CODE GENERATION COMPILER FOR THE OPENMP 4.0 ACCELERATOR MODEL ONTO OMPSS

Guray Ozen

FACULTAT D’INFORM ÀTICA DE BARCELONA
Master in Innovation and Research in Informatics
High Performance Computing Program
Universitat Politecnica de Catalunya (UPC-BarcelonaTECH)

Master of Sciences
Advisors:
Prof Jesus Labarta
Prof Eduard Ayguade
Barcelona, UPC, 2014
"Premature optimization is the root of all evil."
— Donald Knuth

Thanks my mother...
Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisors Prof. Jesus Labarta and Prof. Eduard Ayguade for the continuous support of my Master Thesis study and research, for their patience, motivation, enthusiasm, and immense knowledge. Their guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Master Thesis study.

I thank also Programming Models group, especially Compiler team in Barcelona Supercomputer center.

Last but not the least, I would like to thank my family: for giving birth to me at the first place and supporting me spiritually throughout my life.

*Barcelona, 01 June 2014*  

Guray Ozen
Abstract

The aim of OpenMP which is a well known shared memory programming API, is using shared memory multiprocessor programming with pragma directives easily. Up till now, its interface consisted of task and iteration level parallelism for general purpose CPU. However OpenMP includes in its latest 4.0 specification the accelerator model.

OmpSs is an OpenMP extended parallel programming model developed at the Barcelona Supercomputing Center and it have already supported accelerators without code generation. The main objective of OmpSs is to orchestrate different kind of tasks. The design of OmpSs is highly biassed to delegate most of the decisions to the runtime system, which based on the task graph built at runtime (depend clauses) is able to schedule tasks in a data flow way to the available processors and accelerator devices and orchestrate data transfers and reuse among multiple address spaces.

In this thesis i present a MACC compiler which is partial implementation of this specification in the OmpSs programming model with the aim of identifying which should be the roles of the programmer, the compiler and the runtime system in order to facilitate the asynchronous execution of tasks in architectures with multiple accelerator devices and processors. For this reason implementation of thesis is partial, just considering from 4.0 those directives that enable the compiler the generation of the so called “kernels” to be executed on the target device. Several extensions to the current specification are also presented, such as the specification of tasks in “native” CUDA and OpenCL or how to specify the device and data privatization in the target construct. Finally, the paper also discusses some challenges found in code generation and a preliminary performance evaluation with some kernel applications.

Before starting issues about GPU code generation, PURE GPU programming models are discussed and are showed their pears and pitfalls. Besides all these, challenges of GPU programming are discussed such as memory usage, multi gpu and streaming. Based upon these experience, some methods are developed and they are applied onto MACC compiler. Basically MACC’s extension from OpenMP 4.0 is based on these methods.

Keywords: Compiler, Accelerators, OpenMP, OmpSs, GPGPU, Code generation, Source-to-source compiler, SYCL, CUDA, OpenCL
# Contents

Acknowledgements i  
Abstract iii  
List of figures vii  

## 1 Introduction 1  
1.1 Motivation ................................................. 2  
1.2 Thesis Objectives .......................................... 3  
1.3 Thesis Organization ........................................ 3  

## 2 GPU As An Parallel Computation Target 5  
2.1 CUDA Hardware Model ...................................... 6  
2.1.1 Execution Units ........................................ 6  
2.1.2 Memory Hierarchy ...................................... 7  
2.2 CUDA Programming Model ................................... 8  
2.2.1 Streaming & Concurrency ............................... 9  
2.2.2 Runtime & Environment ................................ 10  
2.3 Accelerator Programming with Intermediate Languages ................................. 12  

## 3 Related Works & State of Arts 15  
3.1 Source–to–Source Transformations .......................... 16  
3.2 Directive Based Standards OpenMP 4.0 & OpenACC ............... 20  
3.3 OmpSs Heterogeneous Hardware Support ..................... 23  
3.3.1 Nanos++ Runtime ..................................... 23  
3.3.2 GPU support for OmpSs ................................. 25  
3.4 C++11 Extended Code Generation ............................ 28  

## 4 MACC: Mercurium ACCelerator Compiler 31  
4.1 Basis on Mercurium & MACC Compiler ........................ 31  
4.1.1 Parsing & Mercurium IR ................................ 32  
4.1.2 Mercurium Compiler Work Flow .......................... 33  
4.1.3 MACC Phases Pipeline .................................. 34  
4.1.4 MACC Configuration ................................... 36  
4.2 ACCelerator Code Generation Phase .......................... 37


## Contents

4.2.1 MACC Accelerator Directives ........................................ 37  
4.2.2 Kernel Configuration .................................................. 38  
4.2.3 Loop Scheduling & Thread Mapping ................................. 39  
4.2.4 Device Local Memory .................................................. 41  
4.2.5 New data sharing attributes for distribute directive ............ 42  
4.2.6 Reductions and Atomic Operations ................................. 43  

5 Evaluations 45  
5.1 SAXPY Application ....................................................... 46  
5.2 Jacobi Application ....................................................... 47  
5.3 DG-Kernel Application .................................................. 49  
5.4 CG Application from NASA Parallel Benchmark Suite .......... 51  
5.5 EP Application from NASA Parallel Benchmark Suite .......... 53  

6 Conclusion 57  
6.1 Summary ................................................................. 57  
6.2 Conclusion ............................................................... 58  

Bibliography 62
# List of Figures

2.1 GPU vs CPU ........................................... 5  
2.2 Common scenarios of thread divergency ....................... 7  
2.3 CUDA Memory Model .................................. 9  
2.4 CUDA Thread Hierarchy ................................ 10  
2.5 Altogether CUDA application ................................ 11  
2.6 CUDA Execution Model .................................. 12  

3.1 Matrix Multiplication by using HiCUDA ....................... 17  
3.2 Evaluation CUDA from plain OpenMP code by using CETUS compiler .... 19  
3.3 Simple code in OpenMP 4.0 using multiple accelerator devices ......... 21  
3.4 Another simple code in OpenMP 4.0 and OpenACC to show data regions .... 22  
3.5 Use of private variables and team/thread level reductions in Jacobi .......... 23  
3.6 NANOS++ Runtime Structure ................................ 24  
3.7 NANOS++ Behaviour ...................................... 25  
3.8 Using multiple accelerators in OmpSs .......................... 25  
3.9 Heterogeneous task example with OmpSs ........................ 27  
3.10 Vector Addition Example by SYCL ................................ 29  
3.11 C++ Parallel Algorithms Library Progress .......................... 29  

4.1 Python script which is involved by Mercurium to generate AST Nodes ........ 33  
4.2 Mercurium Compiler Phases Pipeline ................................ 34  
4.3 MACC Compiler Phases Pipeline ............................... 35  
4.4 MACC configuration file ...................................... 37  
4.5 Comparing Pragma Directives: MACC vs OpenMP 4.0 .................... 38  
4.6 MACC - Deciding dimensionality of the kernels ...................... 39  
4.7 DG-Kernel Application in MACC way ................................ 40  
4.8 Generated MACC Kerneler to configure kernel from Figure 4.7 ............ 41  
4.9 Using Atomic (NAS - IS) ...................................... 43  
4.10 Generated MACC CUDA Kernel from Figure 4.7 ........................ 44  

