Università di Pisa

Department of Computer Science
Master Degree in Computer Science

Master Thesis

CPU Support for CUDA

Candidate
Luca Atzori

Supervisors
Prof. Marco Danelutto
Dott. Vincenzo Innocente
Dott. Felice Pantaleo

Referee
Prof. Stefano Chessa

Academic Year 2014/2015

Contents

Introduction

Nowadays in many machines one of the greatest computational resources is their graphics coprocessors (GPUs), not just their primary CPUs. These devices can achieve high throughput and energy efficiency and they are therefore well suited to be the offload engines for heavy parallel computations. The early efforts to use GPUs as general-purpose processors (GPGPU) required reformulating computational problems in terms of graphics primitives. This cumbersome translation was obviated by the advent of general-purpose programming languages and APIs in the second half of the last decade, allowing programmers to ignore the underlying graphical concepts in favour of more common high-performance computing concepts.

In 2006, NVIDIA released CUDA (Compute Unified Device Architecture), a platform designed to work with programming languages such as C, C++ and Fortran which allows to use GPU resources for general purpose application programming without requiring advanced skills in graphics programming. Although CUDA is at the moment one of the most popular frameworks, its adoption has limited the programmers to execute their programs only on NVIDIA GPUs.

This is unfortunately an undesirable scenario, as programmers who have invested the effort to write a general-purpose application for a GPU should not have to make an entirely separate programming effort to effectively parallelize the application across GPU devices of different vendors. At the same time, the large amount of hardware not necessarily having a GPU is a resource that we would not like to left unused.

Exploiting this resources raises the issue of guaranteeing portability between different architectures. Although the portability of the functionality of the applications must be ensured, our other main interest is performance portability, a major challenge faced today by the heterogeneous high performance programming community.

In this thesis we describe and implement a set of techniques that allow

to execute CUDA programs on systems not necessarily having an NVIDIA GPU, in particular targeting shared memory multi-core CPU architectures.

The accomplishment of this goal is achieved operating at the CUDA source code level, analysing its abstract syntax and defining a set of transformations on it. As result of this transformations, the output will be standard C++ source code.

The choice of using C++ as destination language was done because of the availability on a huge number of platforms, making portability less expensive. Also, taking into account that CUDA is a set of extensions of the C language family (including C++), several similarities can be found between them. This made easier the definition of a set of formal rules describing how the CUDA syntax is transformed into C++ syntax preserving the several key abstractions provided by CUDA the programming model as well as the semantics of the program.

As a proof of concept of the correctness of the transformations presented in this thesis, a prototype source-to-source translator has been implemented. The implementation of this software relies on the use of robust and wide-used frameworks. The Clang compiler front-end provided us the machinery used by our software to accomplish the syntactic analysis and transformation of the code. In order to guarantee comparable performance on the target system, the Intel Threading Building Blocks framework has been used to parallelize the execution of the obtained code.

This software is therefore evaluated, analysing the achieved portability in terms of manual effort still needed in order to run the transformed programs. Also, the performances of the execution of the translated programs are evaluated, comparing the obtained results with standard applications natively conceived to execute on the target system.

Portability

This section describes the concept of portability, the main feature we want to achieve in this thesis work.

In high-level computer programming, portability is the usability of the same software in different environments. The portability prerequirement is the abstraction between the application logic and the underlying system interfaces.

Portability is a typical issue of parallel programming [11], since the increase in number and the diversity of computing units within nodes introduce a challenge for library and application developers, who need to adapt their code to diverse target systems.

From a more abstract perspective, portability can be distinguished among functional and non-functional.

With functional portability we mean the portability of the functionality of the application, namely that the same results are computed on both architectures.

Non-functional portability refers to those properties that do not directly involve the result computed by the program, but rather to the way these results are computed. Among these properties we can list security, fault tolerance and power management. Also performance is a non functional feature of applications. With *performance portability* we mean that a program must preserve a comparable performance when executed on the target system with respect to the performance of the original system.

Related Work

There exist several projects that translate from (or to) the CUDA programming model.

The most relevant related work in this area is a source-to-source translator called MCUDA [24] which aims to execute CUDA kernels on CPUs. There are similarities with our work, mainly related to the use of the same approach in defining some transformations and the choice of implementing source-to-source translation. However, the implementation of MCUDA relies on the Cetus framework[19], a Java-based framework that provides a class hierarchy that represents a program's abstract syntax tree (AST).

Two other source-to-source translator that translate from CUDA were

developed, both targeting OpenCL. The first one, again relying on Cetus, is CUDAtoOpenCL [21]. The other work is CU2CL [20, 23]. Although the targeted platform is different, the main similarity with our work is the fact that CU2CL's implementation relies on Clang.

Swan [16] is a tool made to ease the transition between OpenCL and CUDA. However, Swan is not a source-to-source translator like the tools mentioned above; instead it provides a higher-level library that abstracts both the CUDA and OpenCL APIs.

Another project of interest is GPU Ocelot [12, 13], an open-source dynamic just-in-time compilation framework for GPU compute applications targeting a range of GPU and non-GPU execution targets. Ocelot supports CUDA applications and provides an implementation of the CUDA Runtime API enabling seamless integration. NVIDIA's virtual instruction set architecture is used as a device-agnostic program representation that captures the data-parallel SIMT execution model of CUDA applications. Ocelot supports several backend execution targets – NVIDIA GPUs, AMD GPUs, and a translator to LLVM for efficient execution of GPU kernels on multi-core CPUs.

Document Organization

The thesis is organized as follows: in Chapter 1 are introduced the main concepts behind the CUDA programming model, highlighting the syntactic aspects relevant to this work. Chapter 2 describes the main ideas behind the approach followed in this work, highlighting the techniques developed to allow the translation from CUDA to C++ code. Chapter 3 describes Clang, the tool on which this work relies to accomplish the syntactic analysis of the source code. Chapter 4 gives an high-level description of the software implemented as a proof of concept of the transformations described in this thesis work. Chapter 5 describes the implementation details of the tools created to accomplish the source-to-source translation. In Chapter 6 an evaluation of the software is given. Lastly, Chapter 7 summarizes our work and presents possible future extensions.

Chapter 1

CUDA

This chapter introduces the main concepts behind the CUDA programming model. The first section briefly describes the CUDA programming model while the second section highlights the syntactic aspects relevant to this work.

A full description of the NVIDIA CUDA framework goes out of the scope of this thesis. For more details please refer to the CUDA Programming Guide [22].

1.1 Programming model

CUDA (Compute Unified Device Architecture) is a parallel computing platform created by NVIDIA that allows software developers to use CUDA-enabled graphics processing units (GPU) for general purpose processing. The CUDA platform is designed to work with programming languages such as C, C++ and Fortran providing a few simple extensions that enable expressing fine-grained and coarse-grained data parallelism. The CUDA API model allows developers to exploit that parallelism by writing straightforward C code that will then run in thousands of parallel invocations, or threads, on the GPU.

From an architectural perspective, because threads (and not data) are mapped to the processor and executed, the style of execution of CUDA is called Single-Instruction, Multiple-Thread (SIMT). SIMT is very similar to Single-Instruction, Multiple-Data (SIMD). In SIMD, multiple data can be processed by a single instruction. In SIMT instead, each thread executes the same instruction, but possibly on different data.

Computations that are to be performed on the GPU are specified in the code as explicit kernels. Prior to launching the kernel all the data required for the computation must be transferred from the host (CPU) memory to the GPU (global) memory. A kernel invocation will hand over the control to the CPU, and the specified GPU code will be executed on this data.

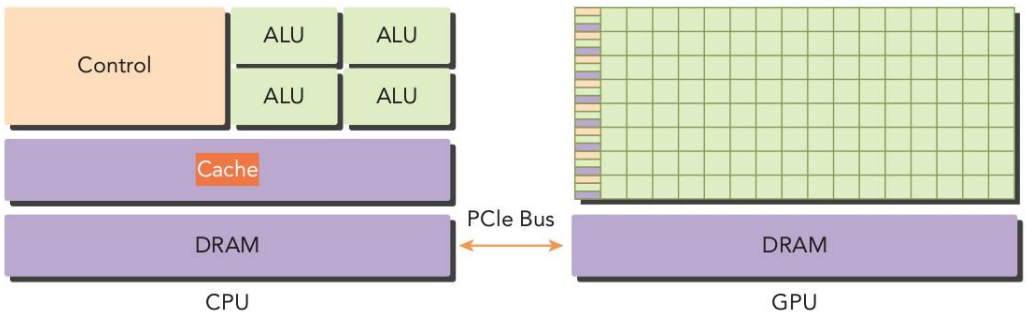

Figure 1.1: Host-Device model.

1.1.1 Thread hierarchy

CUDA provides a thread hierarchy that facilitates the decomposition of an application into a set of parallel tasks. When a CUDA kernel is launched, a *grid* is allocated in the GPU context. A grid is an executing instance of a

kernel and consists of a set of *blocks*. Each block defines an independent task that executes in a task parallel way. Intercommunication among blocks is not possible because the execution order for blocks is not defined at compile time. A block can be split into a set of parallel *threads*. Threads in a block cooperate in a data parallel way. Each thread executes the kernel once. The kernel instructions are executed by each thread usually on a portion of the input data dependent by the index of the thread and the block.

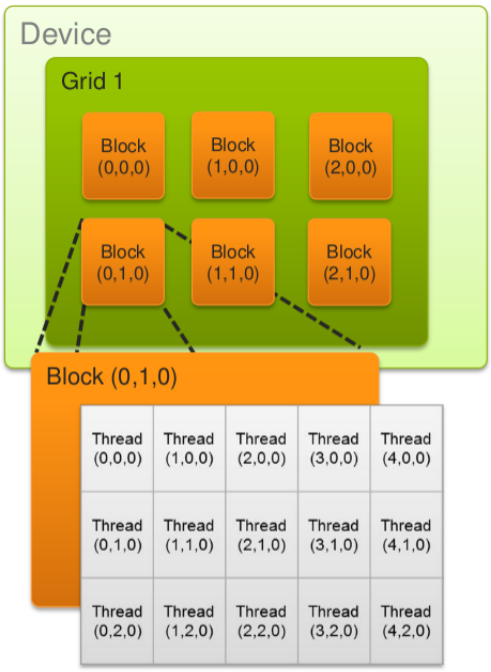

Figure 1.2: CUDA thread hierarchy.

1.1.2 Memory hierarchy

The CUDA memory hierarchy reflects the thread hierarchy described before. Each thread has access to a private register space on the chip for storing single variables and constant size arrays. If the amount of registers is exceeded, threads can access *local memory* for storing large data that does not fit the register space. Local memory resides in global memory space and has the same slow bandwidth for read and write operations. Global memory is a virtual address space that can be mapped to device memory (memory on the graphics card) or page-locked (pinned) host memory. Local memory is allocated in global memory but accessed through different address spaces and caches.

All threads in a block have access to a *shared memory*, which is private to the block, to cooperate on a computation. Local and shared memory visibility is depicted in Figure 1.3.

Figure 1.3: Local and shared memory.

Global memory is visible to all threads. In the host side of a CUDA program it is possible to access only the global memory, therefore, parameters and data used by a kernel are initially located in the global memory. Global memory can also be accessed through two read-only caches known as the constant memory and texture memory for efficient access for each thread. Global memory visibility is depicted in Figure 1.4.

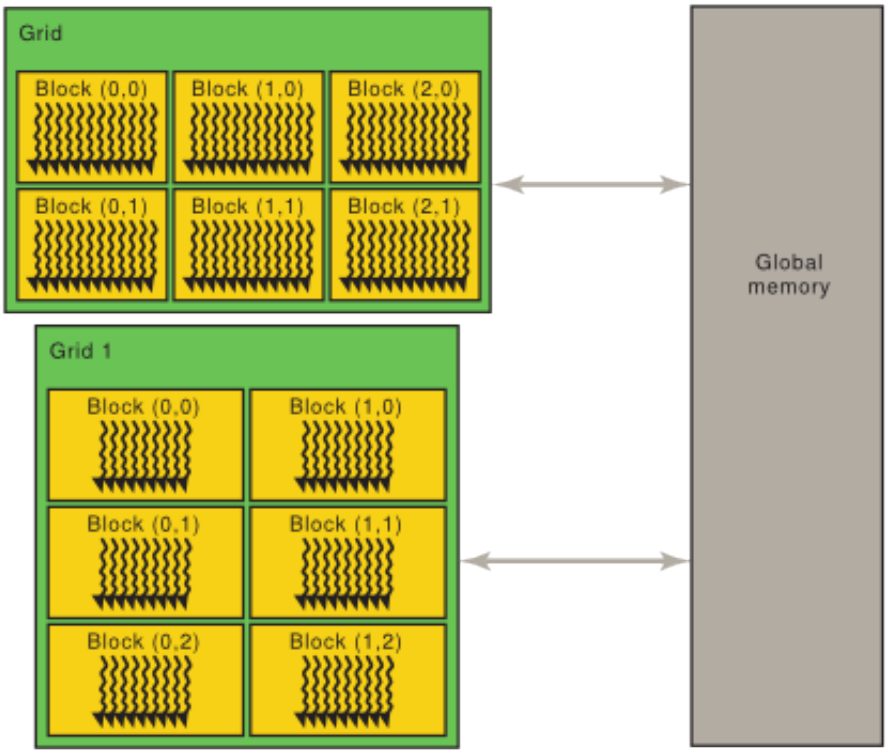

Figure 1.4: Global memory.

1.1.3 Synchronization

Communications between threads in a block require the presence of a mechanism for synchronization. For example if thread A writes a value to shared memory and a thread B wants to do something with this value, thread B can't start its work until the write from thread A is completed. Without synchronization, a race condition is created where the correctness of the execution results depends on the non-deterministic details of the hardware. To address this issue, CUDA allows all the threads in a block to synchronize by means of a barrier defined by the user in the kernel code. Global synchronization of all threads can only be performed across separate kernel launches. This also implies that an implicit synchronization happens every time a kernel is entered or exited.

1.2 Syntax

This section highlights the syntactical aspects of CUDA that are relevant to this thesis work.

Kernels

CUDA allows the programmers to define functions known as *kernels* that, when called, are executed in parallel by the CUDA threads.

