and kernel launch system.

Index Terms—compiler, code transformation, CUDA, OpenMP

I. INTRODUCTION

The past decade has witnessed a proliferation of powerful parallel and distributed systems and an ever increasing demand for high performance computing. The GPGPU (General-Purpose Computing on Graphics Processing Units) domain [1] started as a promising alternative to this increasing urge for speed. Applications were demanding more and more computing power as their complexity and data involved in the process were growing.

These days we are entering the golden age of GPU computing [2]. Unified graphics and computing processors are the norm today allowing full programmability, thus creating the new domain of GPU Computing [3]. Current open architectures allow non-graphics developers to access the GPU parallel computing capabilities through a new data-parallel programming model without requiring casting to graphics APIs [4].

Hundreds of researchers are working to harness this computing power. However the main drawback still remains the programming effort needed to exploit them. When developing a new code that needs to be executed on the GPU, one has to think in terms of GPU parallel programming and in respect of the underlying architecture. However HPC domain is a lot older than GPU computing and billions of lines of legacy code are using older parallel programming models like OpenMP or MPI. With this aspect in mind several solutions, both open-source and commercial, emerged in the ecosystem with the aim to allow porting legacy applications to GPU-enabled systems with a minimal effort.

Following the same idea we present in this paper a source-to-source compiler capable of transforming an OpenMP annotated code into a fully compatible CUDA C application. We wanted the entire process to be as transparent as possible and the generated code to be in a human readable format. Having a user-friendly code in the end allows the developer to further analyze and tweak the application as needed.

The structure of the paper is the following. We start by briefly introducing the OpenMP programming paradigm and the CUDA programming framework with their particularities. Further on, we compare some major approaches that allow us to generate GPU-enabled code from a standard code. The main part of our paper details our approach to transform an OpenMP annotated code into a fully human readable CUDA C compliant code. Finally we conclude and present the main directions for future work.

II. OPENMP VS CUDA

According to Flynn’s classification [5], common parallel architectures are based on the MIMD model and we differentiate, depending on the memory location: shared memory systems using a multithreading programming model and distributed memory systems using a message passing programing model.

A. OpenMP

OpenMP is an implementation of multithreading, a parallel execution scheme whereby the master thread (a series of instructions executed consecutively) forks a specified number of slave threads and a task is divided among them [6]. OpenMP is primarily designed for shared memory multiprocessors, using the SPMD model (Single Program, Multiple Data Stream): all the processors are able to directly access all the memory in the machine, through a logically direct connection. Programs will be executed on one or more processors that share some or all of the available memory. The program is typically executed by multiple independent threads that share data, but may also have some additional private memory zones.

OpenMP allows incremental parallelization of existing application through extensions to an existing sequential lan-
guage; this has the advantage of providing parallel extensions within a familiar programming environment [7].

B. NVIDIA architecture

GPUs fit none of the traditional execution models proposed by Flynn taxonomy, since their architecture is quite different even from the SIMD execution model. As NVIDIA GPUs are widely available and their programming environment reached a stable version, we decided to focus on these architectures. Hence, we will present a brief introduction of the latest NVIDIA architecture, codenamed Fermi.

As a physical layout, NVIDIA GPUs are organized as Streaming Multiprocessors (SM) with simple scalar processors (SP) on chip (Figure 1). Inside the device, threads are able to access data from multiple memory spaces:

- **local memory**: per-thread, private, for temporary data (implemented in external DRAM);
- **shared memory**: for low-latency access to data shared by cooperating threads in the same SM (implemented on chip);
- **global memory**: for data shared by all threads of a computing application (implemented in external DRAM).

In addition to these memories each SM has an important number of registers used to store instructions operands (32768 for the latest devices of compute capability 2.x).

Opposed to the SIMD execution model (Single Instruction Stream, Multiple Data Stream) used for general data-parallel programming, the NVIDIA model is SIMT (Single Instruction Stream, Multiple Threads): the kernel (code execution unit) is executed simultaneously on all SMs by independent blocks of threads; each thread, assigned to a single processor, executes within its own execution environment (register state and instruction address), but they run the same instruction at a time. In contrast with SIMD vector machines, SIMT enables programmers to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads.

To efficiently execute hundreds of threads in parallel, the SM is hardware multithreaded, meaning that it manages and executes up to 1536 concurrent threads in hardware (compute capability 2.x), by blocks of 1024 threads maximum, with zero scheduling overhead [8].