5.1 SAXPY Application with MACC directives .......................... 46  
5.2 Annotated codes for Jacobi application ............................ 48  
5.3 Performance evaluation of Jacobi application ........................ 48  
5.4 Example to explain MACC implementation of shared memory - DG Kernel .... 49
List of Figures

5.5  Performance evaluation for DG kernel ............................................. 50
5.6  CUDA Kernel evaluation for DG kernel ........................................... 51
5.7  Annotated codes for NAS CG benchmark ......................................... 52
5.8  Profiling CG - NAS by Paraver ....................................................... 53
5.9  CG - NAS Application ................................................................. 54
5.10 Annotated codes for NAS EP benchmark ......................................... 55
5.11 Performance evaluation for NAS EP ............................................... 56

6.1  Comparing Discussed Compilers and MACC ................................. 57
1 Introduction

Story of the using powerful core is still a most common way for high performance computing. As is known, CPUs has quite large architecture which consists out-of-order, deeper pipeline, many functional units etc. When considered from point of software, many programming languages and environment are designed for them since they are used for many years. Computer scientist get used to sequential programming and their environment. However as the time went by, people’s essential requirements was changed such as cache efficiency, consuming much electricity and etc. Then multicore CPUs appeared. In addition to this, considering parallelism became necessary at the side of programmers. At that time, many programming languages and interfaces appeared to facilitate to use multicores. OpenMP [12] is one of them.

While multicores are being developed, GPU companies allowed to own card to be programmed for general purpose. This approach is called as GPGPU. Due to the GPU’s higher peak performance and performance per Watt ratio when compared to homogeneous architectures based on multicores, the use of accelerators has been gaining popularity in the last years. They offered also powerful environment and well designed guides. The approach of GPGPU is that CPU (Host) programs can manage massively parallelized general purpose GPU program. Some alternatives have been proposed to address the programmability of these accelerator–based systems. CUDA [8] and OpenCL [10] provide low–level APIs to allow offload computation offloading to accelerators, the management of their memory hierarchy and the data transfers between address spaces.

However, the heterogeneity they introduce (in terms of computing devices and address spaces) makes programming a difficult task. In addition there is another important thing about them
that is GPU’s architecture is totally different than accustomed CPUs. Moreover their architecture is changing often pretty much by the reason of new architecture approach. Moreover GPU's programming model based on SIMT (Single Instruction, Multiple Threads). It is even quite different from usual parallel programming model of CPU such as task or iteration level parallelism. The other side of the coin is not bright enough as it seem. Because GPGPU programmers should think hardware processing logic and manage memory operations. Otherwise they will pay the penalty and their accelerator program probably works even worse than sequential CPU program. This task sometimes is hard even for expert GPU programmers. For instance, changing algorithm may be needed but probably there is no existed algorithm is designed for GPUs. Besides, programmers should even take into account GPU’s many different memory location to leverage advantages of close memory. They almost don't have cache management, coherency, branch prediction. For these reasons, GPU programming may take a long time because of thinking accelerator hardware.

1.1 Motivation

Due to the programmability difficulties of accelerators, many companies announced special libraries which are compatible with them. In this way programmers don't care hardware complexity of them, and no need to have idea about accelerator programming. Only using these libraries as a normal library for CPU is enough. Moreover, these libraries can be ported easily to existed application since some of libraries has really minor changing with CPU version.

Obviously using accelerator by libraries is limited and the solution of libraries are limited also with a particular area. Therefore pragmatic approaches such as OpenACC [11], have appeared with the aim of providing a higher–level directive–based to program accelerator devices. OpenMP [12] also includes in its latest 4.0 specification the accelerator model with the same objective. These solutions based on directives still rely on the programmer for the specification of data regions, transfers between address spaces and for the specification of the computation to be offloaded in the devices; these solutions also put a lot of pressure on the compiler–side who has the responsibility of generating efficient code based on the information provided by the programmer.

On the other hand, the OmpSs [19] proposal has been evolving during the last decade to lower the programmability wall raised by multi–/many–cores, demonstrating a task–based data flow approach in which offloading tasks to different number and kinds of devices, as well as managing the coherence of data in multiple address spaces, is delegated to the runtime
system. Multiple implementations were investigated for the IBM Cell (CellSs [20]), NVIDIA
GPU (GPUSs [21]) and homogeneous multicores (SMPSs [18]) before arriving to the current
unified OmpSs specification and implementation. Initially OmpSs relied on the use of CUDA
and OpenCL to specify the computational kernels.

1.2 Thesis Objectives

The main intention of this thesis is to provide code generation for accelerator. Therefore a
new research compiler which name is MACC was developed based on OmpSs platform and
was adopted OpenMP 4.0 functionalities that are necessary to specify computational kernels
in a more productive way. Since OmpSs has already interoperated accelerator programming,
implementation of MACC relies on OmpSs suite. Basically this thesis presents the latest imple-
mentation of OmpSs which includes partial support for the accelerator model in OpenMP 4.0
specification.

Briefly, MACC compiler allows to programmers to express offloading code scopes and data.
It exploits OmpSs tasks orchestration system and by this means programmers can indicate
different kind of task such as OpenMP 4.0, CUDA[8], OpenCL[10] at the same program.

In this thesis also consists briefly GPU hardware and GPU programming model and analyses
the roles of the programmer, the compiler and the runtime.

1.3 Thesis Organization

The rest of thesis is organized as follows. Chapter 2 explains CUDA architecture and CUDA
programming model. Chapter 3 briefly describes an current state of arts, existed compilers and
future approach of source to source compiler in order to generate accelerator program. Besides
it explains how to works OmpSs. Chapter 4 firstly mentions Mercuirum compiler[2] since it
based on the MACC compiler. Afterwards it explains fundamentals of MACC compiler and
how does MACC generate CUDA codes. Chapter 5 presents and characterizes the performance
evaluation results and chapter 6 concludes the paper.
GPU is a multi-threaded, many-core, massively parallel processor. It is designed principally for computing graphic operation. Obviously these different can be seen at Figure 2.1. In order to fully take advantage of this incomparable strength of GPU, recently, graphic card company NVidia is announced its new general purposed computing architecture "CUDA" which enables dramatically increases in computing performance the computing power of GPU. In addition, Nvidia published CUDA SDK, programming interfaces, super powerful environments such as profiler, debugger and many libraries.

In this chapter CUDA hardware and programming model is mentioned in order to do initial declaration for thesis. First section is going to explain CUDA hardware. Following sections explains its programming model and runtime briefly.
Chapter 2. GPU As An Parallel Computation Target

2.1 CUDA Hardware Model

GPU hardware is quite different than usual CPU hardware. Principally they were not designed for general purpose application. GPUs have many small cores and those can be run in parallel way. Because in order to process graphics big cores are not necessary.