A kernel is defined using the `__global__` declaration specifier. This keyword indicates a function that runs on the device and is called from the host code. To identify the call of a kernel from the host code, CUDA introduces the triple angle brackets notation `<<<...>>>` also called "kernel launch". This execution configuration syntax takes two arguments, the number of blocks per grid and the number of threads per block.

```
1  // Kernel definition
2  __global__ void vector_add(int* v1, int* v2, int* v3)
3  {
4    int i = threadIdx.x;
5    v3[i] = v1[i] + v2[i];
6  }
7
8  int main()
9  {
10   ...
11   // Kernel invocation with NT threads
12   vector_add<<<1, NT>>>(v1, v2, v3);
13   ...
14 }
```

Listing 1.1: A vector addition.

In the previous example (Listing 1.1), each of the `NT` threads that execute `vector_add` performs an addition. There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. This limit depends from the compute capability[1] of the device, and on current GPUs a block may contain up to 1024 threads. However, a kernel

[1]The compute capability of a device identifies the features supported by the GPU hardware and is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU.

can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks. Line 4 shows how the ID of the thread is used to access the data: one common usage for threadIdx (and blockIdx) is to determine the area of data that a thread is to work on.

`dim3` and built-in variables

Each thread that executes the kernel is given a unique thread ID, accessible using the built-in `threadIdx` variable.

`threadIdx` is a 3-component vector defined by the data type `dim3`, so that threads can be identified using (up to) a three-dimensional index, providing a natural way to invoke computation across the elements in a domain such a vector, matrix or volume. The three dimensions of `dim3` type variables can be accessed by the fields `x`, `y` and `z`.

In addition, blocks within the grid are organized into a one-dimensional, two-dimensional, or three-dimensional way, and they can be accessed inside the kernel through the built-in `blockIdx` variable.

The kernel launch parameters can be of type `int` or `dim3`, and their values can be used inside the kernel code. The number of threads per block may be accessed through the built-in `blockDim` variable. The number of blocks per grid may be accessed through the built-in `gridDim` variable. Even if an integer value is passed to the kernel launch configuration, inside the kernel those two variables will be of type `dim3`, with any component left unspecified initialized to 1 by default.

As an example, the kernel on Listing 1.2 adds two matrices `m1` and `m2` of size $N \times N$ and stores the result into matrix m3:

```
1  // Kernel definition
2  __global__ void matrix_add(int m1[N][N], int m2[N][N], int
       m3[N][N])
3  {
4    int i = blockIdx.x * blockDim.x + threadIdx.x;
5    int j = blockIdx.y * blockDim.y + threadIdx.y;
6    if (i < N && j < N)
7      m3[i][j] = m1[i][j] + m2[i][j];
8  }
9
10 int main()
11 {
12   ...
```

```
13    // Kernel invocation
14    dim3 threadsPerBlock(16, 16);
15    dim3 blocksPerGrid(N / threadsPerBlock.x, N /
          threadsPerBlock.y);
16    matrix_add<<<blocksPerGrid,threadsPerBlock>>>(m1, m2, m3);
17    ...
18  }
```

Listing 1.2: A matrix addition.

It is possible to observe the usage of the block IDs in order to access the matrices correctly (Line 4–5). Also, the kernel launch configuration (Line 16) takes now two `dim3` arguments.

`__shared__` and `__syncthreads`

The `__shared__` qualifier declares a variable that resides in the shared memory space of a block. It has lifetime of a block and is only accessible from all the threads within the block. Being on-chip, shared memory is much faster than global memory. Any opportunity to replace global memory accesses by shared memory accesses should therefore be exploited as illustrated by the following example.

```
1  __global__ void stencil_1d(int *in, int *out)
2  {
3    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
4    int global_id = threadIdx.x + blockIdx.x * blockDim.x;
5    int local_id = threadIdx.x + RADIUS;
6
7    //Read input elements into shared memory
8    temp[local_id] = in[global_id];
9    if(threadIdx.x < RADIUS) {
10     temp[local_id - RADIUS] = in[global_id - RADIUS];
11     temp[local_id + BLOCK_SIZE]= in[global_id + BLOCK_SIZE];
12   }
13
14   //Synchronize (ensure all the data is available)
15   __syncthreads();
16
17   //Apply the stencil
18   int result = 0;
19   for(int offset = -RADIUS; offset <= RADIUS; offset++)
20     result += temp[local_id + offset];
21
```

```
22    //Store the result
23    out[global_id] = result;
24  }
```

Listing 1.3: A stencil example.

At Line 3 of Listing 1.3 the `temp` array is declared with the `__shared__` keyword, in order to cache data in shared memory. Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel by calling the `__syncthreads()` intrinsic function (Line 15). `__syncthreads()` acts as a barrier at which all threads in the block must wait before any is allowed to proceed.

Memory management APIs

The CUDA programming model assumes a system composed of a host and a device, each with their own separate memory. Kernels operate out of device memory, so the runtime provides functions to allocate, deallocate, and copy device memory, as well as transfer data between host memory and device memory.

Device memory is typically allocated using `cudaMalloc()` and freed using `cudaFree()` and data transfer between host memory and device memory are typically done using `cudaMemcpy()`. Listing 1.4 shows the vector addition code sample including the memory management in the host code:

```
1  // Device code
2  __global__ void vector_add2(int* v1, int* v2,int* v3, int N)
3  {
4    int i = blockDim.x * blockIdx.x + threadIdx.x;
5    if (i < N)
6      v3[i] = v1[i] + v2[i];
7  }
8
9  // Host code
10 int main()
11 {
12   int N = ...;
13   size_t size = N * sizeof(int);
14
15   // Allocate input vectors h_A and h_B in host memory
16   int* h_A = (float*)malloc(size);
17   int* h_B = (float*)malloc(size);
```

```
18
19    // Initialize input vectors
20    ...
21
22    // Allocate vectors in device memory
23    int* d_A;
24    cudaMalloc(&d_A, size);
25    int* d_B;
26    cudaMalloc(&d_B, size);
27    int* d_C;
28    cudaMalloc(&d_C, size);
29
30    // Copy vectors from host memory to device memory
31    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
32    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
33
34    // Invoke kernel
35    int threadsPerBlock = 256;
36    int blocksPerGrid = (N + threadsPerBlock - 1) /
          threadsPerBlock;
37    vector_add2<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B,
          d_C, N);
38
39    // Copy result from device memory to host memory
40    // h_C contains the result in host memory
41    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
42
43    // Free device memory
44    cudaFree(d_A);
45    cudaFree(d_B);
46    cudaFree(d_C);
47
48    // Free host memory
49    ...
50 }
```

Listing 1.4: Vector addition showing the memory management.

First of all, the vectors need to be allocated (Line 24, 26 and 28). Then the vectors are copied from host memory to device memory (Line 31–32). After the kernel execution the result of the computation is copied from device memory to host memory (Line 41) and finally the device memory is freed (Line 44-46).

Summary

In the first section of this chapter we gave a brief overview of the CUDA programming model, in particular highlighting its thread and memory hierarchy.

In the second section, the CUDA syntax is described, underlining an useful subset for our work.

Chapter 2

Translating CUDA

This chapter describes the main ideas behind the source-to-source approach followed in this thesis work, highlighting the techniques developed to allow the translation from CUDA to C++ code.

The first section discusses the reasons behind the choice of doing a source-to-source translation.

The second section formalizes the mapping of the CUDA programming model, showing the set of transformations applied to the CUDA programming constructs in order to obtain C++ code.

2.1 Source-to-source translation

To accomplish the transformation of the CUDA programs, the chosen approach is the development of a tool capable of translating the source code in regular C++ language.

A **source-to-source translator** (or a transcompiler) is a type of compiler that takes the source code of a program written in a certain programming language as its input and produces the equivalent source code in another programming language. The main difference with a traditional compiler is that a source-to-source compiler translates between programming languages that operate at approximately the same level of abstraction. Instead, a traditional compiler, usually translates from a higher level programming language to a lower level one, typically the assembly language.

Although finding a way to compile CUDA code directly in x86 assembly (or other low-level languages) is of course a valid approach [13], applying a source-to-source translation is a more useful solution, for a set of different reasons.

As stated before, the aim of this work is to run CUDA code on a broad set of architectures, and not to be bound to a specific one. Moreover, being CUDA a relatively small set of extensions of the C/C++ language, it is reasonable and feasible to obtain in output C++ source code.

Targeting a high-level language provides several advantages. The most prominent ones are the possibility to perform further optimizations way more easily as well as simplifying the debugging of the code.

Also, from the implementation side of the problem, this choice reveals to be far-sighted, since the use of Clang is a good fit in this sense, for the reasons explained in Chapter 3.

2.1.1 Abstract Syntax Trees

An **Abstract Syntax Tree** (hereafter referred to as AST) is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. Abstract syntax trees differ from parse trees (concrete syntax trees) because superficial distinctions of form, unimportant for translation, do not appear on them. ASTs do not show the whole syntactic clutter, but represent the parsed string in a structured way, discarding all information that may be important for parsing the string, but

is not needed for analysing it.

For example, in Figure 2.1 is shown an AST for the code in Listing 2.1:

```
1 while( x < 10 )
2 {
3   x = x + 1;
4 }
```

Listing 2.1: A while loop.

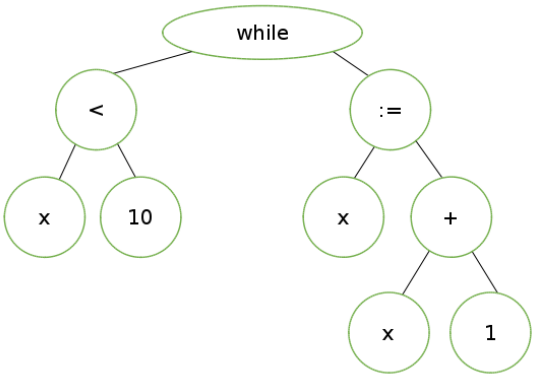

Figure 2.1: An example of AST for the code in Listing 2.1.

Figure 2.1 shows how every detail appearing in the real syntax such as grouping parenthesis or semicolons does not appear in the tree structure.

Taking into account this definition, it seems clear that ASTs are the perfect representation in order to achieve source-to-source translation. In fact, relying on ASTs allows to manipolate the CUDA programming constructs more easily, not having to consider all the stylistic technicalities of the language.

Again, Clang revealed to be the right tool to perform this task, since its ASTs showed to be a very good fit for representing the syntax, as explained in detail in Section 3.2 of Chapter 3.

2.2 Mapping

Mapping the CUDA programming model to a CPU architecture means that all the several key abstractions provided by CUDA, such as hierarchical memory, threads, blocks and barrier synchronization, has to be matched with the corresponding entities available on the target system, in order to be able to execute the resulting program on a CPU architecture.

Even if the model seems quite easy to handle, as described in Chapter 1, the mapping of the computations is not straightforward at all. As shown in the rest of this section, several issues and limitations that arise mainly due to architectural differences has to be considered.

The conceptually simplest implementation would be spawning a CPU thread for every CUDA thread specified in the programming model. However, this approach will be quite inefficient. First of all the benefits deriving from having thread locality would be mitigated. Also, an approach like this, will incur in an extremely large amount of scheduling overhead. Again, this is due to the big difference in the hardware, having usually a number of cores in a GPU architecture way larger than the one in a multi-core x86 CPU.

Therefore, a mapping that maintains the locality expressed in the programming model should be found, with the constraint of not modifying the operating system or the architecture of destination. This also means that somehow the execution of the CUDA threads must be managed in the code in an explicit way.

The approach presented on this work relies on a set of considerations on the nature of the CUDA programming model. First of all, CUDA blocks execution is asynchronous. Taking also into account that each CPU thread should be scheduled to a single core for locality, it appears quite clear that there is a correspondence that can be exploited. This first consideration also implies that the former CUDA threads cannot be treated any more as concurrent entities. Analysing also the fact that the ordering semantics imposed by a potential barrier synchronization point has to be maintained, the proposed approach is to serialize the execution of the CUDA threads, changing the nature of the kernel functions from a per-thread code specification to a per-block one.

The proposed approach is summarized in Table 2.1:

GPU	**CPU**
block (asynchronous)	CPU thread
GPU thread (synchronized with barriers)	sequential, unrolled loop

Table 2.1: Mapping CUDA concepts to C++.

In the next sections, it will be presented through some examples how this goal is accomplished describing how the programming constructs of CUDA

are transformed in C++. After each example, one or more formal rules are defined to describe the applied transformation.

2.2.1 Kernel translation

The transformation of the CUDA kernel source code is the main aspect that has to be managed in order to achieve portability on CPUs.

As stated in the previous section, the idea behind this is the serialization of the execution behaviour of the CUDA threads.

For example, Listing 2.2 shows again the kernel for the vector addition shown in Chapter 1.

```
__global__ void vector_add(int* v1, int* v2, int* v3)
{
  ...
  v3[threadIdx.x] = v1[threadIdx.x] + v2[threadIdx.x];
  ...
}
```

Listing 2.2: Vector addition kernel.

To make the CPU able to execute this kernel, it appears clear that its behaviour has to be handled with respect to the value of the `threadIdx` variable, since in a C++ representation this is not a built-in variable any more. To address this issue, the execution of the former CUDA threads is serialized wrapping the body of the kernel function in a loop. This loop enumerates the values of the previously implicit `threadIdx` variable. Listing 2.3 shows the kernel function after this transformation.

```
void vector_add(int* v1, int* v2, int* v3, dim3 blockIdx,
    dim3 blockDim)
{
  dim3 threadIdx;
  ...

  for(threadIdx.x = 0; threadIdx.x < blockDim.x; threadIdx.x
      ++){
    v3[threadIdx.x] = v1[threadIdx.x] + v2[threadIdx.x];
  }
  ...
}
```

Listing 2.3: Translated vector addition kernel.

Analysing the obtained code, it is possible to notice that the main difference is the for loop shown at Line 6. This loop explicitly serializes the execution of the CUDA threads, executing the content of the kernel (the sum at Line 7) as many times as the number of CUDA threads in the block. As stated above, `threadIdx` is not a built-in variable any more, so now, as shown at Line 3, it has to be explicitly declared. Notice at Line 1 how the `__global__` keyword is simply removed, since we are moving to a C++ function. Lastly, again at Line 1, the other previously implicit variables `blockIdx` and `blockDim` has to be provided to the function, adding them to the parameter list.