- Once a block has its threads assigned to a SM, it is further divided by the SIMT multithreaded instruction unit into 32-thread units called **warps**\(^1\).
- Each SM features two warp schedulers and two instruction dispatch units, allowing two warps to be chosen from a pool of maximum 48 warps to be executed concurrently.

\(^1\)The warps are not part of the CUDA language definition and even though programmers can generally ignore warp execution for functional correctness and think of programming one thread, they can greatly improve performance by having threads in a warp execute the same code path and access memory in nearby addresses.

- Inside a warp, threads may execute differently after branching, but by teams with similar property: those having had the same branching evaluation.

![Figure 1. GPU architecture](image.png)

C. CUDA

On February 2007, NVIDIA Corporation has released the initial version of the **CUDA SDK**. It provides two APIs: the C runtime for CUDA which is delivered through the cudart dynamic library and the CUDA driver API which is delivered through nvidia dynamic library.

CUDA provides all the means of a parallel programming model with the particularity of two types of shared memory: the on-chip shared memory that can be shared by threads of a block executing on a SM and the global memory accessed independently by the blocks running on the GPU. Due to their execution model and memory hierarchy, GPUs support two-level of parallelism [9]:

- An outer fully-parallel `doall` loop level that is supported at the grid level with no synchronization. Thread blocks in a grid are required to execute independently. It must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores, enabling programmers to write scalable code.
- An inner synchronous loop level at the level of thread blocks where all threads within a block can cooperate and synchronize.

Compared with other GPU programming frameworks such as OpenCL, the CUDA language can be considered high-level since it does not need all the low level initializations and transformations. However, dealing with the entire programming framework in order to manage device accesses, memory
transfers between the device can still be a tedious work, on the other hand the OpenMP programming model allows a simple and incremental parallelization. Thus, we chose to create a source-to-source translator for OpenMP C code to CUDA.

III. RELATED WORK

Compilation consists in transforming a source code into a binary program; the process may include different optimization steps of the source code. The advantage of source-to-source transformation is that the generated code can be further improved manually or used as a starting point for other developments.

The idea of a source-to-source compiler is rather old [10], many such compilers have been around for a long time. The goal of the paper is to present our approach for an OpenMP to CUDA C source-to-source translator leading to a human readable code. Thus in this section we compare different source-to-source transformation solutions in order to give an overview of the available possibilities, highlight some particular aspects and introduce the main challenges of such an approach. Table I presents a brief comparison matrix for some source-to-source translators, belonging to open-source projects (ROSE, LLVM, Cetus, Par4All) or commercial ones (PGI, HMPP). The main comparison axes are the input code, the generated code (standard or GPU specific) and the hardware target for the final code.

Our main objectives in respect with the current solutions were the following:

- create an open-source tool
- transform OpenMP standard code into CUDA code
- support NVIDIA GPU architectures (high availability)

IV. FROM A PRAGMA OMP PARALLEL FOR TO A CUDA KERNEL

In this section we present the transformations we brought to the OMPi compiler to support a source-to-source code generation from an OpenMP C code to a CUDA C code. OMPi [13] is an experimental on-going project, developed at the Department of Computer Science of the University of Ioannina. OMPi aims at providing a compiler for the OpenMP programming model. It is a source-to-source translator that takes C code with OpenMP compliant directives and produces equivalent multithreaded C code ready for execution on a shared memory multiprocessor system. OMPi uses POSIX threads for portability, but its architecture allows targeting other thread libraries, as well.

First we introduce the OMPi compiler with the transformation steps involved in generating POSIX threads code and then we present the significant steps involved in transforming OpenMP C code into CUDA.

A. OMPi compiler

The compilation process involved in OMPi consists of several transformation steps to the source code which produce the final multithreaded C file. Essentially, a parallel region is transformed as follows:

- the code composing the parallel region is moved to a new function which will be called later by all threads;
- the original code is replaced by a fragment of code that generates the thread pool that will call the newly created function;
- private variables are re-declared inside the new function;
- shared variables that are global in scope need no special treatment since they are by nature available to all threads;
- non-global shared variables are declared locally as pointers, initialized to point the original variables and variable references are replaced by appropriate pointer references.