Besides communication bus can be PCI-E, if they are not part of CPU. No wonder there are other architecture way though PCI-E is main communication way in HPC area. However in every case GPUs have own memory system and it need to be configured and managed by programmer.

In brief GPU has two number of key blocks; Execution Units which are Streaming Multiprocessor (Parallel Execution Unit) and Streaming Processors (Cores) and Memory.

2.1.1 Execution Units

One of main components is streaming multiprocessors (SM) where is performed actual computations. Each SM consist SP/DP Cuda Cores, Load/Store unit, Warp Scheduler, Special Function Unit, several caches and one interface to access upper level memory. They supports instruction level parallelism but they don't have branch predictor or speculative execution. Consisted cores is quite small. It has floating and integer unit, logic unit, move/compare unit and small branch unit.

GPU is responsible for scheduling the blocks/threads, each block/thread can be scheduled on any available multiprocessor and core, concurrently or sequentially. Therefore execution order of blocks and threads are undefined. In addition, the execution of threads block works completely independently of each other. Since this feature brings to computation to be transparent scalable, since GPUs with different number of multiprocessors can automatically assign each block to any available multiprocessor without worrying about the entanglement between different blocks.

CUDA threads are scheduled in a groups of warps and they are run within a warp as SIMT (Single-Instruction Multiple Thread) way. This work is managed by streaming multiprocessors. Warp is one of the taxonomy in CUDA. In the kepler architecture, 32 threads is bunched by
2.1. CUDA Hardware Model

```c
1 tid = blockDim.x * blockIdx.x + threadIdx.x;
2 if(array[tid] > 0)
3  \<...computation...>
4 else
5  \<...computation...>

endId = array[blockDim.x * blockIdx.x + threadIdx.x];
2 for(int i = 0; i < endId ; ++i)
3  \<...computation...>
```

Figure 2.2 – Common scenarios of thread divergency

one warp. Bunch-able number of threads depends on CUDA architecture. Bunched threads can be run at the same pipeline like SIMD fashion. Therefore all threads in the warp executes same instruction at the same time. In here there is a big problem for GPU programmers. However devil is in the details. Because if one thread in the warp encounters branch, its future instructions may going to be different than rest of threads. If so, there are two different instruction way in the warp. Since warp’s instructions are run at the same time, one of them is stalled. A stalled warp is ineligible to be selected by the warp scheduler. On Fermi it is useful to have at least 2 eligible warps per cycle so that the warp scheduler can issue an instruction. This problem is called thread divergency. It may cause %50 performance loss. In order to clarify this problem, Figure 2.2 shows possible cases. In general using irregular accessing to array as branch condition may cause divergency problem.

On the other hand, NVIDIA GPU has coalesced memory access. When all threads in a warp execute a load instruction, the hardware detects whether the threads access consecutive memory locations. The most favorable global memory access is achieved when the same instruction for all threads in a warp accesses global memory locations. In this favorable case, the hardware coalesces all memory accesses into a consolidated access to consecutive DRAM locations by the special hardware in CUDA GPUs, global memory loads and stores issued by threads of a warp into as few transactions as possible to minimize DRAM bandwidth (on older hardware of compute capability less than 2.0, transactions are coalesced within half warps of 16 threads rather than whole warps).

2.1.2 Memory Hierarchy

In CUDA architecture, host and device have separate memory spaces. All threads in the device are only running on the device memory, the programmer is responsible for allocating the necessary device spaces, transferring the related data between host and device memory, and cleaning up all of the unnecessary device memory. CUDA run time system provides
programmers with APIs to perform all of these activities. Figure 2.3 is illustrated as a device.

1. **Registers**: Registers are the fastest memory, accessible without any latency on each clock cycle, just as on a regular CPU. A thread's registers cannot be shared with other threads.

2. **Shared Memory**: It is comparable to L1 cache memory on a regular CPU. It resides close to the multiprocessor, and has very short access times. Shared memory is shared among all the threads of a given block.

3. **Global memory**: It resides on the device, but off chip from the multiprocessors, so that access times to global memory can be 100 times greater than to shared memory. All threads in the kernel have access to all data in global memory.

4. **Local Memory**: Thread-specific memory stored where global memory is stored. Variables are stored in a thread's local memory if the compiler decides that there are not enough registers to hold the thread's data. This memory is slow, even though it's called "local".

5. **Constant Memory**: 64k of Constant memory resides off-chip from the multiprocessors, and is read-only. The host code writes to the device's constant memory before launching the kernel, and the kernel may then read this memory. Constant memory access is cached each multiprocessor can cache up to 8k of constant memory, so that subsequent reads from constant memory can be very fast. All threads have access to constant memory.

6. **Texture Memory**: It likes constant memory, texture memory is cached on chip, so in some situations it will provide higher effective bandwidth by reducing memory requests to off-chip DRAM. Specifically, texture caches are designed for graphics applications where memory access patterns exhibit a great deal of spatial locality.

### 2.2 CUDA Programming Model

CUDA provides programming model that is ANSI C, extend several keywords and constructs. Programmers writes a single source program that contains both host (CPU) and device (GPU) code. These two parts automatically separated compiled by the CUDA compiler tool chain. CUDA program which is callable is called as kernel. As it is mentioned before CUDA architecture is designed to run instructions in parallel way. Therefore it can be run as massively parallel.
2.2. CUDA Programming Model

Figure 2.3 – CUDA Memory Model

Figure 2.4 is illustrated CUDA thread hierarchy. As it is seen, grids and blocks can be configured multi dimensional. This approach is used to leverage data locality by coalesced access pattern. However these dimensions need to be configured manually by programmers. There is no auto decision mechanism in order to find optimum number of thread size. Figure 2.5 shows sample CUDA program with host program. Line 47-48 finds number of block and grid. However this calculation may be change each CUDA program. On the other hand, CUDA programming model programmers need to manage memory allocation and flow. During the execution, each thread may access data from different types of device memory Figure 2.3. First of all, all threads in all blocks can read and write the global memory which is allocated and released by calling the `cudaMalloc` and `cudaFree` functions respectively. In the simple cuda example Figure 2.5 is handled at Line 40–41, 44, 45 and 54. Kernel is invoked at line 50 with parameters which are couple pointers which holds address at GPU and kernel dimensions which are calculated at line 47-48. When examined CUDA kernel starts with Line 1, using shared memory rules can be seen. While line 4-5 allocates area from shared memory, line 12–19 fills these areas from global memory.

2.2.1 Streaming & Concurrency

CUDA programming model is not totally data parallel programming model. In fact it supports concurrency by task level parallelism. When examined parallelism context, a task can be many different things. For instance, an CUDA application can be executing two tasks: some computation with one thread while downloading something with another thread. These tasks proceed in parallel, despite having nothing in common. Although the task parallelism on
GPUs is not currently as flexible as a general-purpose processor’s, it still provides opportunities for us as programmers to extract even more speed from our GPU-based implementations. These facility can be used if GPU’s computation capability is higher than 2.0.