To change the nature of the kernel function to a per-block specification, an iterative structure that wraps the body of the function was introduced. Taking into account that we are operating with three-dimensional data types, this iterative wrapping will be, in the general case, constituted by three nested loops, iterating on the values of the three components:

```
1  ...
2  for(threadIdx.z=0; threadIdx.z < blockDim.z; threadIdx.z++){
3    for(threadIdx.y=0; threadIdx.y < blockDim.y; threadIdx.y
         ++){
4      for(threadIdx.x=0; threadIdx.x < blockDim.x; threadIdx.x
           ++){
5        //Kernel body
6        ...
7      }
8    }
9  }
10 ...
```

Listing 2.4: The complete iterative wrapping in the three-dimensional scenario.

The behaviour shown in the previous examples is now formalized, giving generic rules for this type of transformation. Taken a generic kernel function, its translation is defined as follows:

Rule 1. The `__global__` keyword is removed.

$$\texttt{\_\_global\_\_ void kernel(...)\{...\}} \longrightarrow \texttt{void kernel(...)\{...\}}$$

Rule 2. The `dim3 blockIdx` and `dim3 blockDim` formal parameters are added to the parameter list $P_1, \ldots, P_n$.

$$\texttt{void kernel(P}_1\texttt{ , ... , P}_n\texttt{)\{...\}}$$
$$\downarrow$$
$$\texttt{void kernel(P}_1\texttt{ , ... , P}_n\texttt{, dim3 blockIdx, dim3 blockDim)\{...\}}$$

Rule 3. The body of the kernel is wrapped by the three nested loops, as shown in Listing 2.4.

```
void kernel(...)
{
    Stmt_1
    ...
    Stmt_n
}
```

$\longrightarrow$

```
void kernel(...)
{
    ITERATIVE_WRAPPING{
        Stmt_1
        ...
        Stmt_n
    }
}
```

Rule 4. The `dim3 threadIdx;` variable declaration is inserted at the beginning of the body of the function.

```
void kernel(...)
{
    ...
}
```

$\longrightarrow$

```
void kernel(...)
{
    dim3 threadIdx;
    ...
}
```

2.2.2 Ensuring synchronization

So far, the translation is correct, and the semantics of the output code is the same of the input one. Unfortunately is not that simple, and more problems arise when an explicit synchronization between CUDA threads has to be taken into account.

For example, Listing 2.5 shows again the stencil kernel shown in Chapter 1.

```
1 __global__ void stencil_1d(int *in, int *out)
2 {
3   __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
4   int global_id = threadIdx.x + blockIdx.x * blockDim.x;
5   int local_id = threadIdx.x + RADIUS;
6
7   //Read input elements into shared memory
8   temp[local_id] = in[global_id];
```

```
9   if(threadIdx.x < RADIUS) {
10    temp[local_id - RADIUS] = in[global_id - RADIUS];
11    temp[local_id + BLOCK_SIZE]=in[global_id + BLOCK_SIZE];
12  }
13
14  //Synchronize (ensure all the data is available)
15  __syncthreads();
16
17  //Apply the stencil
18  int result = 0;
19  for(int offset = -RADIUS; offset <= RADIUS; offset++)
20    result += temp[local_id + offset];
21
22  //Store the result
23  out[global_id] = result;
24 }
```

Listing 2.5: A stencil kernel example.

At Lines 8–12 the input elements into the shared memory are read. After the initialization, a synchronization of the CUDA threads is needed (Line 15) because, being threads executed in parallel, it has to be ensured that all the data is available at this moment. Once the threads are synchronized the stencil operation can be applied, that consists in a simple for loop (Line 19–20), and the calculated result is stored in the output array (Line 23).

It appears clear that even in the translated version, wrapping the content of the kernel inside the nested for loops, does not guarantee that the data needed from a certain thread will be available.

To solve this issue, the proposed solution is just to close the for loops when a synchronization call is encountered, and to open another set of nested loops right after it. This works because of the fact that the values of all the thread IDs are iterated, so closing and reopening the iterative wrapping implicitly emulates the behaviour of the synchronization among the CUDA threads.

The translated output is reported on Listing 2.6:

```
1 void stencil_1d(int *in, int *out, dim3 blockDim, dim3
    blockIdx)
2 {
3   dim3 threadIdx;
4   for(threadIdx.z=0; threadIdx.z < blockDim.z; threadIdx.z
      ++){
```

```
5     for(threadIdx.y=0; threadIdx.y < blockDim.y; threadIdx.y
          ++){
6       for(threadIdx.x=0; threadIdx.x < blockDim.x; threadIdx
            .x++){
7         __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
8         int global_id=threadIdx.x + blockIdx.x * blockDim.x;
9         int local_id = threadIdx.x + RADIUS;
10        //Read input elements into shared memory
11        temp[local_id] = in[global_id];
12        if(threadIdx.x < RADIUS) {
13          temp[local_id - RADIUS] = in[global_id - RADIUS];
14          temp[local_id + BLOCK_SIZE] = in[global_id +
                BLOCK_SIZE];
15        }
16      }
17    }
18  }
19  //Implicit synchronization
20  for(threadIdx.z=0; threadIdx.z < blockDim.z; threadIdx.z
      ++){
21    for(threadIdx.y=0; threadIdx.y < blockDim.y; threadIdx.y
        ++){
22      for(threadIdx.x=0; threadIdx.x < blockDim.x; threadIdx
          .x++){
23        //Apply the stencil
24        int result = 0;
25        for(int offset= -RADIUS; offset <= RADIUS;offset++)
26          result += temp[local_id + offset];
27
28        //Store the result
29        out[global_id] = result;
30      }
31    }
32  }
33 }
```

Listing 2.6

Lines 16–22 show how the synchronization call is substituted by the closing and reopening of the iterative wrapping.

To formalize the behaviour shown in the previous example, a new rule that describes this transformation is introduced:

Rule 5. Every `__syncthreads()` statement is removed. At its place, is inserted the closure of the iterative wrapping and another wrapping is sud-

denly after reopened.

```
void kernel(...)
{
    ITERATIVE_WRAPPING{
        ...
        __syncthreads(); ⟶
        ...
    }
}
```

```
void kernel(...)
{
    ITERATIVE_WRAPPING{
        ...
    }
    ITERATIVE_WRAPPING{
        ...
    }
}
```

2.2.3 Emulating thread-local memory

There is another issue introduced by the presence of synchronization. As said in Section 1.1.2, every variable declared without the shared attribute is visible only locally by the thread. So, in this case, closing and reopening the iterative wrapping will cause the loss of the values associated to every variable for that specific thread (ID). This happens because on every iteration the previous value is overwritten before using it.

To address this issue, the adopted technique is to augment the dimensionality of those variables, and to access them by thread ID inside the for loops:

```
1  void stencil_1d(int *in, int *out, dim3 blockDim, dim3
       blockIdx)
2  {
3    dim3 threadIdx; int tid;
4    int temp[BLOCK_SIZE + 2 * RADIUS];
5    int global_id[];
6    int local_id[];
7    for(threadIdx.z=0; threadIdx.z < blockDim.z; threadIdx.z
       ++){
8      for(threadIdx.y=0; threadIdx.y < blockDim.y; threadIdx.y
         ++){
9        for(threadIdx.x=0; threadIdx.x < blockDim.x; threadIdx
           .x++){
10         tid = threadIdx.x + threadIdx.y*blockDim.x +
             threadIdx.z*blockDim.y;
11         global_id[tid] = threadIdx.x+blockIdx.x*blockDim.x;
12         local_id[tid] = threadIdx.x + RADIUS;
13
```

```
14          //Read input elements into shared memory
15          temp[local_id[tid]] = in[global_id[tid]];
16          if(threadIdx.x < RADIUS) {
17            temp[local_id[tid] - RADIUS] = in[global_id[tid] -
                RADIUS];
18            temp[local_id[tid] + BLOCK_SIZE] = in[global_id
                [tid] + BLOCK_SIZE];
19          }
20        }
21      }
22    }
23    //Implicit synchronization
24    for(threadIdx.z=0; threadIdx.z < blockDim.z; threadIdx.z
        ++){
25      for(threadIdx.y=0; threadIdx.y < blockDim.y; threadIdx.y
          ++){
26        for(threadIdx.x=0; threadIdx.x < blockDim.x; threadIdx
            .x++){
27          tid = threadIdx.x + threadIdx.y*blockDim.x +
                threadIdx.z*blockDim.y;
28          //Apply the stencil
29          int result = 0;
30          for(int offset= -RADIUS;offset <= RADIUS; offset++)
31            result += temp[local_id[tid] + offset];
32
33          //Store the result
34          out[global_id[tid]] = result;
35        }
36      }
37    }
38  }
```

Listing 2.7: Translated stencil kernel.

Analysing Listing 2.7 is possible to observe first of all that all the variables having as scope the entire kernel body are declared now outside the iterative wrapping (Lines 4–6). The `__shared__` attribute is simply removed from the declarations (Line 4). The thread-local variables `global_id` and `local_id` are now declared as arrays (Lines 5–6) and their size will correspond to the size of the block, that is the number of threads, since a private value for every thread ID needs to be stored. In case an initialization is found, the declaration and the initialization are split, leaving the latter in the original position (Lines 11–12).

The `tid` index used to access this dimensionality-augmented variables is calculated remembering that once again three dimensional data types are used. In order to have the correct index for every loop iteration, a linearisation is defined as follows:

```
tid = threadIdx.x + threadIdx.y*blockDim.x + threadIdx.z*blockDim.y;
```

This is inserted at the beginning of the content of every iterative wrapping (Lines 10 and 27).

Again, the formal rules that define the transformations shown in the previous example are introduced:

Rule 6. Every declaration `T var;` of variables having as scope the entire kernel body are moved at the beginning of the body. Initializations are not moved.

```
void kernel(...)
{
    ITERATIVE_WRAPPING{
        T var_1;
        ...
        T var_i;
        ...
        T var_j = expr;
        ...
    }
}
```

$\longrightarrow$

```
void kernel(...)
{
    T var_1;
    T var_i;
    T var_j;
    ITERATIVE_WRAPPING{
        ...
        ...
        var_j = expr;
        ...
    }
}
```

Rule 7. For every declaration having the `__shared__` keyword, the keyword is removed.

`__shared__ T var` $\longrightarrow$ `T var`

Rule 8. The `tid` variable is declared at the beginning of the body, and its initialization added at the beginning of each iterative wrapping.

```
void kernel(...)                          void kernel(...)
{                                         {
    ...                                       int tid;
    ITERATIVE_WRAPPING{                       ...
        ...                    ⟶             ITERATIVE_WRAPPING{
    }                                             tid = ...;
    ...                                           ...
}                                             }
                                              ...
                                          }
```

Rule 9. The dimensionality of the formerly thread-local variables is augmented in their declaration.

$$\texttt{T var;} \longrightarrow \texttt{T var[];}$$

Rule 10. At each reference `var` to a variable which dimensionality was augmented in the declaration, is added an access by the value of `tid`.

$$\texttt{var} \longrightarrow \texttt{var[tid]}$$

Operating this last transformations, the kernel function is pure C++ syntax, with the same semantics of the CUDA version, and thus executable on different architectures.

2.2.4 Host translation

This section describes how the kernel invocation on the host side of the CUDA source is transformed, in order to iterate through the block indexes and call the translated kernel function once for every block.

Memory management instructions are temporarily ignored, since they are interesting more from an implementation point of view (as described in Chapters 4 and 5). Thus the mapping of the host code resides mainly in handling the kernel calls.

In Listing 2.8 is shown a basic example of a kernel call.

```
1 int main(){ ...
2   kernel<<<blocksPerGrid, threadsPerBlock>>>(...);
3   ... }
```

Listing 2.8: Kernel invocation example.

It is clear that the kernel execution configuration, identified by the triple angle bracket notation, is the main aspect that has to be managed in order to perform the translation.

The corresponding translation is shown in Listing 2.9:

```
1 int main()
2 {
3   ...
4   for(i=0; i < blocksPerGrid; i++)
5     kernel(..., threadsPerBlock, i);
6   ...
7 }
```

Listing 2.9: Translated kernel invocation.

By doing this transformation, the semantics of the host code is preserved, since the kernel function is executed for every block. The new function call shows two new parameters, `threadPerBlock` and the `i` index of the for loop (Line 5). This is coherent with the formal parameters added to the kernel definition in the previous section (`blockdim` and `blockIdx`).

Again, remembering that also `blocksPerGrid` is a three-dimensional variable, in the general scenario, the wrapping of the kernel invocation is constituted of three nested loops, each one iterating on the value of one of the three components. This is shown in Listing 2.10.

```
1  int main()
2  {
3    ...
4    for(z=0; z < blocksPerGrid.z; z++){
5      for(y=0; y < blocksPerGrid.y; y++){
6        for(x=0; x < blocksPerGrid.x; x++){
7          kernel(..., threadsPerBlock, dim3(x,y,z));
8        }
9      }
10   }
11   ...
12 }
```

Listing 2.10: Translated kernel invocation in the three-dimensional scenario.

The last argument in the kernel call (Line 7), corresponding to the `blockIdx` parameter in the kernel definition, is now a variable of type `dim3`, constructed at each kernel invocation with the values of the three indexes.

It is worth pointing out that this transformation for the kernel call results now in a serialization also of the blocks execution. According to the programming model of CUDA, since blocks execute independently, the main idea behind achieving block parallelism relies on the fact that concurrent CPU threads will execute the kernel function. This is done in practice relying on the Intel Thread Building Blocks (TBB) template library, substituting the three for loops wrapping the kernel invocation with a `parallel_for` pattern. This substitution goes beyond the topic of this chapter, since does not involves abstract syntax transformations of the CUDA input code, and will be discussed in detail in Chapters 4 and 5.

Summary

In the first section of this chapter we discussed the reasons behind the choice of doing a source-to-source approach, highlighting the portability advantages and describing the AST representation.

In the second section, we formalized the set of transformations to apply to the CUDA programming constructs in order to obtain the equivalent C++ version. First we introduced the concept of wrapping the kernel content in an iterative structure. Then, we shown how closing and reopening such iterations can ensure synchronization. Finally, we discussed how to emulate the former thread-local variables and how to transform the kernel invocation.

Chapter 3

Clang

This chapter describes Clang, the tool we used for the syntactic analysis of the source code.

After a brief overview of the framework in the first section, the second section focuses more on Clang's Abstract Syntax Trees, giving a basic description of the nodes and how the Clang ASTs are structured and represented. The third section lists the Clang libraries used in our implementation.