The OMPi translator uses a scanner/parser written entirely in C (flex/bison) which generates an abstract syntax tree (AST) representing the source code. The AST is accompanied by a hashed symbol table which reduces the overhead implied by working directly with strings. The transformation process consists exclusively in transformations in the AST which in the end is traversed to generate the final code.

More technically, AST is built based on a set of 7 nodes structures: astexpr, astspec, astdecl, aststmt, ompec on, ompdir and ompclause, used correspondingly for expressions, the specifier part of a declaration, the declarator part of a declaration, statements, OpenMP constructs, OpenMP directives and OpenMP clauses.

One should note that the AST is not annotated, so for example identifiers met inside expressions do not have any link to their declarations. The OMPi symbol table is “imperative” (non-persistent), thus if one needs to examine, analyze, alter or transform the AST, there is no way of having the information needed unless the symbol table is recreated while traversing the AST, exactly the same way it was built-up when parsing the file.

Our goal was to extend the OMPi translator in order to generate a CUDA C compliant code capable of being executed on NVIDIA GPUs starting from an OpenMP annotated code. In order to accomplish this task we modified all the stages involved in the code transformation with the aim of supporting the CUDA C language. A brief list of the modifications that have been operated is detailed bellow and will be expanded in the following subsections:

- the scanner and parser have been modified in order to support the special function and variable type qualifiers;
- the pragma omp parallel for transformation routine generates a kernel function instead of a simple function;
- all implied CUDA memory copies to/from GPU are added to the initial code along with the kernel call functions;
- private and shared variables are transferred to the
Table I
COMPARISON MATRIX FOR A SOME SOURCE-TO-SOURCE SOLUTIONS WITH PARALLEL SUPPORT

<table>
<thead>
<tr>
<th>Input code</th>
<th>Open-source projects</th>
<th>Commercial projects</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP-like annotated native native native native</td>
<td>OpenMP-like annotated</td>
<td>OpenMP-like annotated</td>
</tr>
<tr>
<td>standard C code</td>
<td>linked with OpenMP libraries through GCC framework linking</td>
<td>linked with own libraries</td>
</tr>
<tr>
<td>Output code</td>
<td>standard C code</td>
<td>standard C code</td>
</tr>
<tr>
<td>GPU code</td>
<td>CUDA</td>
<td>CUDA</td>
</tr>
<tr>
<td>other</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware support</td>
<td>CPU</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>NVIDIA</td>
<td>NVIDIA</td>
</tr>
<tr>
<td></td>
<td>CPU+GPU</td>
<td>X</td>
</tr>
</tbody>
</table>

GPU and treated accordingly with respect to their visibility and the GPU memory hierarchy.

B. Scanner / parser modifications

The CUDA parallel programming model is designed so as to transparently scale applications parallelism to leverage the increasing number of processor cores. In order to maintain a low learning curve for the programmers familiar with the standard C language, the CUDA C language exposes the GPU programmable features using a minimal set of language extensions backed-up by a runtime library. The entire list of extensions is presented in Appendix B of the NVIDIA CUDA Programming Guide [17], but the current version of our compiler only features a subset of them.

The scanner and parser of OMPi are fully compliant with the C99 standard. Thus, we only had to take into account the extensions brought by CUDA C. The most important ones are the function type qualifiers and variable type qualifiers.

- Using one of __device__, __global__ or __host__, a CUDA programmer can instruct the compiler to generate a device function, a kernel or a host function.
- Each of the following qualifiers specifies the memory location of a variable on the device: __device__ for global memory variables, __constant__ for constant memory variables, __shared__ for shared memory variables.
- Built-in variables specifying the grid and block dimensions and the block and thread indices are not necessarily added to the parser since they are only generated in the transformation process and will be treated like any other variable.
- In the same manner the built-in vector types are not explicitly added to the grammar, but they are used in the code generation using their C user defined structure taken from the CUDA headers.

We made the above transformations by modifying the lex/flex and yacc/bison files, adding the required rules.

C. Transforming a “pragma omp parallel for” into a kernel

The pragma omp parallel for transformation is actually accomplished like in the original OMPi, where a combined parallel for construct gets replaced by a new parallel construct which has as its body a single for construct, which in turn has as its body the original body of the combined statement (Figure 2).

Original code:

```c
#pragma omp parallel for <clauses>
<original body>
```

Transformed code:

```c
#pragma omp parallel <some clauses>
#pragma omp for <some clauses>
<original body>
```

Figure 2. Combined construct transformation

Another key point is the way clauses are split between the two new constructs: the parallel construct is in charge of nearly all the clauses, except the private and firstprivate ones; the latter are transferred to the omp for directive.