### 2.2.2 Runtime & Environment

In favor of programmability, the latest releases of the Nvidia CUDA architecture improved programming productivity by moving some of the burden to the CUDA runtime, including Unified Virtual Addressing (CUDA 4) to provide a single virtual memory address space for all memory in the system (enabling pointers to be accessed from GPU) no matter where in the system they reside) and Unified Memory (CUDA 6) to automatically migrate data at the level of individual pages between host and devices, freeing programmers from the need of allocating and copying device memory. Although it may be seen as a need for beginners, it makes it possible to share complex data structures and eliminate the need to handle "deep copies" in the presence of pointed data inside structured data. Carefully tuned CUDA codes may still use streams and asynchronous transfers to efficiently overlap computation with data movement when the CUDA runtime is unable to do it appropriately due to lack of lookahead. Figure 2.6 shows both of approach.
```c
__global__ void matrixMultiply( float* A, float* B, float* C, int numARows, int NCA, int numBRows, int NCB, int numCRows, int numCColumns)
{
    int bx = blockIdx.x, by = blockIdx.y, tx = threadIdx.x, ty = threadIdx.y,
    Row = by * TILE + ty, Col = bx * TILE + tx;

    float Pvalue = 0;

    for (int m = 0; m < (NCA - 1)/TILE + 1; ++m) {
        if (Row < numARows && m*TILE+ tx < NCA)
            ds_M[ty][tx] = A[Row *NCA+m*TILE+tx];
        else
            ds_M[ty][tx] = 0;

        if (Col < NCB && m*TILE+ty < numBRows)
            ds_N[ty][tx] = B[(m*TILE+ty)*NCB+Col];
        else
            ds_N[ty][tx] = 0;

        __syncthreads();
        for (int k = 0; k < TILE; ++k)
            Pvalue += ds_M[ty][k] * ds_N[k][tx];
        __syncthreads();
    }

    if (Row < numCRows && Col < numCColumns)
        C[Row *numCColumns+Col] = Pvalue;
}

int main(int argc, char* argv[])
{
    int NRA= 3000, NCA =4000, NCB =5000;

    float* d_a, * d_b, * d_c;
    float* h_a = (float*) malloc(NRA*NCA*sizeof(float));
    float* h_b = (float*) malloc(NCA*NCB*sizeof(float));
    float* h_c = (float*) malloc(NRA*NCB*sizeof(float));

    init(a, b);

    cudaMalloc(&d_a, NRA*NCA*sizeof(float));
    cudaMemcpy(d_a, h_a, NRA*NCA*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, NCA*NCB*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_c, h_c, NRA*NCB*sizeof(float), cudaMemcpyHostToDevice);

    dim3 dimGrid((NCB-1)/TILE+1, (NRA-1)/TILE+1, 1);
    dim3 dimBlock(TILE, TILE, 1);
    dimGrid<<dimBlock>>>(d_a, d_b, d_c, NRA,NCA,NCB, NRA, NCB);
    cudaMemcpy(h_c, d_c, NRA*NCB*sizeof(float), cudaMemcpyDeviceToHost);
    check_result(h_c);
}
```

Figure 2.5 – Altogether CUDA application
Chapter 2. GPU As An Parallel Computation Target

2.3 Accelerator Programming with Intermediate Languages

Another novel approach for developing accelerator program is using intermediate language. Currently there are two approaches such as PTX(Parallel Thread eXecution)[13] and SPIR(Standard Portable Intermediate Representation) [17]. Main idea of both is providing instruction set for accelerators.

PTX belongs to NVidia and it can be used only NVidia accelerators. The goals for PTX include the following:

- Provide a stable ISA that spans multiple GPU generations.
- Achieve performance in compiled applications comparable to native GPU performance.
- Provide a machine-independent ISA for C/C++ and other compilers to target.
- Provide a code distribution ISA for application and middleware developers.
- Provide a common source-level ISA for optimizing code generators and translators, which map PTX to specific target machines.
- Facilitate hand-coding of libraries, performance kernels, and architecture tests.
- Provide a scalable programming model that spans GPU sizes from a single unit to many parallel units.
2.3. Accelerator Programming with Intermediate Languages

On the other hand, the SPIR belongs to Khronos group, therefore it is able to use for every vendor. In short SPIR:

- Vendor neutral
- Is not C source code
- Supports all core features and KHR extensions for version 1.2 of OpenCL C
- Is designed to support vendor extensions.
- Is compact.
- Is designed to be efficiently loaded by an OpenCL implementation.
3 Related Works & State of Arts

It should not be forgotten that, Accelerator–specific programming initially puts all responsibility to the programmer, who should take case of transforming computational intensive pieces of code into kernels to be executed on the accelerator devices and write the host code to orchestrate data allocations, data transfers and kernel invocations with the appropriate allocation of GPU resources. Nvidia CUDA [8] and OpenCL [10] are the two APIs commonly used today. Moreover there are many details to consider in the pure accelerator programming. Due to the those details to consider, obtaining speed-up may become hard and optimizing may very harder even for CUDA experts.

The another thing is portability. According to Nvidia, CUDA codes are portable on condition that move to accelerators should remain NVidia. If you are NVIDIA user, it may good deal. However even if these applications are portable among NVidia graphic cards, nobody give guarantee about the codes remain optimized. Besides some code parts must be changed such as used shared memory size, number of threads and blocks due to the various CUDA architecture (Basically it depends on Computation Capability of CUDA device). Therefore this thesis strictly defends at least some parts of codes should be generated by compiler for accelerators.

With the aim of providing a smoother and portable path to program accelerator-based architectures, existed researches, OpenACC and OpenMP 4.0 provide a directive-centric programming interface. Before they appeared, some research projects based on directive programming are presented. Directive based approach is selected since it is familiar for programmer. It allow the programmer to specify the code regions to offload to accelerators, how to map loops inside
those regions onto the resources available in them, the data mapping in their memory and data copying. With this, the compiler and runtime (if exist) system takes care of accelerator startup and shutdown, code offloading and implementing data allocations/transfers between the host and accelerator, as specified by the programmer. If the accelerator can access the host memory directly, the implementation may avoid this data allocation/movement and simply use the host memory.

This directive–based approach imposes a high responsibility to the compiler who needs to be able to generate optimized device–specific kernels, considering architectural aspects such as the memory hierarchy or the amount of resources available. This set of directives free the programmer from the need to write accelerator-specific code for the target device (e.g. CUDA or OpenCL kernels). I think this is important in terms of programming productivity, but I also believe that the directive–based approach should allow a migration path for existing kernels or the reuse of device-specific OpenCL or CUDA kernels already optimized by experienced programmers.

On the other hand, a new programming style of GPUs is presented. In this techniques is exploiting by C++11 lambda facility and generation intermediate assembly codes (SPIR or PTX). However NVidia version is still draft.

In sum, this section is going to discuss current three approach about accelerator programming. First part reviews ideas of exist research papers and their implementation model. Second part shows directive based standards such as OpenMP and OpenACC and OmpSs programming model approach. and last part mentions GPU programming with C++.