A full description of Clang and its ASTs goes beyond the topic of this thesis work. For more details it is possible to consult the user manual [3] or the official documentation [2].

3.1 Description

Clang is a compiler front-end for the C language family, including C++, Objective-C and Objective-C++. It is written in modern C++ and it is part of the **LLVM** compiler framework. The LLVM Project [6] is a collection of modular and reusable compiler and toolchain technologies. The LLVM Core libraries provide code generation support for many architectures, along with a target-independent optimizer. Clang uses LLVM as its back-end and is part of the LLVM release cycle. Clang is open-source and developed by Apple but other companies such as Microsoft and Google are involved.

Instead of being a monolithic compiler binary like gcc, Clang has a library-based and modular design [14], which makes it more flexible and easy to embed into other applications. It supports different uses such as code refactoring, static analysis and code generation, and offers fast compilation and low memory consumption. Nevertheless, the libraries may be used independently from the driver to create other source-level tools.

Among the various dialects of the C language family supported, Clang now allows to parse also CUDA C/C++ syntax. This is a fundamental feature for this work, since it relieves us from modifying Clang's parsing engine in order to make it able to handle CUDA code. It also allows to benefit from the advantages of the Clang's AST representation, as explained in the next section.

3.2 Clang's AST

As stated in Chapter 2, Clang's ASTs are a good fit for the syntactic representation. This is due to the fact that they resemble more the written C++ code. In a certain sense, Clang's ASTs are "less abstract" than the ASTs produced by other compilers. For example, parenthesis expressions and compile time constants are available in an unreduced form in the AST. This makes Clang's AST a good fit for refactoring tools.

Listing 3.1 shows a simple example of a while loop.

```
1 int main(){
2   int x;
3   while( x < 10 ) { x = x + 1; }
4 }
```

Listing 3.1

A simple way to transform Clang AST into a textual representation is by using the Clang compiler itself with the command

```
clang -Xclang -ast-dump -fsyntax-only <input file>
```

Listing 3.2 shows the AST produced by Clang for the code shown in Listing 3.1.

```
1  TranslationUnitDecl 0x89d0820 <invalid sloc> <invalid sloc>
2  ...omitting Clang internal declarations...
3  `-FunctionDecl 0x89d1240 <while.cpp:1:1, line:7:1> line:1:5
       main 'int (void)'
4    `-CompoundStmt 0x8a14c98 <col:11, line:7:1>
5      |-DeclStmt 0x89d13f8 <line:2:1, col:6>
6      | `-VarDecl 0x89d1398 <col:1, col:5> col:5 used x 'int'
7      `-WhileStmt 0x8a14c78 <line:3:1, line:6:1>
8        |-<<<NULL>>>
9        |-BinaryOperator 0x89d1470 <line:3:8, col:12> '_Bool'
             '<'
10       | |-ImplicitCastExpr 0x89d1458 <col:8> 'int' <
             LValueToRValue>
11       | | `-DeclRefExpr 0x89d1410 <col:8> 'int' lvalue Var 0
             x89d1398 'x' 'int'
12       | `-IntegerLiteral 0x89d1438 <col:12> 'int' 10
13       `-CompoundStmt 0x8a14c58 <line:4:1, line:6:1>
14         `-BinaryOperator 0x8a14c30 <line:5:2, col:10> 'int'
               lvalue '='
15           |-DeclRefExpr 0x89d1498 <col:2> 'int' lvalue Var 0
                 x89d1398 'x' 'int'
16           `-BinaryOperator 0x89d1520 <col:6, col:10> 'int'
                 '+'
17             |-ImplicitCastExpr 0x89d1508 <col:6> 'int' <
                   LValueToRValue>
18             | `-DeclRefExpr 0x89d14c0 <col:6> 'int' lvalue
                   Var 0x89d1398 'x' 'int'
19             `-IntegerLiteral 0x89d14e8 <col:10> 'int' 1
```

Listing 3.2: AST dump produced by Clang for the code shown in Listing 3.1

The nodes of the Clang AST are organized in multiple class hierarchies, which do not share a common base class.

The basic nodes of the Clang AST are `Decl` (i.e. a declaration), `Stmt` (i.e. a statement), `Expr`[1] (i.e. an expression) and `Type` (i.e. a type). These

[1]Note that expressions (`Expr`) are also statements (`Stmt`) in Clang's AST node hierarchy.

basic nodes are the base classes for rather large class hierarchies. There are also a multitude of nodes in the AST that are not part of a larger hierarchy, and are only reachable from specific other nodes, like `CXXBaseSpecifier`. A full specification of the Clang AST node hierarchy is available in [5].

The main entry point into the Clang AST are the translation units (i.e. a preprocessed input file with all its headers). In Listing 3.2 this is identified by the `TranslationUnitDecl` node (Line 1), that can contain function declarations, type declarations or declarations of global variables.

Once started from the `TranslationUnitDecl` node, the full AST is recursively traversed through the class `RecursiveASTVisitor`.

Line 3 of Listing 3.2 shows for example a `FunctionDecl` node, which corresponds to the declaration of the `main` function on Line 1 of Listing 3.1. On Line 4 it is possible to observe a `CompoundStmt` node, corresponding to the presence of a statement block surrounded by curly braces, containing a `DeclStmt` node (Lines 5–6), which refers to the declaration of the `x` variable, and a `WhileStmt` node (Line 7). It is possible to observe that every node, in addition to its name, carries a set of additional informations: an unambiguous address and the source code location(s) referring to the node. If the node represents a declaration (i.e. Line 6), the name of the declared entity is reported, as well as its type and a field indicating whether is used or not. If it is a function declaration, also the type of the parameters is reported. If the node represents an operator (i.e. the `BinaryOperator` node in Line 9), the return type and the name of the operator are listed. If the node represents a Literal (`IntegerLiteral` node, Line 12) the type and the value are shown.

In order to obtain a Clang AST from a CUDA program, the CUDA runtime libraries have to be included when the input file is parsed by Clang. Listing 3.3 shows a snippet of code containing some CUDA reserved keywords (in red).

```
1 __global__ void kernel(){
2   __shared__ int some_array[10];
3 }
4
5 int main(){
6   kernel<<<1, 1>>>();
7 }
```

Listing 3.3

The obtained Clang AST is showed below in Listing 3.4, where the addresses and the source locations are hidden for readability.

```
1  |-FunctionDecl 0x95eac58 [...] used kernel 'void (void)'
2  | |-CompoundStmt [...]
3  | | `-DeclStmt [...]
4  | |   `-VarDecl [...] some_array 'int [10]' static
5  | |     `-CUDASharedAttr [...]
6  | `-CUDAGlobalAttr [...]
7  `-FunctionDecl [...] main 'int (void)'
8    `-CompoundStmt [...]
9      `-CUDAKernelCallExpr [...] 'void'
10       |-ImplicitCastExpr [...] 'void (*)(void)' <
             FunctionToPointerDecay>
11       | `-DeclRefExpr [...] 'void (void)' lvalue Function 0
             x95eac58 'kernel' 'void (void)'
12       `-CallExpr [...] 'cudaError_t':'enum cudaError'
13         |-ImplicitCastExpr [...] 'cudaError_t (*)(dim3, dim3
               , size_t, cudaStream_t)' <FunctionToPointerDecay>
14         | `-DeclRefExpr [...] 'cudaError_t (dim3, dim3,
               size_t, cudaStream_t)' lvalue Function [...] '
               cudaConfigureCall' 'cudaError_t (dim3, dim3,
               size_t, cudaStream_t)'
15         ...
```

Listing 3.4: Simplified Clang AST dump for the code in Listing 3.3

It is possible to observe in the Listing above that Clang treats the CUDA specific syntax adding specialised nodes to the hierarchy, but always matching them with the corresponding nodes of the regular C++ syntax. This is a clear symptom of the CUDA nature of being an extension of C++. For example, the kernel declaration results in a normal function declaration, having in addition the `CUDAGlobalAttr` node. The same happens for the declaration of a shared variable, simply identified by the node `CUDASharedAttr` (Lines 3–5). The kernel call is identified by the `CUDAKernelCallExpr` node, which is just a subclass of a normal `CallExpr` in the node hierarchy.

It is worth pointing out that Clang's AST are immutable, meaning that their correctness is not guaranteed after any operation on its nodes, like reordering, insertion of removal. This property impacts the design of the solution presented in this thesis, as explained in detail in Section 4.1 of Chapter 4.

3.3 Clang libraries

This section briefly describes the libraries provided by Clang used in this work. Again, a full description is out of the scope of this thesis. For a complete overview please refer to [1].

libbasic

This library provides fundamental features such as diagnostics and file system caching for input source files. It also contains the `SourceLocation` class, which encodes the concept of a location in the source code. Technically, a source location is simply an offset into the manager's view of the input source, which is all input buffers (including macro expansions) concatenated in an effectively arbitrary order.

liblex

This library provides lexing and preprocessing features. The `Lexer` class provides a simple interface that turns a text buffer into a stream of tokens. This is exploited in the `Preprocessor` class, in order to compute the source location just past the end of a certain token. The `Preprocessor` also handles pragmas and macro expansion.

libast

libast provides classes to represent the C AST (the node class hierarchy shown in the previous section), the C type system, builtin functions, and various helpers for analyzing and manipulating the AST (visitors, pretty printers, etc). In particular, it contains the `ASTConsumer` class, an abstract interface that should be implemented by clients that read ASTs. This abstraction layer allows the client to be independent of the AST producer (e.g. parser vs AST dump file reader, etc). This library also contains the `ASTContext` class, which bundles all the informations about the AST for a translation unit. This class allows traversal of the whole translation unit starting from the `getTranslationUnitDecl` method.

librewrite

This library provides editing of text buffers, important for code rewriting transformation. The `Rewriter` class is the main interface to the rewrite

buffers. Its primary job is to dispatch high-level requests such as text insertions or removals to the low-level `RewriteBuffers` that are involved.

libtooling

LibTooling is a library to support writing standalone tools based on Clang. Tools built with LibTooling, like Clang plugins, run front-end actions over code. For a standalone tool to run Clang, it first needs to figure out what command line arguments to use for a specified file. To that end there is the `CommonOptionsParser` class that takes the responsibility to parse command-line parameters.

Summary

In the first section of this chapter we gave a brief description of the Clang compiler front-end, while in the second section we focused on the description of its ASTs. We shown how a Clang AST is represented and in particular how it represents CUDA syntax. In the last section we gave an overview of the libraries provided by Clang.

Chapter 4

Prototype design

This chapter gives an high-level description of the software implemented as a proof of concept of the transformations described in this thesis work. The first section gives a high level overview of the software, while the second and the third section describe in more detail the Augmenter and the Translator, the two main components of the software.

4.1 Overview

The overall translation process goes through a set of two steps and involves three files, as shown in Figure 4.1.

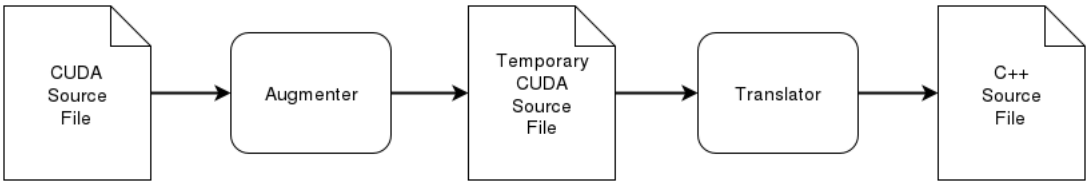

Figure 4.1: Translation process overview.

The CUDA source code files are given in input to the first component, the Augmenter, which applies a set of preliminary transformations, mainly related to the emulation of thread-local memory, as described in more detail in Section 4.2. The Augmenter writes as output a temporary file containing this preliminary changes to the CUDA code.

This temporary file is therefore given in input to the second component, the Translator, which has the task of implementing the main translation process, as explained in Section 4.3. The produced output is then a C++ source code file, ready to be compiled and executed on a CPU.

Both Expander and Translator are Clang plugins, since they rely on Clang's parser in order to create the AST. Then they apply the changes exploiting the tools provided by the Clang libraries. The two plugins traverse the trees in a recursive descent way through iterators on the nodes of the tree.

Splitting the overall translation process in two components has been necessary because the operations performed by the Augmenter involve the displacement and the reordering of instructions of the input code. This is also reflected on the structure of the AST of the program, since moving instructions on the code means reordering the nodes of the tree. However, modifying the structure of a Clang AST should be avoided, since Clang ASTs are designed to be immutable, thus it is not possible to guarantee their correctness after any node reconstructing. For this reason, the output of the Augmenter is saved on a temporary file, and then parsed again by the Translator. However, being this done at compile time, the delay introduced is negligible.

4.2 Augmenter

Figure 4.2 shows more in detail the Augmenter component.

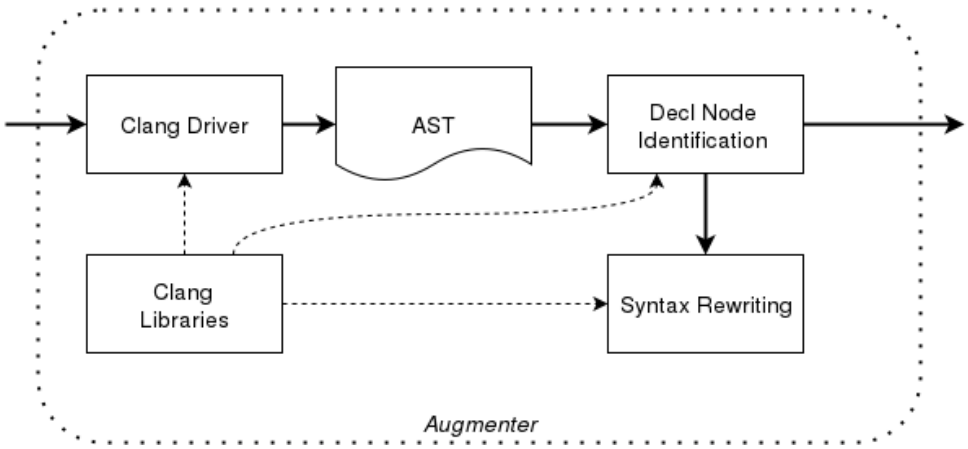

Figure 4.2: An expanded view of the Augmenter component.

The content of the CUDA source code files is given to Clang's main driver. Clang handles the parsing and the AST generation as during normal compilation.