Transforming the parallel construct is the most crucial part of the process. A CUDA kernel is created based on the body of the structure. We use outlining techniques to address this transformation. The for construct body will be the main code which will be executed on the GPU, while the actual construct is mapped on the GPU programming model using thread blocks and grids to dispatch the required number of threads.
The first transformation consists in creating the new kernel function declaration. Complying with the CUDA C standard, the new kernel is defined as __global__ void function. Since the initial code might have several parallel constructs, the name of the kernel function should be unique, so its name is generated as _kernelName_, where _N_ is incremented by 1 for each parallel region. The kernel parameters create a slight problem since the kernel has to take all the variables whether they are shared, private, firstprivate or lastprivate. Further more it has to take variables from both the parallel and the for construct, thus the list is generated in two steps. A general code transformation for a parallel structure is presented in Figure 3:

- In the original code the parallel structure is replaced with a complex instruction block that takes in charge the kernel launch (see Section IV-E).
- In addition the block is preceded and succeeded by CUDA memory allocations and CUDA memory transfers for each variable detected in the clauses (see Section IV-D).

Original code:

```plaintext
#pragma omp parallel
<body>
```

Transformed code – main function:

```plaintext
<CUDA memory allocations>
<CUDA memory copies (host to device)>
<kernel launch>
<CUDA memory copies (device to host)>
<CUDA memory frees>
```

Transformed code – kernel function:

```plaintext
__global__ void _kernel_(void *args)
{
  <block_id and thread_id initializations>
  <slightly modified body>
}
```

Figure 3. Kernel generation

A second stage of the transformation involves processing the for structure. Assume the general fragment of OpenMP code presented in Figure 4, where without loss of generality we assume that _ub_ ≥ _lb_ and _step_ > 0; notice that if the terminating condition was _i_ <= _q_, it would be transformed into the equivalent form _i_ < _ub_, where _ub_ = _q_ + 1.

```plaintext
#pragma omp for
for (i = _lb_; i < _ub_; i += _step)
  <LoopBody>
```

Figure 4. An OpenMP for construct

The current version of our compiler is able to transform single pragma omp parallel for constructs with one loop level. Dealing with nested for loops is more complex, requires further transformations and will be supported in future versions of our compiler.

This type of construct will be mapped to the CUDA execution model where GPU threads are automatically generated at runtime. Each GPU thread will execute the loop body independently on a CUDA core, thus the for statement is completely removed from the code. The information contained in the for statement such as the lower bound, the upper bound and the step will be transmitted to the kernel, so that together with the thread id it can compute the index of each thread. This transformation is required in order to have the correct memory access patterns. The exposed mapping is equivalent to a static schedule with a chunk size of 1 in the OpenMP world. This has no limitation as the number of GPU threads is large enough to accommodate large sets of data.

Finally, the loop body is transformed before being copied to the kernel body: private and shared variables are replaced with their counterparts in the GPU memory, the loop indices are replaced to take into account the thread id and to remain into the bounds (see section IV-F). The following subsection presents specific transformations for the parallel construct variables according to their scope.

D. Variables visibility (shared, private)

Multiple threads within an OpenMP parallel program execute inside the same shared address space and can share access to variables within this address space. Sharing variables between threads makes interthread communication very simple: threads send data to other threads by assigning values to shared variables and receive data by reading values from them.

In addition to sharing access to variables, OpenMP also allows a variable to be designated as private to each thread rather than shared among all threads, in order to allow the threads to do their computations independently. Each thread has a restricted access to a private copy of this variable for the duration of the parallel construct. Some variables may also be declared as firstprivate or lastprivate if their content needs to be send/recovered to/from the thread. These types of variables need a special treatment in order to act in the same way inside the CUDA programming model. In either case equivalent variables have to be allocated on the GPU so that the kernels can access the data: the last part of this section focuses on the required transformations depending on the variables types.

The reader should note that in order to allocate each variable on the GPU, it is important to decide from the beginning whether it is a scalar or an array:

- Scalar variables do not imply a great deal of transformation as they only need a simple cudaMemcpy call.