3.1 Source–to–Source Transformations

HiCUDA

It can be said that HiCUDA[22] is almost first translation approach in respect of directive in this field. It was developed at University of Toronto. Shortly HiCUDA is a code generator with own pragma directives. Its pragmas start with "#pragma hicuda". Basically every kernel line bases on almost one pragma directive. Therefore know-how about GPU programming is expected from programmers. Therefore there are many type of directives. It expects from programmers to do many details like Pure GPU programming such as memory allocation and
3.1. Source-to-Source Transformations

**HiCUDA Program**

```c
int main(int argc, char *argv[])
{
    float A[64][128];
    float B[128][32];
    float C[64][32];
    init(A, B);
    #pragma hicuda global alloc A[*][*] copyin
    #pragma hicuda global alloc B[*][*] copyin
    #pragma hicuda global alloc C[*][*]
    #pragma hicuda kernel matrixmul tblock (4, 2) thread (16,16)
    #pragma hicuda loop_partition over_tblock over_thread
    for(int i=0; i<64;++i)
        for(int j=0; j<32;++j)
            float sum= 0.0;
            for(int kk =0; kk <128; kk+=32)
                #pragma hicuda shared alloc A[i][kk:kk+31] copyin
                #pragma hicuda shared alloc B[kk:kk+32][j] copyin
                #pragma hicuda barrier
                for(int k=0; k<32; ++k)
                    sum += A[i][kk+k] * B[kk+k][j];
            #pragma hicuda barrier
                for(int k=0; k<32; ++k)
                    sum += A[i][kk+k] * B[kk+k][j];
            #pragma hicuda shared remove A B
            #pragma hicuda kernel_end
    C[i][j]=sum;
}
```

**Generated CUDA Code**

```c
__global__ void matrixMul (float *A, float *B, float *C, int wA, int wB)
{
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int aBegin = wA * 16 * by + wA * ty + tx;
    int bBegin = 16 * bx + wB * ty + tx;
    int bStep = 32 * wB;
    __shared__ float As[16][32];
    __shared__ float Bs[32][16];
    float Csub = 0.0;
    for(int a = aBegin, b = bBegin; a < aEnd; a += aStep, b += bStep)
        As[ty][tx]=A[a]; As[ty][tx+16]=A[a+16];
        Bs[ty][tx]=B[b]; Bs[ty+16][tx]=B[b+16*wb];
        __syncthreads();
        for(int k=0; k<32; ++k)
            Csub += As[ty][k] * Bs[k][tx];
        __syncthreads();
}
```

**Figure 3.1 – Matrix Multiplication by using HiCUDA**

In order to clarify, the code at the right side at Figure 3.1 shows an example which is matrix multiplication with HiCUDA directives. This example consists using shared memory, tiling and memory coalescing techniques. Normally these techniques are known and used frequently among CUDA programmers. Figure 3.1 shows this issue with matrix multiplication example. It can be seen at the first glance that user needs to figure out when should kernel start and end according at Line 12 and 36. Also HiCUDA allows explicit usage of shared memory and global memory. Line 8–10 shows how to use HiCUDA pragmas in order to allocate global memory and Line 23-24 is also for shared memory. Also users should free them at Line 31, 38, 39. Also thread synchronization must be used at Line 26 and 30 to make sure shared memory transfer completion like the Pure GPU programming. It also offers "loop_partition" directive in order to distribute iteration. This directive has clauses to indicate CUDA Blocks, and CUDA Threads.
Chapter 3. Related Works & State of Arts

OpenMP to GPGPU[23]

The another important state–of–art is a cetus compiler which is a research project[23] in order to understand whether code generation for accelerator is feasible or not with existed OpenMP directive (pre 3.0). They didn't used own special directives since it uses existed OpenMP pre 3.x directives. Existed directives take on a new meaning.

Directives are classified into four categories by them:

* **Parallel construct**: Their compiler identifies these regions as a CUDA kernel region.

* **Work Sharing construct (omp for, omp section)**: In order to indicate loop distribution or coarse grain region.

* **Synchronization construct**: (barrier, flush, critical): These directives sometimes can split region into two kernel.

* **Data shared construct**: Data movement and allocation onto device.

This approach also covers auto optimization techniques. Figure 3.2 shows evolution of Cetus compiler. Firstly it optimizes OpenMP program without considering accelerator programming at example A, B, C. After then it starts to translate and optimize CUDA program at example D, E.

**Early experience with OpenMP 4.0 RC2**

One of the recent research project[24] is implemented with draft OpenMP 4.0 directives. This research paper examined the accelerator model of OpenMP (RC2) and shared their experiences with developing preliminary compiler. Their implementation based on ROSE compiler framework [16]. Their research shows how to generate and optimize CUDA code from OpenMP 4.0 (RC2) directives. They explain clearly own approaches and provide couple examples. As it was before, their only focused to generate CUDA codes. They didn't present a runtime.
3.1. Source–to–Source Transformations

(A) Input OpenMP code

```c
#pragma omp parallel for
for (i=1; i<=SIZE; i++) {
    for (j=1; j<=SIZE; j++) {
        a[i][j] = (b[i-1][j] + b[i+1][j] + b[i][j-1] + b[i][j+1])/4;
    }
}
```

(B) Cetus-parallelized OpenMP code

```c
#pragma omp parallel for
#pragma cetus parallel
for (j=1; j<=SIZE; j++) {
    for (i=1; i<=SIZE; i++) {
        a[i][j] = (b[i-1][j] + b[i+1][j] + b[i][j-1] + b[i][j+1])/4;
    }
}
```

(C) OpenMP output by OpenMP stream optimizer

```c
#pragma omp parallel for schedule(static, 1)
for (j=1; j<=SIZE; j++) {
    for (i=1; i<=SIZE; i++) {
        a[i][j] = (b[i-1][j] + b[i+1][j] + b[i][j-1] + b[i][j+1])/4;
    }
}
```

(D) Internal representation in O2G translator

```c
for (tid=1; tid<=SIZE; tid++) {
    for (i=1; i<=SIZE; i++) {
        a[i][tid] = (b[i-1][tid] + b[i+1][tid] + b[i][tid-1] + b[i][tid+1])/4;
    }
}
```

(E) GPU code

```c
if (tid<=SIZE) {
    for (i=1; i<=SIZE; i++) {
        a[i][tid] = (b[i-1][tid] + b[i+1][tid] + b[i][tid-1] + b[i][tid+1])/4;
    }
}
```

Figure 3.2 – Evaluation CUDA from plain OpenMP code by using CETUS compiler
3.2 Directive Based Standards OpenMP 4.0 & OpenACC

OpenMP is a standards of API that supports shared memory multiprocessor programming by C, C++, Fortran. It managed by nonprofit OpenMP Architecture Review Board. The consortium decides and announces standards, then compilers programmers develop OpenMP onto own compiler. OpenMP was supporting only shared memory multiprocessors until version 4.0. However then they decided to supports accelerator devices with version 4.0. On the other hand, although OpenACC is a standards like OpenMP, they only focused accelerators. Therefore even if it is newer than OpenMP, couple of compilers support their standards to code generation for accelerator.