Since the AST representation is obtained, the Augmenter starts recursively the traversal of the tree. For every node encountered, it checks if it is a declaration node (Decl Node Identification, Fig. 4.2). The declaration nodes also having an attribute pointing out the allocation on shared memory are discarded, and the attribute deleted. This deletion can be considered already part of the Syntax Rewriting stage. Every declaration found is therefore stored in a temporary data structure. In the Decl Node Identification stage is checked also the presence of expressions that refer to previously declared variables (i.e. the usage of a variable). In particular, only the expressions referring to the declarations stored in the temporary data structure and not having the "shared" attribute are considered.

The Syntax Rewriting phase shown in Fig. 4.2 is performed initially on the declarations, augmenting their dimensionality (i.e. scalar variables become arrays). Secondly, the nodes containing expressions referring to declared variables, identified by the previous stage, are modified adding an access by thread ID. This two operations were described in Section 2.2.3 of Chapter 2. All the declaration statements in the temporary data structure that have also an initialization part are split in two. The declarations are moved at the beginning of the body of the CUDA kernel, in order to avoid

their presence inside the iterative wrapping that will be applied by the translator, as explained in Section 4.3. The initialization part is instead left at the original location of the code.

The source code containing the changes applied by the Augmenter is written in output on a temporary file, which will be given as input to the second component, the Translator.

4.3 Translator

The Translator takes in input the temporary file containing the source code produced by the Augmenter. As for the previous component, Clang handles the parsing of the code and the generation of the AST like in a normal compilation scenario.

Figure 4.3 shows the stages composing the Translator.

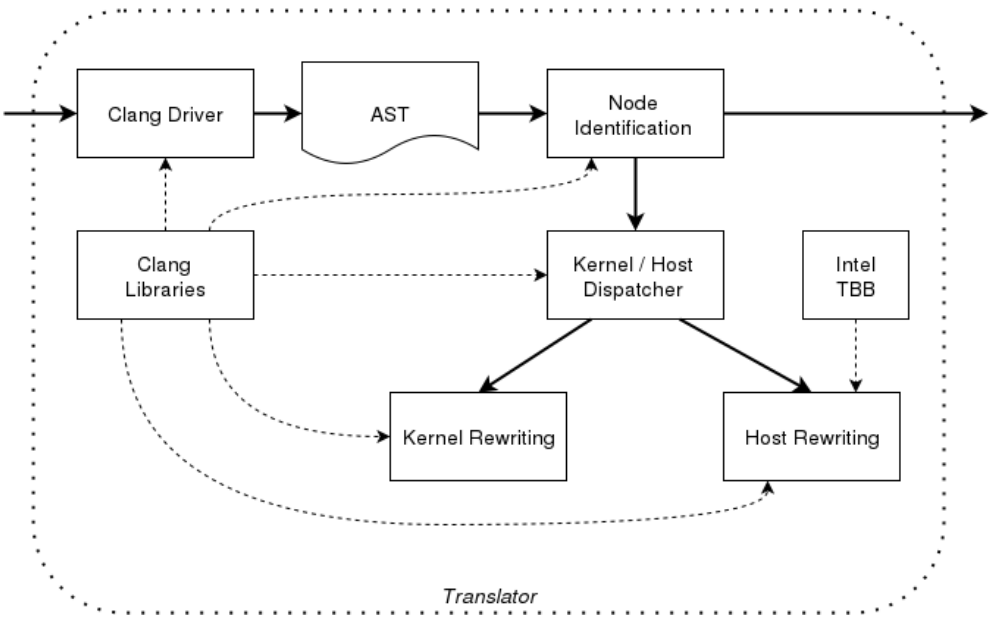

Figure 4.3: An expanded view of the Translator component.

The AST is traversed again in a recursive descent way. The Node Identification stage has a very similar behaviour of the Decl Node Identification stage of the Augmenter depicted in Figure 4.2. However, the Translator has now to take into account not only the declarations but also the expression and statement nodes. The identified node is therefore passed to the Kernel / Host Dispatcher, which checks if the node belongs to the kernel part or the

host part of the code. This is accomplished simply analysing if the function containing the node being processed is declared with the `__global__` attribute.

The Kernel Rewriting stage performs the transformations described in Section 2.2.1 and 2.2.2. First of all, for each declaration node processed, any CUDA specific attribute such as `__global__` or `__shared__` is removed. In particular, the declaration of the kernel function is rewritten adding the block dimension and the block identifier to the formal parameter list. The rewriting of the body of the kernel consists mainly in the insertion of the iterative wrapping, namely the three nested for loops iterating among the values of the identifier of the thread shown in Listing 2.4. Since the augmenter moved all the declarations at the beginning of the kernel body, the iterative wrapping starts exactly after the last declaration, and finishes after the last statement of the body, inserting the brackets that close the loops. Therefore, if a node corresponds to a `__syncthreads()` expression, the rewriter deletes this function call and inserts the brackets closing the iterative wrapping, immediately opening another iterative wrapping.

The Host Rewriting stage performs two operations corresponding to the transformation shown in Section 2.2.4. The first operation is related to the transformation of the kernel function call. Both three-dimensional parameters representing the grid dimension and the blocks dimension, as well as the triple angle bracket notation, are deleted. In particular the block dimension is now inserted as a regular argument of the function call. The second operation consists in the insertion of the iterative wrapping of the new function call, which will be described in Section 4.3.1.

The Host Rewriting stage is also in charge of the rewriting of the CUDA API calls. As said in Section 1.2, the CUDA programming model assumes a system composed of a host and a device, each with their own separate memory. Kernels operate out of device memory, so the runtime provides functions to allocate, deallocate, and copy device memory, as well as transfer data between host memory and device memory. The most important functions are the ones concerning memory management. In particular, the most prominent three are considered: `cudaMalloc`, `cudaFree` and `cudaMemcpy`. When these functions are encountered, they are replaced with the corresponding C++ standard library functions, as shown in Table 4.1:

Although the translation of `cudaMalloc` and `cudaFree` is quite intuitive and straightforward, the `cudaMemcpy` call could be simply ignored, since

CUDA	C++
`cudaMalloc(devPtr, size)`	`devPtr = (type) malloc(size)`
`cudaFree(devPtr)`	`free(devPtr)`
`cudaMemcpy(dst, src, size, kind)`	`memcpy(dst, src, size)`

Table 4.1: Memory management API rewriting.

now the execution happens in an environment that does not distinguish any more between host and device memory. Thus, it can be given as a parameter to the kernel just the (former) host memory data. However, since there is no guarantee that the kernel function left unmodified the content of its parameters, this could lead to an incorrect behaviour, if this data is used again after the kernel execution. For this reason, the `memcpy` between the two memory areas is still needed.

4.3.1 Intel TBB parallelization

The Host Rewriting stage of the Translator, handles the wrapping of the kernel function call, adding the three nested for loops iterating among the values of the former grid dimension parameter, as described in Listing 2.10. The index of those loops is also added to the arguments of the function call.

As previously anticipated at the end of Chapter 2, the main idea behind achieving block parallelism relies on the fact that concurrent CPU threads will execute the kernel function.

This section introduces how this is accomplished relying on the **Intel Thread Building Blocks** (TBB) template library. TBB is a C++ template library, developed by Intel, for writing software programs that take advantage of multi-core processors. A full description of the TBB framework goes out of the scope of this thesis work. For more informations please refer to [4].

Among the various parallel patterns provided by TBB, the template function `tbb::parallel_for` showed to be the most straightforward way to parallelize the execution of the kernel function.

When a function has to be applied to each element of an array, and it is safe to process each element concurrently, a `tbb::parallel_for` can be used to implement this functionality, but with parallelism enabled.

For example, Listing 4.1 shows a function `foo` applied to each of the `n`

elements of an array `a`.

```
1 for( size_t i=0; i!=n; ++i )
2         foo(a[i]);
```

Listing 4.1

The iteration space here is of type `size_t`, and goes from `0` to `n-1`. The template function `tbb::parallel_for` breaks this iteration space into chunks, and runs each chunk on a separate thread. `tbb::parallel_for` takes four parameters: `first`, `last`, `step` and `function`. Listing 4.2 illustrates how the `foo` application is transformed:

```
1 parallel_for(size_t(0), n, size_t(1) , [=](size_t i) {
2   foo(a[i]);
3 });
```

Listing 4.2: A `parallel_for` usage example.

The `[=]` introduces a lambda expression. The expression creates a function object that, when local variables like `a` and `n` are declared outside the lambda expression, but used inside it, captures them as fields inside the function object. The `[=]` specifies that capture is by value.

To show how `tbb:parallel_for` is used, below is listed again the wrapping of the kernel invocation (previously shown in Listing 2.10):

```
1  int main()
2  {
3    ...
4    for(z=0; z < blocksPerGrid.z; z++){
5      for(y=0; y < blocksPerGrid.y; y++){
6        for(x=0; x < blocksPerGrid.x; x++){
7          kernel(..., threadsPerBlock, dim3(x,y,z));
8        }
9      }
10   }
11   ...
12 }
```

Listing 4.3: Serial invocation of the kernel function.

In Listing 4.4 below, the for loops are replaced by a call to the function `tbb::parallel_for`, which divides up the iterations in to tasks and provides them to the library's task scheduler for parallel execution. The body of the loop in the code stays the same (Line 5).

```
...
tbb::parallel_for(0, (int)blocksPerGrid.z, 1, [=](int z){
  tbb::parallel_for(0, (int)blocksPerGrid.y, 1, [=](int y){
    tbb::parallel_for(0,(int)blocksPerGrid.x, 1, [=](int x){
      kernel(..., threadsPerBlock, dim3(x, y, z));
    });
  });
});
...
```

Listing 4.4: Parallel invocation of the kernel function.

Applying this substitution, the execution of the `kernel` function is parallelized, since the block indexes are partitioned arbitrarily among concurrently executing CPU threads.

Summary

In this chapter we gave a description of the design of the software implemented in this master thesis. After having briefly shown the overall process in the first section, we highlighted the execution stages involved in the two main components: the Augmenter and the Translator. In particular, in the third section, we also shown how in the Translator stage CPU thread parallelism is achieved through Intel TBB.

Chapter 5

Implementation

This chapter describes the implementation details of the tool created to accomplish the source-to-source translation.

In the first section is presented a part of the code common to the two tools, mainly concerning the creation of a Clang plugin.

The second and the third section show the implementation of the Augmenter and the Translator plugins.

5.1 Clang machinery

This section describes some parts of the code that are necessary for the creation of the two tools. Since they are identical for both tools, they are reported only once. The first part in common consists in the `main` function, shown in Listing 5.1.

```
1 int main(int argc, const char **argv) {
2 CommonOptionsParser op(argc, argv, MatcherSampleCategory);
3 ClangTool Tool(op.getCompilations(),op.getSourcePathList());
4 return Tool.run(newFrontendActionFactory<MyFrontendAction>()
    .get());
5 }
```

Listing 5.1: The `main` function of a Clang plugin.

At Line 2 there is the creation of a `CommonOptionParser` object named `op`, which is a parser for options common to all command-line Clang tools. At Line 3 is called the constructor for the `ClangTool` class. This is an utility to run a `FrontendAction` over a set of files. This is done at Line 4, where the `run` method of the `ClangTool` class is called. It takes as parameter an `ASTFrontendAction`.

Listing 5.2 shows the definition of the `MyFrontendAction` class.

```
1  class MyFrontendAction : public ASTFrontendAction{
2  public:
3    MyFrontendAction(){}
4    void EndSourceFileAction() override {
5      TheRewriter.getEditBuffer(TheRewriter.getSourceMgr().
         getMainFileID()).write(llvm::outs());
6    }
7
8    std::unique_ptr<ASTConsumer> CreateASTConsumer(
        CompilerInstance &CI, StringRef file) override {
9      TheRewriter.setSourceMgr(CI.getSourceManager(), CI.
         getLangOpts());
10     return llvm::make_unique<my_ast_consumer>(&CI, &
         TheRewriter);
11   }
12 private:
13   Rewriter TheRewriter;
14 };
```

Listing 5.2: The `ASTFrontendAction` class definition.

Lines 4–6 define the callback at the end of processing an input file. Lines 8–11 show the creation of an ASTConsumer object. This is an abstract interface that should be implemented by clients that read ASTs. The semantics of the operations performed by the two tools are inside this class definition, and they will be described in the next sections.

5.2 Augmenter

This section describes the implementation details of the Augmenter component. This class takes care of augmenting the dimensionality of the local CUDA thread variables and of the movement of the declarations in the right locations.

```
1  std::string initsupport;
2  std::string nt;
3  std::set<std::string> KernelDecls;
4  std::vector<std::string> NewDecls;
5  SourceLocation kernelbodystart;
6
7  class dimensionality_augmenter : public ASTConsumer {
8  public:
9    dimensionality_augmenter(CompilerInstance *comp, Rewriter
       * R) : ASTConsumer(), CI(comp), Rew(R) { }
10   virtual ~dimensionality_augmenter() { }
```

Listing 5.3: Preliminary part of the `dimensionality_augmenter` class.

Listing 5.3 shows the preliminary part of the `dimensionality_augmenter` class. At Lines 1–5 are declared some global support variables that will be needed in the following of the code. Line 7 shows the declaration of the class, with the constructor at Line 9, containing the initialization of some Clang-specific components, such as the Rewriter.

```
1  virtual bool HandleTopLevelDecl(DeclGroupRef DG) {
2    //Walk and rewrite declarations in group
3    for (DeclGroupRef::iterator i = DG.begin(), e = DG.end();
       i != e; ++i) {
4      //Handles globally defined functions
5      if (FunctionDecl *fd = dyn_cast<FunctionDecl>(*i)) {
6        if (fd->hasAttr<CUDAGlobalAttr>()) {
7          //kernel function
8          RewriteKernelFunction(fd);
9        } else {
```

```
10          if (Stmt *body=fd->getBody()) RewriteHostStmt(body);
11        }
12      }
13    }
14    return true;
15  }
```

Listing 5.4: The `HandleTopLevelDecl` method.