Approximately 2.9E+17 threads can be generated since the grid size is three dimensional, each dimension has a limit of 65535 blocks in recent computation capability (2.0) and a block can also have a maximum of 1024 threads.
• However the array variables need a special treatment since their size needs to be recovered in order to allocate the same amount of data on the GPU memory. As our goal is to focus on OpenMP code that may have an interesting behaviour when transferred on the GPU, our transformation tool supports only static allocated arrays. Dynamically allocated arrays, whose use may even make loose performance would additionally rise a lot of difficulties since the compiler would for example need to search for the last dynamic allocation or reallocation for the specified arrays.

1) Shared variables: Shared variables either scalar or arrays have an identical treatment. These need to be transferred on the GPU and recovered at the end of kernel execution. This implies generating CUDA memory transfers code. Figure 5 presents shared variables transformation code. In order to assure unique variable naming, shared variables counterparts on the GPU are named using a “_d_” prefix.

Original code:

```c
float sum;
float a[1000];
<sequential code initializations>
#pragma omp parallel shared(a)
<body>
```

Transformed code – main function:

```c
float sum;
float a[1000];
<sequential code initializations>
float _d_sum;
cudaMalloc(&_d_sum, 1 * sizeof(sum));
cudaMemcpy(_d_sum, sum, 1 * sizeof(sum), cudaMemcpyHostToDevice);
float (* _d_a);
cudaMalloc(&_d_a, 1000 * sizeof(a[0]));
cudaMemcpy(_d_a, a, 1000 * sizeof(a[0]), cudaMemcpyHostToDevice);
```

Transformed code – kernel function:

```c
int _d_sh_j[ 32];
_d_sh_j[thread_id % 32] = _d_j[for_lb + thread_id * for_step];
```

Figure 5. Shared variable transformation

2) Private variables: Private variables need a special treatment due to the CUDA specific execution model. As presented in section II, thread scheduling is strictly an implementation concept. The warp (32 threads) is the unit of the thread scheduler in SMs. Based on this concept OpenMP private variables are implemented as arrays in the shared memory of the GPU. In order to keep track of each private variable of a CUDA thread, an array is allocated having the same size as the total number of threads that are going to be used for the kernel execution. In this way each CUDA thread will have its own private version of the private variable. For each private variable an array will be allocated in the global memory of the GPU. The allocation resembles that used for the shared variables. In order to separate the variables in the global memory they have been prefixed with “_d_”. However taking into account the execution model, an optimization has been made for the access to those arrays. Since only the 32 cases from the array needed during the execution of a warp are actually accessed, that exact cases will be copied into the shared memory of the GPU (Figure 6). The 32-case array is declared inside the kernel and has a distinctive prefix “_d_sh_”.

Original code:

```c
int j;
#pragma omp parallel private(j)
<body>
```

Transformed code – main function:

```c
int j;
int (* _d_j);
cudaMalloc(&_d_j, CUDA_thread_num * sizeof(j));
```

Transformed code – kernel function:

```c
int _d_sh_j[ 32];
_d_sh_j[thread_id % 32] = _d_j[for_lb + thread_id * for_step];
```

Figure 6. Private variable transformation

3) “Firstprivate” and “lastprivate” variables: The previous mechanism allows to easily implement the special data clauses firstprivate and lastprivate.

• Taking into account this type of variables needs a transformation similar to the private variables (array allocations).

• Additionally a firstprivate variable requires the initial value to be duplicated in each case of the initial array in the global memory so that it is accessible to each thread.

• Opposed to firstprivate variables, for a lastprivate variable one of the cases of the array has to be transferred back from the GPU to the host at the end of the parallel construct.

E. Kernel launch

CUDA C programming language supports two ways of launching kernels in the runtime application programming interface (see Figure 7). The most frequent approach used by the vast majority of programmers uses a C extension of a function call syntax, allowing to specify the kernel execution configuration parameters between <<< and >>>. This method has the advantage of simplicity, but breaks the C99 standard. In order to support kernel launch in our compiler
we opted for the second method offered by CUDA, but far less used by programmers due to its higher complexity.

In this alternative approach three function calls are made in a strict order to initialize the kernel parameters and for actual kernel execution (see the second method in Figure 7).