In the following subsections, OpenMP 4.0 and OpenACC constructs are going to been summarized briefly with the aim of identifying those functionalities.

Offloading, kernel configuration and loop execution

OpenMP 4.0 offers the target directive to start parallel execution on the accelerator device. Similarly, OpenACC offers the parallel directive with the same purpose. In OpenACC these regions can be declared asynchronous removing the implicit barrier at the end of the accelerator parallel region, allowing the host to continue with the code following the region. Inside these regions, the programmer can specify teams, representing a hierarchy of resources in the accelerator: a league of num_teams thread teams, each with thread_limit threads. The execution in the teams region initially starts in the master thread of each team. Later, the distribute and parallel for directives can be used to specify the mapping of iterations in different loops to the resources available on the accelerator. OpenACC offers kernels, i.e. regions that the compiler should translate into a sequence of kernels for execution on the accelerator device. Typically, each loop nest will be a distinct kernel. OpenACC also includes the loop directive to describe what type of parallelism to use to execute a loop and declare loop-private variables and arrays and reduction operations. Inside kernels and loops resources are organized in gangs, workers and vectors (indicated with the num_gangs, num_workers and vector_length clauses, respectively), similar to the teams and threads in OpenMP 4.0.

Line 9–10 in Figure 3.3 shows a simple OpenMP 4.0 code where the programmer defines the thread hierarchy and maps to it the execution of the loop in line 11. The target region is inside a task, so in this case the execution in the device is asynchronous to the execution of the master thread in the processor.
3.2. Directive Based Standards OpenMP 4.0 & OpenACC

```c
for(begin=0 ; begin < n ; begin+=stride)
{
  int end = begin + stride - 1;
  int dev_id = (begin / stride) % omp_get_num_devices();

  #pragma omp task
  #pragma omp target device(dev_id) 
  map(to: y[begin:end], x[begin:end]) map(from: z[begin:end])
  #pragma omp team num_teams(16) thread_limit(32)
  #pragma omp distribute parallel for
  for(i = 0 ; i < stride ; ++i)
  { z[i] = a * x[i] + y[i]; }
}
```

Figure 3.3 – Simple code in OpenMP 4.0 using multiple accelerator devices

The `target` directive in OpenMP 4.0 includes the `device` clause, which offloads the execution of the region kernel to a given psychical device (indicated by the integer value in the clause). This direct mapping makes it difficult to write applications that dynamically offload work to accelerators in order to achieve load balancing or adapt to device adaptability. The use of multiple accelerators within a `target data` region is not clear since at most one `device` clause can appear on the directive.

Figure 3.3 shows how the programmer could statically map consecutive `target` regions to the accelerators available in the target architecture (line 4 to compute the device identifier and `device` clause in line 7. Observe that the iteration range for the `for` loop at Line 11 goes from 0 to `stride`, so the program is not sequentially equivalent since it should iterate from `begin` to `end`. I assume that this has been done in this way to ease code generation by the compiler at the expenses of reducing code portability and reusability, in addition to potential programming errors.

**Data motion**

Data copying clauses may appear on the `target`, `target data` and `target update` constructs in OpenMP 4.0 and `parallel`, `kernels` and `data` constructs in OpenACC. The programmer specifies the data motion needed to bring in and out the data for the execution of the region in the accelerator.

For the data items (including array regions) that appear in an OpenMP 4.0 `map` clause, corresponding new data items are created in the device data environment associated with the construct. Each data item has an associated map type which specifies the data copying on entry and exit (`to`, `from` or `tofrom`) or just allocation (`alloc`). OpenACC offers similar clauses (`copying`, `copyout`, `copy` and `create`) and some additional ones (`present_or_ to tell`) the implementation to test if data is already present in the accelerator memory. With all this information,
Example 1 – OpenMP 4.0

```c
void calculate(double *a) {
    #pragma omp target map(to: a)
    // <...code to offload...>
}
```

```c
void foo(double *a) {
    #pragma omp target data map(to: a)
    {
        // <...calculation and offload...>
        fill_oncpu(a);
        #pragma omp target update to(a)
        calculate(a);
    }
}
```

```c
void foo_2(double *a) {
    calculate(a);
}
```

Example 2 – OpenACC

```c
void calculate(double *a) {
    #pragma acc data pcopyin(a)
    // <...code to offload...>
}
```

```c
void foo(double *a) {
    #pragma acc data copyin(a)
    {
        // <...calculation and offload...>
        fill_oncpu(a);
        #pragma acc update device(a)
        calculate(a);
    }
}
```

```c
void foo_2(double *a) {
    calculate(a);
}
```

Figure 3.4 – Another simple code in OpenMP 4.0 and OpenACC to show data regions.

The compiler inserts the associated data allocations and transfers, if they are necessary, to be performed in the most appropriate moment.

Both OpenMP 4.0 and OpenACC offer the possibility of defining data environments in the accelerator for the extent of a region: target data and data, respectively. Inside one of these regions, multiple kernel offloading actions may occur. An executable directive (target update in OpenMP 4.0 and update in OpenACC) is offered to the programmer to update, inside the scope of a data region, the data from the host to the device or vice-versa.

The example in Figure 3.3 shows the use of the map clause in Line 8. It is important to notice that map(to: ...) forces the movement of data when the target region is found; similarly for map(from: ...) which copies from device to host when the target region finishes. The new example in Figure 3.4 shows the use of data regions and the use of the present_or_clauses in OpenACC to instruct the compiler about the use of presence checks. In this simple code, function calculate can be invoked from a previously created data region or not. But array a is in already device at one of them with target data directive. Other function have only one copy of array a in the main memory. Therefore user should extra definition to copy for array a. Besides, there are two different array a after target data directive at foo function. And fill_oncpu changes values of array a which is in main memory. Programmers should be aware if you want to use same data which is in main memory at the device. Thus they use extra directive as target update to notify compiler at Line 13.
Memory hierarchy in the accelerator device

Finally, let’s comment that private, firstprivate and reduction clauses in distribute and parallel for directives give the compiler hints about the use of the memory hierarchy inside the accelerator. Again, OpenMP 4.0 and OpenACC rely on the programmer for the management of the memory hierarchy, having a direct impact in the quality of the kernel codes to be executed on the accelerator device. The simple code in Figure 3.5 shows the use of some of these clauses.

Current Compilers

There is no OpenMP 4.0 compiler which supports accelerators. However OpenACC has 3 different options. PGI, CAPS and CRAY has OpenACC compiler which provide 2.0 version.

3.3 OmpSs Heterogeneous Hardware Support

3.3.1 Nanos++ Runtime

Nanos++[3] is a runtime which has been developing in order to provide parallel environments. It’s also main part of OmpSs suite. Data parallelism is also supported by means of services mapped on top of its task support. Nanos++ also provides support for maintaining coherence across different address spaces (such as with GPUs or cluster nodes). It provides software directory and cache modules to this end.
Chapter 3. Related Works & State of Arts

The main purpose of Nanos++ is to be used in research of parallel programming environments. As such it is designed to be extensible by means of plug-ins. Figure 3.6 shows independent components/plug-ins of Nanos. Currently, runtime plug-ins can be added (and selected for each execution) for:

- Task scheduling policy
- Thread barrier
- Device support
- Instrumentation formats
- Dependences approach
- Throttling policies

Several of such plugins are already available from the library distribution: different scheduling and throttling policies. In particular, device support for CUDA tasks and execution in a Cluster environment.