Listing 5.4 shows the first method of the class, `HandleTopLevelDecl`, automatically called by the Clang parser to process every top-level declaration, represented by the `DG` parameter of type `DeclGroupRef` (Line 1). This declaration group is iterated, searching for function declarations (Line 5). If a function is found, the presence of the `__global__` attribute is checked (Line 6) and if the result is true, meaning that a kernel function was identified, the `RewriteKernelFunction()` private method shown below is called. Otherwise, the `RewriteHostStmt` method (shown in Listing 5.8) is called on the body of the function.

```
1  void RewriteKernelFunction(FunctionDecl* kf) {
2    if (Stmt *body = kf->getBody()){
3      kernelbodystart = PP->getLocForEndOfToken(body->
           getLocStart());
4      replicate(body); //Call to the analysis of the variables
5    }
6  }
```

Listing 5.5: The `RewriteKernelFunction` method.

The `RewriteKernelFunction` method is shown in Listing 5.5. This method simply checks the presence of the body of the function (Line 2). If the function is not empty, the location of the kernel body is saved (Line 3) in the `kernelbodystart` global variable (declared at Line 5 of Listing 5.3). Therefore, the main method of this class, `replicate`, is called (Line 4).

```
1  void replicate(Stmt *s){
2    if (DeclStmt *ds = dyn_cast<DeclStmt>(s)){
3    DeclGroupRef DG = ds->getDeclGroup();
4    for (DeclGroupRef::iterator i2=DG.begin(), e=DG.end(); i2
         !=e; ++i2){
5      if(*i2){ //not null
6      if(VarDecl *vd = dyn_cast<VarDecl>(*i2)){
7      if (CUDASharedAttr *sharedAttr = vd->getAttr<
           CUDASharedAttr>()) {
```

```
8       SourceRange declrange = SourceRange(vd->
            getTypeSpecStartLoc(), PP->getLocForEndOfToken(vd->
            getLocEnd()));
9       StringRef decl_text = Lexer::getSourceText(
            CharSourceRange(declrange, false), *SM, *LO);
10      NewDecls.push_back(decl_text.str()+";");
11      SourceRange fullrange = SourceRange(SM->
            getExpansionLoc(vd->getLocStart()), PP->
            getLocForEndOfToken(PP->getLocForEndOfToken(vd->
            getLocEnd())));
12      CharSourceRange cs = CharSourceRange(fullrange,false);
13      Rew->RemoveText(cs);
14    } else { //augment
15      KernelDecls.insert(vd->getNameAsString()); //Matching
            set
16      NewDecls.push_back(vd->getType().getAsString()+" "+vd
            ->getNameAsString()+"[numThreads];");
17      if(vd->hasInit()){
18        Rew->ReplaceText(SourceRange(vd->getLocStart(), PP->
              getLocForEndOfToken(vd->getLocEnd())), vd->
              getNameAsString() + "[__ttid_] = " + getStmtText(
              vd->getInit()) + ";");
19      } else {
20        Rew->ReplaceText(SourceRange(vd->getLocStart(), PP->
              getLocForEndOfToken(vd->getLocEnd())), "");
21      }
22    }
23    }
24    }
25  }
26    [...]
27  }
```

Listing 5.6: The first part of the `replicate` method.

Listing 5.6 shows the first part of the `replicate` method. A generic statement `s`, is dynamically cast to a `DeclStmt` (Line 2) and then variable declarations are searched (Line 6). For every `VarDecl` found, two behaviours are possible regarding the presence or the absence of the `__shared__` attribute (Line 7). If the attribute is found, the corresponding variable does not need to have its dimensionality augmented. However, the attribute is deleted, and declaration needs to be moved at the beginning of the kernel

body. In order to accomplish that, a `SourceRange`[1] is created (Line 8), representing the part of code of the declaration without the attribute. The content of the range is therefore taken in textual representation (Line 9), and inserted (Line 10) in the `NewDecls` global vector (declared on Line 4 of Listing 5.3). Then, another range is calculated (Lines 11–12), this time representing the declaration statement in its entirety (including the attribute). Lastly, the Rewriter deletes the content of the range (Line 13).

If the `__shared__` attribute was not found (`else` branch, Line 14), it means we are dealing with a declaration of a local variable, thus its dimensionality needs to be augmented. First of all, at Line 15, the name of the declared variable is inserted in the `KernelDecls` set (declared on Line 3 of Listing 5.3). The declaration is therefore saved in the `NewDecls` vector as a string, adding the `"[numThreads]"` part at the end of it (Line 16). If the variable is initialized (Line 17), only the initialization part is left on this source location, adding the access by thread ID, represented by the `"[__ttid_]"` string, after the variable name (Line 18). Otherwise, the entire statement is deleted (Line 20).

```
1  void replicate(Stmt *s){
2    [...]
3    } else if(DeclRefExpr *dre = dyn_cast<DeclRefExpr>(s)){
4      if(KernelDecls.find(dre->getNameInfo().getAsString()) !=
           KernelDecls.end()){
5        Rew->ReplaceText(SourceRange(dre->getLocStart(), dre->
             getLocEnd()), dre->getNameInfo().getAsString()+"[
             __ttid_]");
6      }
7    }
8
9    for (Stmt::child_iterator s_ci = s->child_begin(), s_ce =
       s->child_end(); s_ci != s_ce; ++s_ci) {
10     if(*s_ci) replicate(*s_ci);
11   }
```

Listing 5.7: The second part of the `replicate` method.

The second part of the `replicate` method is shown in Listing 5.7. The generic statement `s` is cast to a `DeclRefExpr` (Line 3), which encodes the information about how a declaration is referenced within an expression. If the variable expression is found in the `KernelDecls` set (Line 4), it means

[1] A pair of two `SourceLocations`

we are dealing with a variable which dimensionality was augmented. In this case, similarly as the initializations in the previous case, an access by thread identifier is added (Line 5) after the name of the variable. Finally, an iterator is defined on the children of the statement `s` (Line 9). For every child node found, the `replicate` method is recursively called on it (Line 10).

```
1  void RewriteHostStmt(Stmt *s) {
2    if (Expr *e = dyn_cast<Expr>(s)) {
3      if (clang::CUDAKernelCallExpr *kce = dyn_cast<clang::
           CUDAKernelCallExpr>(e)) {
4        CallExpr *kernelConfig = kce->getConfig();
5        Expr *block = kernelConfig->getArg(1);
6        if(auto C = dyn_cast<CXXConstructExpr>(block)){
7          if(auto A = dyn_cast<DeclRefExpr>(C->getArg(0)->
               IgnoreImpCasts())){
8            if(VarDecl *vd = dyn_cast<VarDecl>(A->getDecl())){
9              if(CXXConstructExpr * cce = dyn_cast<
                   CXXConstructExpr>(vd->getInit())){
10               nt = getStmtText(cce->getArg(0)) + "*" +
                     getStmtText(cce->getArg(1)) + "*" +
                     getStmtText(cce->getArg(2));
11             }
12           }
13         }
14       }
15     }
16   } else {
17     for (Stmt::child_iterator CI = s->child_begin(), CE = s
         ->child_end(); CI != CE; ++CI) {
18       if (*CI) RewriteHostStmt(*CI);
19     }
20   }
21 }
```

Listing 5.8: The `RewriteHostStmt` method.

Listing 5.8 shows the `RewriteHostStmt` method, called by `HandleTopLevelDecl` in case the function declaration did not have the `__global__` attribute, as previously described in Listing 5.4. The purpose of this method, is the analysis of the kernel call statement in the main function, in order to obtain the value of the parameter that identifies the dimension of a block. For every generic statement `s`, this method applies a set of casts, checking if we are dealing with an expression (Line 2), then a

CUDA kernel call (Line 3). The call configuration, corresponding to the arguments inside the triple angle brackets, is therefore taken (Line 4), and its second argument is saved into the `block` variable (Line 5). More casts are performed in order to obtain the corresponding `DeclRefExpr` (Lines 6–7) and the declaration (Lines 8–9). At Line 10, the three values representing the block dimension are stored in the `nt` global string (declared at Line 2 of Listing 5.3). Again, an iterator is defined on the children of the statement `s` (Line 17) and for every child node found, the `RewriteHostStmt` method is recursively called on it (Line 19).

After the execution of the methods presented above, the transformations concerning the movement of the declarations at the beginning of the kernel body are stored in the temporary support structures globally declared (Listing 5.3). To effectively apply this modifications to the source code, the behaviour of the `EndSourceFileAction` method previously shown in Listing 5.2 is extended.

```
1  class Replication : public ASTFrontendAction{
2    [...]
3    void EndSourceFileAction() override {
4      initsupport += "\ndim3 threadIdx;\nint __ttid_;\n";
5      initsupport += "int numThreads = "+nt+";\n";
6      for(int i = 0; i < NewDecls.size(); i++){
7        initsupport += NewDecls[i] + "\n";
8      }
9     TheRewriter.InsertTextAfter(kernelbodystart,initsupport);
10   [...]
11 };
```

Listing 5.9: The `EndSourceFileAction` method for the `Replication` class.

Listing 5.9 above shows the new `EndSourceFileAction` method of the `Replication` class, the specific `ASTFrontendAction` for the Augmenter component. The parts not shown of the class definition are identical to the ones in Listing 5.2.

At Line 4 the declaration of the now explicit variable `threadIdx` is added to the `initsupport` string, as well as the one of the `__tid_` index to access the augmented variables. The declaration of the `numThreads` variable is added at Line 5, initialized with the value previously saved in the `nt` string, as explained in Listing 5.8. Also, all the declarations previously saved in the `NewDecls` vector are inserted in the in the `initsupport` string. Finally, the rewriter inserts the content of the string at the beginning of the body

of the kernel, identified by the `kernelbodystart` source location (saved in the `RewriteKernelFunction` method shown in Listing 5.5).

5.3 Translator

This section describes the implementation of the Translator component.

```
1  class Translator : public ASTConsumer {
2  public:
3    Translator(CompilerInstance *comp, Rewriter *R) :
       ASTConsumer(), CI(comp), Rew(R){ }
4    virtual ~Translator() { }
5
6  virtual bool HandleTopLevelDecl(DeclGroupRef DG) { [...] }
7
8  private:
9      std::string TL_START1 = "for(threadIdx.z=0; threadIdx.z
         < blockDim.z; threadIdx.z++){\n";
10     std::string TL_START2 = "for(threadIdx.y=0; threadIdx.y
         < blockDim.y; threadIdx.y++){\n";
11     std::string TL_START3 = "for(threadIdx.x=0; threadIdx.x
         < blockDim.x; threadIdx.x++){\n";
12     std::string TL_START = TL_START1+TL_START2+TL_START3+"
         __ttid_=threadIdx.x + threadIdx.y*blockDim.x +
         threadIdx.z*blockDim.y;";
13     std::string TL_END = "}}}";
```

Listing 5.10: Preliminary part of the `Translator` class.

Listing 5.10 describes the preliminary part of the `Translator` class. The first method, `HandleTopLevelDecl` (Line 6) is identical to the one for the Augmenter, shown in Listing 5.4 of the previous section. Lines 9–13 present the declaration of the strings containing the iterative wrapping needed in the following. The `TL_START` string represents the opening of the three nested loops, while `TL_END` represents the closing.

```
1  void RewriteKernelFunction(FunctionDecl* kf) {
2    std::string SStr;
3    llvm::raw_string_ostream S(SStr);
4    S << kf->getCallResultType().getAsString() << " " << kf->
       getNameAsString() << "(";
5    for( int j = 0; j < kf->getNumParams(); j++){
6      S << kf->getParamDecl(j)->getType().getAsString() << " "
         << kf->getParamDecl(j)->getQualifiedNameAsString()
```

```
        << ", ";
7   }
8   S << "dim3 gridDim, dim3 blockDim, dim3 blockIdx)";
9   SourceLocation start = SM->getExpansionLoc(kf->getLocStart
        ());
10  SourceLocation end = PP->getLocForEndOfToken(SM->
        getExpansionLoc(kf->getParamDecl(kf->getNumParams()-1)
        ->getLocEnd()));
11  SourceRange range(start, end);
12  Rew->ReplaceText(start, Rew->getRangeSize(range),S.str());
13  if (Stmt *body = kf->getBody()){
14    RewriteKernelBody(body, true);
15  }
16 }
```

Listing 5.11: The `RewriteKernelFunction` method.

Listing 5.11 shows the `RewriteKernelFunction` method, called by the `HandleTopLevelDecl` method in case a `__global__` attribute was found. This method handles the transformation of the heading of the kernel function. A string is created, inserting the original returned type and the name (Line 4), then all the original formal parameters (Lines 5–6). At Line 8, are added to the parameter list the previously built-in variables `gridDim`, `blockDim` and `blockIdx`. Then, the source range of the old heading is calculated (Lines 9–11), and the new string is inserted at its place (Line 12). If the body of the function is not empty (Line 13), the `RewriteKernelBody` method is called (Line 14).

```
1  void RewriteKernelBody(Stmt *s, bool first){
2    if(first){ //We are entering the method for the first time
3      SourceLocation begin = s->getLocStart();
4      if(CompoundStmt * cs = dyn_cast<CompoundStmt>(s)){
5        for(Stmt::child_iterator i = cs->body_begin(), e = cs
           ->body_end(); i!=e; ++i){
6          if(*i){
7            if(DeclStmt *vd = dyn_cast<DeclStmt>(*i)){
8              begin =PP->getLocForEndOfToken(vd->getLocEnd());
9            }
10           else break;
11         }
12       }
13     }
14     Rew->InsertTextAfter(begin, "\n"+TL_START+"\n");
15     Rew->InsertTextBefore(s->getLocEnd(), "\n"+TL_END+"\n");
```

```
16      first = false;
17    }
18
19    [...]
20
21    else if (CallExpr *ce = dyn_cast<CallExpr>(s)){
22      if(ce->getDirectCallee()->getNameAsString() == "
          __syncthreads"){
23        //simply closing and reopening a thread loop
24        std::string a = "\n"+TL_END+"\n//"+getStmtText(ce)+";\
            n"+TL_START+"\n";
25        ReplaceStmtWithText(ce, a, *Rew);
26      }
27    } else {
28      for (Stmt::child_iterator s_ci = s->child_begin(), s_ce
          = s->child_end(); s_ci != s_ce; ++s_ci) {
29        if(*s_ci){
30          RewriteKernelBody(*s_ci, false);
31        }
32      }
33    }
34  }
```

Listing 5.12: The `RewriteKernelBody` method.