First method (the most common, non C standard):

```c
_kernelN_ <<< grid_size, threads_per_block >>> (A, B);
```

Second method (less common, C99 compliant):

```c
cudaConfigureCall(grid_size, threads_per_block, 0, NULL);
offset = 0;
ptr = (void*)(size_t)A;
ALIGN_UP(offset, __alignof(ptr));
cudaSetupArgument(&ptr, sizeof(ptr), offset);
offset += sizeof(ptr);
ptr = (void*)(size_t)B;
ALIGN_UP(offset, __alignof(ptr));
cudaSetupArgument(&ptr, sizeof(ptr), offset);
cudaLaunch("_kernelN_");
```

Figure 7. Two equivalent kernel launch methods

- The `cudaConfigureCall` function species the grid and block dimensions for the device call to be executed similar to the execution configuration using the `<<< >>>` syntax.
- Each of the kernel parameters is specified before execution using a stack model: after alignment, calling `cudaConfigureCall` pushes the parameters into the stack.
- Finally a `cudaLaunch` call will execute the kernel whose name has been specified as a character string. This function call concludes the steps initialized by the `cudaConfigureCall` call, since it is popping the data that was previously pushed to the execution stack.

C grammar modifications needed to support kernel launches have been practically reduced to zero by using the second approach since it uses only standard C calls.

A similar simplifying approach was used to declare the block size (`blockDim`) and grid size (`gridDim`). In the standard CUDA runtime environment these declarations use a C4+ style constructor for a `dim3` type variable. However the CUDA library declares a `dim3` type as a three-field user defined structure which can be accessed using the point operator. Thus a declaration like

```c
dim3 grid_size(num_blocks_x, num_blocks_y);
```

is equivalent with two assertion instructions like

```c
grid_size.x = num_blocks_x;
grid_size.y = num_blocks_y;
```

**F. AST transformations**

All the transformations detailed in the previous sections take place in the abstract syntax tree (AST). Operations are conducted at all levels of the AST, either leaves or branches and consist in either modifying the existing nodes or adding new nodes or entire subtrees. In order to explain this modifications a simple example of a two vectors sum is presented. The first section of Figure 8 shows the initial body of the `omp parallel for` loop, while the second one presents the transformed code that is put in the kernel to replace the initial pragma body. Figure 9 represents the same code, but in terms of ASTs. One can clearly note that the identifiers (variable names) are replaced with their new equivalents on the GPU (see Section IV-D). However, for all its occurrences, the `i` variable is completely replaced by an entire new subtree which computes the new index in term of the loop bounders.

Original code:

```c
c[i] = a[i] + b[i];
```

Transformed code:

```c
_d_c[for_lb+thread_id*for_step] = _d_a[for_lb+thread_id*for_step] + _d_b[for_lb+thread_id*for_step];
```

Figure 8. Loop body for a two vectors sum

Only after all the modifications have been made, the final code is dumped into a file by a depth-first search graph traversal.

**V. CONCLUSIONS AND PERSPECTIVES**

After four years of existence and the same number of version iterations, the CUDA programming framework has become a mature product, but is still in a constant evolution. Each new major version brings important features and support for the new available architectures. However, writing a parallel code that can benefit from the GPU acceleration remains a painstaking job for the programmer. In order to deal with this problem a source-to-source compiler seems to be the best approach to quickly develop CUDA enabled code. The initial code is OpenMP C code due to the ease of programmability and simple parallelization of serial code.

This paper presented our implementation of a source-to-source compiler from an OpenMP C code to CUDA code. We highlighted the key points involved in each step of the code transformation and noted the challenges of developing such a tool. We are able to transform single loop `omp parallel for` constructs into CUDA compliant kernels able to execute on the GPU. The main objective was to
choosing the threads number with respect to the amount deduce the optimal number of threads per block (the use the optimal values to construct the effective number making a partial compilation and an execution tuning, in

Accelerator GPU Model.html and Programming

clusters using three levels: internode communication using MPI in an envelope allowing to map the model on a multiGPU

Another part of our work consists in providing multiGPU support. We already allow distributing computations over several GPUs inside a node using OpenMP and CUDA [18]; our next step in this direction is to integrate the entire framework in an envelope allowing to map the model on a multiGPU cluster using three levels: internode communication using MPI

ACKNOWLEDGMENT

The authors acknowledge the financial support of the CEA DAM/Ile-de-France and Région Champagne-Ardenne and would also like to thank “Centre de Calcul Champagne-Ardenne ROMEO” of University of Reims Champagne-Ardenne for the hardware resources provided and the technical support offered in conducting this research.

REFERENCES