Figure 3.7 shows Nanos++ behaviour. Basically when Nanos++ will handle a task, firstly it follows these rules respectively: task generation, data dependency analysis, task scheduling. It supports different address spaces such as CPU, GPU and Clusters. Moreover it can handle
3.3. OmpSs Heterogeneous Hardware Support

automatically multi gpu and gpu concurrency by using streams. By OmpSs directives as is seen at Figure 3.8 at Line 4, it can do transparently data-management on GPU side (allocation, transfers, ...) and synchronization. One manager thread in the host per GPU which responsible for: Transferring data from/to GPUs, Executing GPU tasks, Synchronization.

3.3.2 GPU support for OmpSs

With the device clause the programmer informs the compiler and runtime system about the kind of device that can execute the task, not an integer number that explicitly maps the offloading to a certain device as done in OpenMP 4.0. This is a big difference that improves programming productivity when targeting systems with different number and type of accelerators and regular cores. Figure 3.8 illustrates the use of multiple devices in OmpSs. Observe that the programmer does not need to map iterations to devices as was done in the same example in Figure 3.3. The runtime system schedules tasks dynamically in order to make use of all devices available in that moment in the system and to achieve load balancing.

The acc device type is used to specify that the task will make use of OpenMP 4.0 directives

```c
for (begin=0; begin < n; begin += stride) {

#pragma omp target device(acc) copy_in(y[begin:end], x[begin:end]) copy_out(z[begin:end])
#pragma omp task
#pragma omp teams num_teams(16) thread_limit(32)
for (i = 0; i < stride; ++i)
  z[i] = a * x[i] + y[i];
}
```

Figure 3.7 – NANOS++ Behaviour

Figure 3.8 – Using multiple accelerators in OmpSs
to specify what to execute on the accelerator device, relying on the compiler to generate the
kernel code to be executed on the device. I will describe in more detail the current MACC
implementation in Section 4.

The accelerator support in the OmpSs programming model leverages the tasking model with
data directionality annotations already available in the model (that influenced the new depend
clause in OpenMP 4.0) used to compute task dependences at execution time.

OmpSs offers target directive with the following syntax:

```c
#pragma omp target [clauses]
```

where clauses specify:

- device- the kind of devices that can execute the construct (smp, cuda, opencl or acc).
- copy_in - shared data that needs to be available in the device before the construct can
be executed.
- copy_out - shared data that will be available after the construct is executed.
- copy_inout - a combination of copy_in and copy_out above.
- copy_deps - copy semantics for the directionality clauses in the associated task construct
  (i.e., in will also be considered copy_in, output will also be considered copy_out and
  inout as copy_inout).
- implements - an alternative implementation of the function whose name specified in
  the clause for a specific kind of device.
- ndrange - specification of the dimensionality, iteration space and blocking size to repli-
  cate the execution of the CUDA or OpenCL kernel.
- shmem - data that should be mapped into shared–memory among teams in the device.
3.3. OmpSs Heterogeneous Hardware Support

The support for accelerators in OmpSs is rooted on the tasking model and directionality information provided for each task. That information is used by the runtime system to dynamically build the dependence task graph for the program and to schedule the execution of tasks to the available resources, including accelerator devices. The argument directionality information is also used by the runtime, together with the lookahead provided by the availability of the task graph dynamically generated, to schedule data copying actions between address spaces (movements between host and accelerator or between two accelerator devices if needed). This is the purpose of the copydeps clause. If not specified in the task, the programmer can still specify copying actions using the copy_in, copy_out and copy_inout clauses.

Figure 3.9 shows a simple example based on SAXPY. In this example, the task computing saxpy is offloaded to a device using the CUDA architecture while the task check_results is executed in the host. Observe that the output of CUDA task instances are inputs of SMP task instances. The dependences computed at runtime will honor these dependences and the runtime system will take care of doing the data copying operations based on the information contained in the task graph dynamically generated at runtime. The ndrange clause is used to replicate the execution of the CUDA kernel in the device block/thread hierarchy (one dimension with na * na iterations in total to distribute among teams of na iterations in this example).

Multiple implementations tailored to different accelerators/cores can be specified for the same task (currently only available for tasks that are specified at the function declaration/defi-
In this case, the programmer is delegating in the runtime system the responsibility of dynamically selecting the most appropriate device/core to execute each task instance, based on for example the availability of resources or the availability of the data needed to execute the task in the device. With the `implements` clause the programmer can indicate alternative implementations for a task function tailored to different devices (accelerator or host). Figure 3.9 shows the use of the `implements` clause: the `saxpy_smp` function in Line 11 is defined as an alternative implementation to the CUDA implementation of `saxpy` at Line 3. Observe that the programmer simply invokes `saxpy` in Line 29, delegating in the runtime the selection of the most appropriate implementation for each task instance.

### 3.4 C++11 Extended Code Generation

Yet another accelerator code generation model is performing with C++11 lambda facilities. Basically this approach almost same with directive based as one code base idea. However when considering lambda function, it can be say that these are more flexible than directive based. At the first glance, no need to add new directives for every mechanism. The another usable thing is that C++ enable in accelerator.

Currently there is only one release implementation which is SYCL is announced recently by Khronos Group[6]. Another similar approach is from NVIDIA, however their implementation is draft currently. According to the description, SYCL is abstraction layer to provide portability, efficiency of OpenCL while adding easy-of-use and flexible C++. Basically it supports homogeneous code base it means one code base. Developing accelerator code is performed by new C++ facilities. The another thing lies behind of the its is using SPIR backend. Section 2.3 explains what is it and how it works.

On the other hand, NVidia has idea[9] which is almost same with SYCL such as one code base and taking advantage C++. There are no clear information about it, since it is still draft. Figure 3.11 shows several function signatures belong to STD library of C++. Calculation has done by passing function as f parameter.
```cpp
#include <CL/sycl.hpp>
#include <iostream>
using namespace cl::sycl;

constexpr size_t N = 3;
using Vector = float[N];

int main() {
    Vector a = {1, 2, 3};
    Vector b = {5, 6, 8};

    float c[N];
    queue myQueue;
    buffer<float> A { std::begin(a), std::end(a) };
    buffer<float> B { std::begin(b), std::end(b) };
    buffer<float> C(c, N);
    command_group (myQueue, [&] () {
        auto ka = A.get_access<access::read>();
        auto kb = B.get_access<access::read>();
        auto kc = C.get_access<access::write>();
        parallel_for(range<1> {N},
                      kernel_lambda<class vector_add>{ [=] (id<1> index) {
                          std::cout << index.get(0) << " ";
                          kc[index] = ka[index] + kb[index];
                      } }) ;
        std::cout << "Result: " << std::endl;
        for(int i = 0; i < N; i ++)
            std::cout << c[i] << " ";
        std::cout << std::endl;
        return 0;
    });
}
```

Figure 3.10 – Vector Addition Example by SYCL

```cpp
std::vector<int> vec = ... // previous standard sequential loop
std::for_each(vec.begin(), vec.end(), f);

// explicitly sequential loop
std::for_each(std::seq, vec.begin(), vec.end(), f);

// permitting parallel execution
std::for_each(std::par, vec.begin(), vec.end(), f);
```

Figure 3.11 – C++ Parallel Algorithms Library Progress
In this section, mercurium[2] compiler is explained and its new compilation phase are introduced. A new compilation phase MACC\(^1\) has been included in the Mercurium compiler supporting the OpenMP 4.0 accelerator model in an integrated way with OmpSs. MACC takes care of kernel configuration, loop scheduling and use of shared memory for those tasks whose target device is set to acc, leveraging the functionalities already available in the NANOS++\(^3\) runtime and. Therefore in this chapter, first of all is going to mention mercurium compiler.