The `RewriteKernelBody` method is introduced in Listing 5.12. This method defines the transformations performed in order to handle the synchronization statements. Line 2 shows the check for the boolean parameter `first`, needed to open the first iterative wrapping inside the kernel body. This is accomplished iterating through all the statements encountered (Line 5), and updating the `begin` source location for each declaration found (Lines 7–8). After the last declaration, the iterative wrapping is introduced (Line 14). This behaviour is correct since the Augmenter stage moved all the global declarations at the beginning of the kernel body. The iterative wrapping is therefore closed at the end of the body (Line 15). In the general scenario, the `RewriteKernelBody` method casts the statement `s` dynamically, in order to search for `__syncthreads()` calls (Lines 21–22). If such a call is found, a string containing the closing and reopening of the iterative wrapping is inserted (Lines 24–25). Finally, an iterator is defined on the children of the statement `s` (Line 28). For every child node found, the `RewriteKernelBody` method is recursively called on it (Line 30).

```
1  void RewriteHostStmt(Stmt *s){
2      if (Expr *e = dyn_cast<Expr>(s)) {
3          std::string str;
4          if (RewriteHostExpr(e, str)) ReplaceStmtWithText(e,
               str, *Rew);
5      } else {
6          for (Stmt::child_iterator CI = s->child_begin(), CE
               = s->child_end(); CI != CE; ++CI) {
7              if (*CI) RewriteHostStmt(*CI);
8          }
9      }
10 }
```

Listing 5.13: The `RewriteHostStmt` method.

Listing 5.13 describes the `RewriteHostStmt` method. This method searches for expressions (Line 2) and if it finds one calls the `RewriteHostExpr`, method described in Listing 5.14, which modifies the `str` parameter, therefore passed to the `ReplaceStmtWithText` method which substitutes the original epxression `e` with the new string. Then, an iterator is defined on the children of the statement `s` (Line 6). For every child node found, the `RewriteHostStmt` method is recursively called on it (Line 7).

```
1  bool RewriteHostExpr(Expr *e, std::string &newExpr) {
2
3      SourceRange realRange(SM->getExpansionLoc(e->getLocStart
           ()), SM->getExpansionLoc(e->getLocEnd()));
4
5      //Rewriter used for rewriting subexpressions
6      Rewriter exprRewriter(*SM, *LO);
7
8      if (clang::CUDAKernelCallExpr *kce = dyn_cast<clang::
           CUDAKernelCallExpr>(e)) {
9          newExpr = RewriteCUDAKernelCall(kce);
10         return true;
11     } else if  (CallExpr *ce = dyn_cast<CallExpr>(e)) {
12         if (ce->getDirectCallee()->getNameAsString().find("
               cuda") == 0) { //CUDA API
13           return RewriteCUDACall(ce, newExpr);
14         }
15     }
16     bool ret = false;
17     //Do a DFS, recursing into children, then rewriting this
           expression
```

```
18      //if rewrite happened, replace text at old sourcerange
19      for (Stmt::child_iterator CI = e->child_begin(), CE = e
            ->child_end(); CI != CE; ++CI) {
20          std::string s;
21          Expr *child = (Expr *) *CI;
22          if (child && RewriteHostExpr(child, s)) {
23              //Perform "rewrite", which is just a simple
                    replace
24              ReplaceStmtWithText(child, s, exprRewriter);
25              ret = true;
26          }
27      }
28      newExpr = exprRewriter.getRewrittenText(realRange);
29      return ret;
30  }
```

Listing 5.14: The `RewriteHostExpr` method.

The `RewriteHostExpr` method is shown in Listing 5.14. After calculating the range of the expression being analysed (Line 3), this method searches for CUDA kernel invocations (Line 9), and if it finds one calls the `RewriteCUDAKernelCall` method (Line 10). If a normal call expression is found, then it is checked if the name of the called function begins with the string `"cuda"`, meaning a CUDA API was found, and then the method `RewriteCUDACall` is called. Otherwise, the `RewriteHostExpr` is recursively called on the subexpressions, if any.

```
1   std::string RewriteCUDAKernelCall(clang::CUDAKernelCallExpr
        *kernelCall) {
2       CallExpr *kernelConfig = kernelCall->getConfig();
3       Expr *grid = kernelConfig->getArg(0);
4       Expr *block = kernelConfig->getArg(1);
5       std::string SStr;
6       std::string tbb_wrap_before (
7         "tbb::parallel_for(0, (int) " + getStmtText(grid) + ".
              z, 1, [=](int __z_){\n"
8         "tbb::parallel_for(0, (int) " + getStmtText(grid) + ".
              y, 1, [=](int __y_){\n"
9         "tbb::parallel_for(0, (int) " + getStmtText(grid) + ".
              x, 1, [=](int __x_){\n");
10      llvm::raw_string_ostream S(SStr);
11      S << tbb_wrap_before;
12      S << getStmtText(kernelCall->getCallee()) << "(";
13      for(int i = 0; i < kernelCall->getNumArgs(); i++){
```

```
14          S << getStmtText(kernelCall->getArg(i)) << ", ";
15      }
16      S << getStmtText(grid) << ", " << getStmtText(block) <<
            ", dim3(__x_, __y_, __z_));";
17      std::string tbb_wrap_end ("\n});\n});\n});\n");
18      S << tbb_wrap_end;
19      return S.str();
20  }
```

Listing 5.15: The `RewriteCUDAKernelCall` method.

The `RewriteCUDAKernelCall` is shown in Listing 5.15. This method handles the kernel invocation expression, applying the parallelization through the
`tbb::parallel_for` template function insertion described in Section 4.3.1 of Chapter 4. A new string for the whole kernel invocation is created, adding the beginning of the wrapping (Line 12), the name of the function (Line 13) and the formal parameters (Lines 14–15) including the dimension of the gird, the dimension of the block and the index of the loop (Line 17). Therefore the wrapping closure is inserted (Line 19) and the new string is returned.

```
1   bool RewriteCUDACall(CallExpr *cudaCall, std::string &
        newExpr) {
2     std::string funcName = cudaCall->getDirectCallee()->
          getNameAsString();
3     if(funcName == "cudaMemcpy"){
4       Expr *dst = cudaCall->getArg(0);
5       Expr *src = cudaCall->getArg(1);
6       Expr *count = cudaCall->getArg(2);
7       std::string newDst, newSrc, newCount;
8       RewriteHostExpr(dst, newDst);
9       RewriteHostExpr(src, newSrc);
10      RewriteHostExpr(count, newCount);
11      newExpr = "memcpy("+newDst+","+newSrc+","+newCount+");";
12    } else if(funcName == "cudaFree"){
13      std::string newarg;
14      RewriteHostExpr(cudaCall->getArg(0), newarg);
15      newExpr = "free(" + newarg + ");";
16    } else if(funcName == "cudaMalloc"){
17      std::string newarg, newsize;
18      Expr *ptr = cudaCall->getArg(0);
19      Expr *size = cudaCall->getArg(1);
20      RewriteHostExpr(ptr, newarg);
```

```
21      RewriteHostExpr(size, newsize);
22      if(CStyleCastExpr *CSCE = dyn_cast<CStyleCastExpr>(ptr))
           {
23        if(UnaryOperator *UO=dyn_cast<UnaryOperator>(CSCE->
             getSubExpr())){
24          std::string newse;
25          RewriteHostExpr(UO->getSubExpr(), newse);
26          if (DeclRefExpr* sube=dyn_cast<DeclRefExpr>(UO->
               getSubExpr())){
27            newExpr = sube->getNameInfo().getAsString() + " =
                 (" + sube->getType().getAsString() + ") malloc(
                 " + newsize + ");";
28          }
29        }
30      } else {
31        if(UnaryOperator *UO = dyn_cast<UnaryOperator>(ptr)){
32          std::string newse;
33          RewriteHostExpr(UO->getSubExpr(), newse);
34          if (DeclRefExpr* sube=dyn_cast<DeclRefExpr>(UO->
               getSubExpr())){
35            newExpr = sube->getNameInfo().getAsString() + " =
                 (" + sube->getType().getAsString() + ") malloc(
                 " + newsize + ");";
36          }
37        }
38      }
39    }
40    return true;
41  }
```

Listing 5.16: The `RewriteCUDACall` method.

Listing 5.16 shows the `RewriteCUDACall` method. This method handles the CUDA APIs concerning the memory management, as described at the end of Section 4.3 of Chapter 4. If a `cudaMemcpy` is found (Line 3), its arguments are taken (Lines 4–6) and inserted in a `memcpy` call (Line 11). If a `cudaFree` is found (Line 12), its argument is inserted in a `free` call (Line 15). If a `cudaMalloc` is found (Line 16), the first argument `ptr` representing the pointer to the memory to be allocated is further analyzed. In order to find any special case of explicit cast in cudaMalloc (i.e. cudaMalloc((void**) var, size)) a `CStyleCastExpr` node is searched (Line 22) and eventually ignored. Therefore, a `malloc` call is created (Lines 27 and 35).

Summary

In this chapter we provided a detailed description of the implementation of the Augmenter and the Translator tools. In particular, in the first section are highlighted the code parts related to the creation of a Clang plugin. The second section shows the implementation of the Augmenter component, exposing all the methods of the `dimensionality_augmenter` class. Similarly, in the third section is shown the implementation of the Translator component.

Chapter 6

Benchmarks

This chapter evaluates the software presented in this thesis work.

The first section describes the applications used to test the software.

The second section assesses functional portability, calculating the manual effort needed to run the output code.

The third section assesses the performance portability, showing the results achieved when executing the translated applications on a set of different architectures.

6.1 Test applications

In order to evaluate the software developed in this work, four test applications were used: Stencil, Vector Addition, SRAD and BFS. The first two are synthetic applications used as a preliminary test. The last two applications are taken from the the Rodinia Benchmark Suite [9, 10]. Each application was only modified to add code to measure the run time of the program.

In the following, the four applications are described in more detail.

Stencil1D

The first application is a one-dimensional stencil example, whose kernel is similar to the one already reported in Listing 2.5 of Chapter 2. Given an input array of integer elements, it calculates an output array of the same length, whose elements contains the sum of input elements within a given radius.

vectorAdd

The vector addition application example is taken from the official NVIDIA CUDA SDK [7]. Is a very simple application that generates two random vectors of floating point elements in host memory and copies them over to the GPU's global memory. The kernel performs the addition and stores them in a third vector allocated in global memory. The resulting vector is then copied back to host memory.

SRAD

The SRAD (Speckle Reducing Anisotropic Diffusion) application example is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations (PDEs). It is used to remove locally correlated noise, known as speckles, without destroying important image features.

SRAD consists of several pieces of work: image extraction, continuous iterations over the image (preparation, reduction, statistics, two computations) and image compression. The sequential dependency between all of these stages requires synchronization after each stage (because each stage operates on the entire image). Each stage is a separate kernel (due to synchronization requirements) that operates on data already present in GPU memory. The code features efficient GPU reduction of sums. Some of

the kernels use GPU shared memory for additional improvement in performance.

BFS

The last application example is a GPU implementation of the breadth-first search (BFS) algorithm which traverses all the connected components in a graph. The code implements the algorithm given in [15], which uses level synchronization. The visit traverses the graph in levels; once a level is visited it is not visited again.

6.2 Functional portability

This section evaluates the functional portability of the software developed in this thesis work. The chosen metrics to assess functional portability is the translation coverage, that is the number of lines that need to be changed manually to run the translated application on the target system. The test applications vary in length from a couple of dozens of source lines of code to more than five hundred.

Table 6.1 shows the translation coverage of the four test applications kernels, reporting the number of total source lines of code, the number of modifications needed and the percentage of the manually modified lines with respect to the total number.

Application	**Source Lines**	**Changed**	**%**
Stencil1D	29	0	0
vectorAdd	10	0	0
SRAD	261	0	0
BFS	76	2	2.63

Table 6.1: Automatic translation coverage - Kernel only.

It is possible to observe that the software accomplishes the translation without any manual effort in all cases except from the *BFS*. This is due to the fact that the kernel was defined in a separate header file, therefore the software was not able to get the value of the block dimension from the kernel invocation.

Table 6.2 describes instead the translation coverage of the three applications in their entirety, considering both kernel and host code.

Application	Source Lines	Changed	%
Stencil1D	74	0	0
vectorAdd	212	10	4.7
SRAD	563	0	0
BFS	306	10	3.27

Table 6.2: Automatic translation coverage.

Even extending the analysis to the host code, the software behaves quite well. The *vectorAdd* application needed some manual effort in order to be able to compile. This is due to the presence of a set of error checks wrapping the CUDA API calls not handled by our software. The *SRAD* example, instead, does not present this error checking in the code (in order to achieve better performances), therefore the translation does not incur in this errors. The *BFS* example needed manual effort also in the host code because of the presence of some repeated semicolon characters in the translated version of some API calls. In each case, only a few lines of host or kernel code had to be manually modified, never exceeding the 5% of the total length of the original source code. Of the manual changes, none are particularly difficult to handle and automated support for these will be added in future work.

6.3 Performance portability

In this section, the performance portability is evaluated. In order to do that the applications were compiled and run on a set of different systems:

- GPU: Tesla C2050, 14 Multiprocessors, 448 CUDA cores clocked at 1.15 GHz;
- GPU: Tesla K40c, 15 Multiprocessors, 2880 CUDA cores clocked at 745MHz;
- CPU: AMD Opteron Processor 6176, 24 cores, clocked at 2.3GHz;
- CPU: Intel Core i7-3632QM, 4 cores, clocked at 3.2GHz;
- CPU: 2×Intel Xeon E5-2650, 8 cores, clocked at 2.0GHz.

While running each application, the run time is taken to be the time starting from the first data copy from the host to GPU device memory and

ending after the last copy back to host memory is finished. Each code was executed a total of ten times, and their run times were averaged.

Table 6.3 shows the average execution time of the applications on the architectures cited in the list above.

Application	**Tesla C2050**	**Tesla K40**	**Opteron**	**Intel i7**	**Xeon E5**
Stencil1D	6.0ms	5.3ms	25.6ms	18.0ms	39ms
vectorAdd	27.3ms	27.9ms	58ms	33.3ms	72.2ms
SRAD	17.5ms	13.6ms	146ms	210ms	126ms
BFS	32.7ms	20.5ms	148ms	87.4ms	91.1ms

Table 6.3: Average run-time.

The *Stencil1D* and the *vectorAdd* examples present a similar behaviour in terms of which CPU architecture executes faster: the Intel i7 outperforms the other two CPU architectures since it exploits better the vectorization and has a faster cache. In particular, the *vectorAdd* case for this architecture has an average execution time almost identical to the GPU one, due to the stream nature of the problem and the low reuse.

The *SRAD* example instead, benefits more from pure parallelism, and the larger number of cores of the Opteron and Xeon architectures allow to execute the translated application faster than the Intel i7.

In the *BFS* case, the Opteron architecture performs worst than the other two, due to the smaller size of the cache.

Taking as baseline the execution on the Tesla K40c GPU, the fastest GPU available in our experiments, this considerations are further summarized on Table 6.4 and Figure 6.1, which show the slowdown of the translated CPU version with respect to the CUDA version, calculated as the ratio between the CPU average execution time and the GPU average execution time.

Application	**Opteron**	**Intel i7**	**Xeon E5**
Stencil1D	4.83	3.39	7.35
vectorAdd	2.07	1.19	2.58
SRAD	10.73	15.44	9.26
BFS	7.22	4.26	4.44

Table 6.4: Slowdown w.r.t. CUDA on different CPUs.

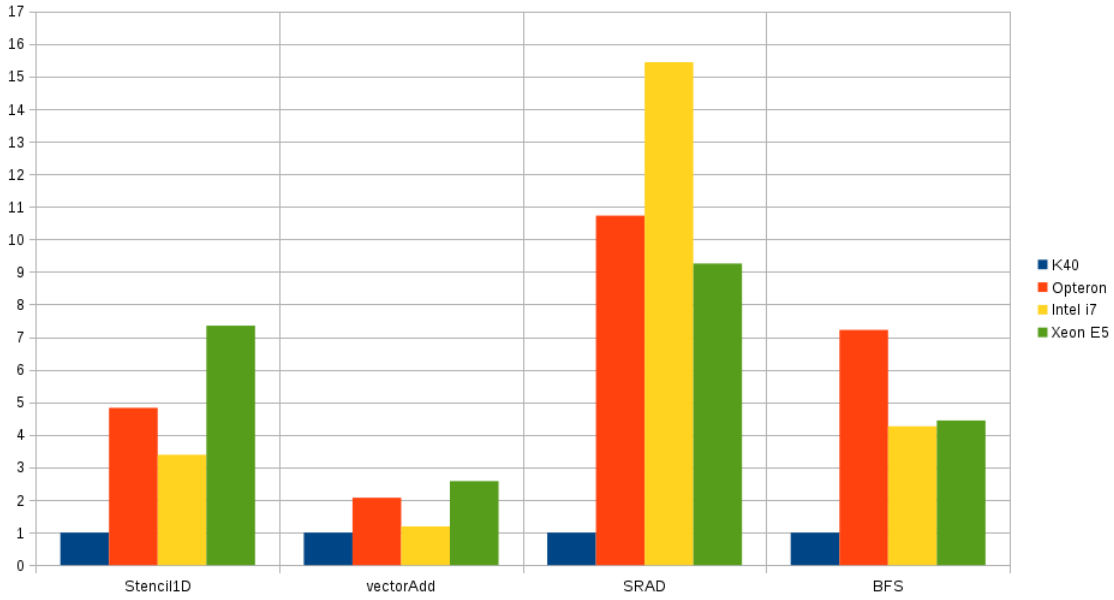

Figure 6.1: Slowdown w.r.t. CUDA on different CPUs.

The first case, *Stencil1D*, shows an higher slowdown than the *vectorAdd* case, due to the fact that it shows more locality and uses shared memory in its implementation.

The *SRAD* example shows how the translated version spends, depending on the CPU architecture used, from nine to fifteen times more than the GPU version to execute. Although the larger number of cores on the GPU is the main difference, this is due also to the fact that the CUDA version was highly optimized, since it comes from a benchmark suite, involving heavy use of shared memory.

A smaller slowdown is shown by the *BFS* example, where the CPU versions spend from four to seven times more than the GPU version. Again, the larger number of cores is the main difference, as well as the fact that this algorithm is specifically conceived for a GPU architecture.

The Rodinia Benchmark Suite provides also a native CPU version for the *SRAD* and the *BFS* examples, implemented with the OpenMP APIs [8]. The OpenMP versions of the applications does not need any `memcpy` instruction in their implementation. Therefore, any `memcpy` instruction is removed also from the translated version obtained by our software, in order to compare it with the native CPU implementation.

In Table 6.5 and Figure 6.2 is shown the slowdown of the translated versions with respect to the native versions, compiled and run on each ar-

chitecture. The slowdown is calculated as the ratio between the translated application average execution time and the OpenMP implementation average execution time.

Application	Opteron	i7	Xeon
SRAD	1.81	2.17	2.81
BFS	0.74	0.89	1.08

Table 6.5: Slowdown w.r.t. native OpenMP implementation.

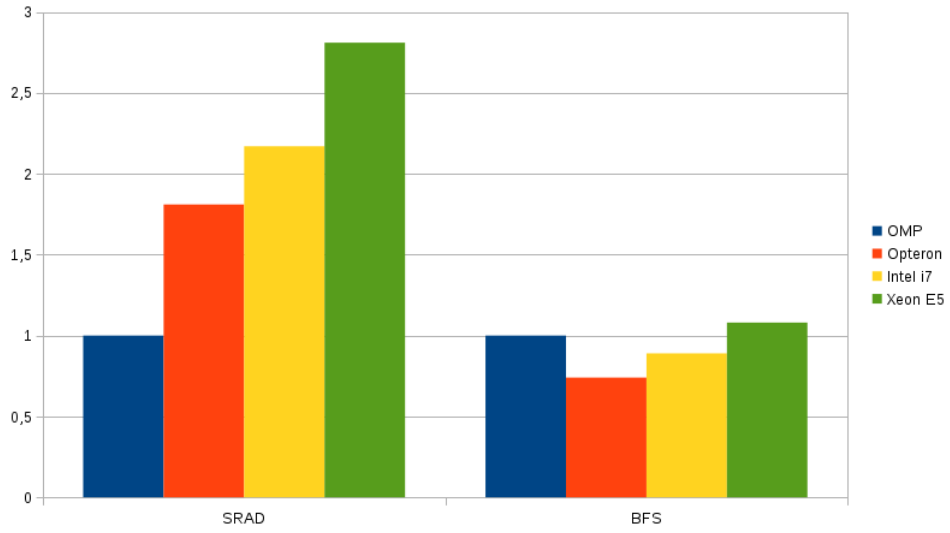

Figure 6.2: Slowdown w.r.t. native OpenMP implementation.

Again, the *SRAD* application shows how the translated version executes in about the double of the time of the native implementation. Conversely, the *BFS* case shows a speedup compared to the native OpenMP version. Analyizing in more detail this implementation, it is possible to observe that the parallelization is applied to one of the two for loops representing the steps of the BFS visit. The translated version parallelizes instead both loops.

Summary

In this chapter we evaluated the software implemented in this thesis work. The first section gave a brief description of the applications used for the benchmark. The second section presented the evaluation of the translation

coverage of the software. We shown that applications can be successfully translated with few or none manual effort. In the third section is presented the evaluation of the performances of the translated application. We also showed that a good level of performance portability is achieved, guaranteeing comparable performance taking into account the architectural differences between the targeted architectures.

Chapter 7

Conclusion

7.1 Summary

GPU-based heterogeneous computing has seen significant growth and interest in recent years, and there exist several approaches to provide a way of programming this accelerator devices. Some approaches take the form of a proprietary framework, like CUDA, providing ease of development on a single family of devices.

When developing an accelerated application, performance and programmability are the strengths of the CUDA framework that often sway the decision that leads to its use. However, portability is also a significant consideration, to provide a wider user-base for accelerated software and to reduce the development cost required to access alternative (or future) devices that may afford comparable or increased performance. Amongst this alternative devices, the large amount of hardware not necessarily having a GPU is a resource that should not be left unused. A broad range of applications have been developed in CUDA that are currently "vendor-locked" to NVIDIA platforms, with time-intensive manual translation to another programming framework left as the only way to achieve portability.

In this thesis we presented an automated way to port CUDA applications on different architectures, in particular focusing on shared-memory multi-core CPU architectures. The accomplishment of this goal is achieved operating at the CUDA source code level, analysing its abstract syntax and defining a set of transformations on it. As result of this transformations, we produced regular C++ code in output.

As a proof of concept of the correctness of the transformations presented,

a prototype source-to-source translator was implemented. By leveraging the Clang compiler framework, we were able to take advantage of its powerful source-level tools to perform the translation. As an early prototype, we focused on a useful subset of the CUDA language. We therefore evaluated this software, analysing if any further effort was needed after the translation to preserve the functionality of the original CUDA program. This first set of tests shown that in half of the cases no manual effort was required, while in the other half very few manual modifications were needed. Lastly, we evaluated the performance of the obtained applications, showing how comparable performances between the architectures were guaranteed.

7.2 Future work

The work presented in this thesis may be extended in several ways. Since the implementation was mainly oriented to present a functional prototype, the extensions presented below were not implemented only for time reasons, although the solution for these problems is conceptually already designed.

Extending CUDA support Although the preliminary results seem promising, a remarkable part of the CUDA syntax is not handled by the software implemented in this work. For example constant memory and texture memory support, as well as the mapping of the CUDA atomic kernel instructions, if handled, could bring major advantages to the translated applications, also in terms of performances.

AST mutability As said in Chapters 3 and 4, Clang's AST are immutable. It could be useful implementing a correctness check of the ASTs in order to avoid to parse again the code after the execution of operations involving the reordering or the removal of the nodes.

Selective replication In the emulating thread-local memory phase of the transformations, some local variables may have live ranges completely contained within an iterative wrapping. In order to use less memory space, an algorithm that creates arrays for local variables only when necessary should be developed. The previously cited related work MCUDA [24] presents a technique to accomplish this, called selective replication and based on live variable analysis.

Accurate declaration reordering At the moment multiple declarations of variables with the same names in different scopes are not handled. Therefore moving everything at the top of the body of the kernel, can incur in a failure of the translation process. A possible solution could be the displacement of the declarations at the beginning of the scope on which they are declared, and not at the beginning of the body.

Run time kernel configuration If the number of the threads in a block is given at run time, the translation process requires manual effort, since we cannot know in the kernel the size of the dimensionality added to the local-thread variables.

Fresh variable names A live-variable analysis should be implemented, in order to avoid problems concerning the presence of variables in the original code having the same name of the ones introduced by our software (such as `__ttid_`).

Preprocessor A preprocessing phase should be implemented, since complex `#include` directives involving kernel functions are not handled. Again, Clang can be exploited since it provides a set of libraries to handle the preprocessing phase of the compilation.

Customizing CPU parallelization Additional techniques to parallelize the execution of the translated applications could be easily implemented, adding the possibility to chose on which framework to rely on. In the translation phase, for example, the user could choose to insert OpenMP compilation pragmas or rely on a Standard Library C++ thread pool.

Kernel invocation tuning Kernels having a semantics not dependent from the value of the dimension of the grid, could be invoked with different parameters, in order to schedule more or less threads with respect to the original number of blocks and bring performance advantages to the translated application.

`memcpy` removal CUDA programming model necessitates that the data used by a kernel has to be copied from host to device memory. However, an analysis of the usage of this data after the kernel execution could guarantee

that there is no need to make a copy between two memory areas in the translated version, thus bringing considerable performance advantages.

Bibliography

[1] Clang - Features and Goals. http://clang.llvm.org/features.html.

[2] Clang documentation. http://clang.llvm.org/.

[3] Clang user manual. http://clang.llvm.org/docs/UsersManual.html.

[4] Intel® Threading Building Blocks (Intel® TBB) 4.4. https://www.threadingbuildingblocks.org/.

[5] Introduction to the Clang AST. http://clang.llvm.org/docs/IntroductionToTheClangAST.html.

[6] The LLVM Compiler Infrastructure. http://www.llvm.org/.

[7] NVIDIA. CUDA Toolkit & SDK. http://developer.nvidia.com/cuda-toolkit-sdk.

[8] The OpenMP® API specification for parallel programming. http://openmp.org/wp/.

[9] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.

[10] Shuai Che, J.W. Sheaffer, M. Boyer, L.G. Szafaryn, Liang Wang, and K. Skadron. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–11, Dec 2010.

[11] Marco Danelutto. *Distributed Systems: Paradigms and Models.*

[12] Gregory Diamos. The design and implementation Ocelot's dynamic binary translator from PTX to multi-core x86. Technical report, 2009.

[13] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 353–364, New York, NY, USA, 2010. ACM.

[14] Christopher Guntli. Architecture of clang. *Analyze an open source compiler based on LLVM*, 2011.

[15] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag.

[16] M.J. Harvey and G. De Fabritiis. Swan: A tool for porting CUDA programs to OpenCL. *Computer Physics Communications*, 182(4):1093–1099, 2011.

[17] C. Kessler, U. Dastgeer, S. Thibault, R. Namyst, A. Richards, U. Dolinsky, S. Benkner, J.L. Traff, and S. Pllana. Programmability and performance portability aspects of heterogeneous multi-/manycore systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 1403–1408, March 2012.

[18] D.B. Kirk and W.W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach.* Applications of GPU Computing Series. Elsevier Science, 2010.

[19] Sang-Ik Lee, Troy A. Johnson, and Rudolf Eigenmann. Cetus–an extensible compiler infrastructure for source-to-source transformation. In *Languages and Compilers for Parallel Computing*, pages 539–553. Springer, 2004.

[20] Gabriel Martinez, Mark Gardner, and Wu-chun Feng. CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-Core Architectures.

In *17th IEEE International Conference on Parallel and Distributed Systems*, Tainan, Taiwan, December 2011.

[21] Deepthi Nandakumar. Automatic translation of CUDA to OpenCL and comparison of performance optimizations on GPUs. Master's thesis, University of Illinois, 2011.

[22] NVIDIA® . *CUDA C Programming Guide*, 7.5 edition, 2015.

[23] Paul Sathre, Mark Gardner, and Wu-chun Feng. Lost in Translation: Challenges in Automating CUDA-to-OpenCL Translation. In *5th International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, Pittsburgh, PA, September 2012.

[24] John A. Stratton, Sam S. Stone, and Wen-mei Hwu. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In *Languages and Compilers for Parallel Computing*, pages 16–30. Springer, 2008.

[25] Alexandros Tzannes. Enhancing productivity and performance portability of general-purpose parallel programming. 2012.