## 4.1 Basis on Mercurium & MACC Compiler

Mercurium\(^2\) is a open source source–to–source compiler in order to provide agile prototyping. Currently it supports C/C++ and Fortran. It don’t deal to generate object code. Its intention are modifying, rewriting, translation, mixing etc. with an example, providing one code space for heterogeneous programs are possible. Because mercurium can generate many files for different architecture and pass straightly to different back-end compiler.

mercurium has a parser which is able to process C/C++ and Fortran in order to generate abstract–syntax–tree then to use that in common internal representation (IR) format. Immediately afterwards it need to translate own IR to output source code since it’s source-to-source compiler. This phase is called as ”prettyprinting”. One or more than one type of output sources are handled by mercurium driver. In the last step, output codes are passed straightly favorable

\(^1\)MACC is abbreviation of ”Mercurium[2] Accelerator Compiler”.

\(^3\)NANOS++ is a runtime system for parallel applications in C++.
backend compilers which is configured in the mercurium configuration file. In addition to this step, they are linked and binary file is generated by backend compiler.

Using style of mercurium is generally with OpenMP directives by Nanos++ runtime, however it was also extended for many environment such as Cell, software transactional memory or distributed shared memory. It has own plug-in type in order to provide developing extension. mercurium plug-ins can be developed by C++, however it contains C, C++ and Python.

4.1.1 Parsing & Mercurium IR

Mercurium uses GNU Bison[1] for parsing and Flex[5] for lexer. Since it’s working on C/C++ which are ambiguities languages in CF, it should understand ambiguities as well. During parsing ambiguities are detected and Abstract–Syntax–Tree is generated on the fly. Therefore it gathers all ambiguities, summarizes them in the an ambiguity AST node which has all possible interpretations. Syntax is represented by AST. It has many types of AST nodes. Moreover it has also AST nodes for OpenMP and suchlike pragmas. Its AST nodes are generated by a python script.

Figure 4.1 shows how to add distribute directive which was developed to support accelerators in this thesis as a new AST nodes. The current mercurium which i used, already has AST of OpenMP for construct and execution environment. Definition of them can be seen at Line 1–14. Here exec-environment (Line 1) and for-construct (Line 13) are combined nodes. For instance exec-environment can be NODECL_OPEN_M_P*FIRSTPRIVATE or NODECL_OPEN_M_P*SHARED. This means, if a node have exec-environment, mercurium can parse all construct which belong to exec-environment for this node. Line 16 was added as part of OpenMP 4.0 accelerator model. This line is interpreted that DISTRIBUTE node was defined as subset of NODECL_OPEN_M_P, also node DISTRIBUTE has exec-environment and for-construct.

It collects all symbolic information and performs type checking and remove ambiguities.

Then it starts own compiler pipelines. The last step of Mercurium is to produce modified output source code. This step is called as prettyprinting. Finally it invokes back-end compiler and linkers as how are they configured.
4.1. Basis on Mercurium & MACC Compiler

4.1.2 Mercurium Compiler Work Flow

Even though mercurium is a source-to-source compiler, it drives all of compilation workflow. Figure 4.2 is illustrated a workflow from beginning to end with a compilation example. The loop from the compiler driver back to the input source denotes that in some cases, for a given input file, the process is repeated one or more times. As is seen from Figure 4.2, Mercurium uses a plug-in architecture. By this mean mercurium compiler can be extended by means of compiler phases. Besides each plug-in is a compiler phase at the same time. In the beginning of the process, its front-end creates a common Internal Representation (IR) for C/C++ and Fortran. This IR is common for all phases inside of the mercurium. It can be used for analysing or it can be transformed to pass a new version of the IR to next phase. After all is said and done, input IR was already transformed and it is passed to a back-end compilers (e.g., gcc, nvcc). This is because Mercurium has been designed to provide support to multiple programming models. For example, right now it gives support to StarSs and OpenMP pragmas (with the data-flow extensions). When building a new phase for Mercurium it is not always necessary to start from zero. It is chose for CUDA code generation by this facility. Besides it is obvious that Mercurium can contains many compiler phases. Therefore dynamically loadable *.so files are used to keep phases.

Using compiler phases can be changed by pre-defined compiler parameter. For instance there is yet another useful facility of Mercurium is usability of profiling. Because mercurium is able to generate automatically profiling data in the target application format (e.g. paraver[14]). Basically the compiler work where is here that a new phase is added at the end of compilation phases.

At the end of the compiler phases, output source codes are generated from final IR. Output source generation phase is called as prettyprinting phase. As well as generation output source codes by prettyprinting, it has a driver to invoke back-end compiler and linker. Among the

Figure 4.1 – Python script which is involved by Mercurium to generate AST Nodes
these things, output source file or files are available to see for users. After invocation back end compilers, Mercurium finish work flow and it generates object file as is seen from last step of Figure 4.3.

### 4.1.3 MACC Phases Pipeline

MACC compiler consists of several compiler phases. Basically it uses only a phase of NANOS++ runtime due to take advantage of tasking facility. Except this, it doesn’t ignore existed phase and it uses such as OpenMP, Pure Heterogeneous Accelerator support etc.

Compiler pipeline of MACC is illustrated from Figure 4.2. The Last phase is defined in the compiler that generates CUDA codes automatically. It is especially explains at rest of this section.

**OpenMP directive to parallel IR**: After parsing, Mercurium parallel IR which is built by Mercurium parser starts to being treat by this phase. Briefly this phase lowers the semantics of OpenMP into the parallel IR of Mercurium. After parsing, raw IR is lowered here. Basically OpenMP abstract–syntax–tree is built. Because in the each OpenMP directives and clauses is represented with one tree node. for instance "NODECL_OPEN_M_P*DISTRIBUTE" from Figure 4.1 is one AST item and OpenMP directive at the same time.
4.1. Basis on Mercurium & MACC Compiler

Nanos Runtime Source-Compiler Versioning Interface: It's not directly used in the MACC. This phase enables support for '#pragma nanos', the interface for versioning runtime and compiler for Nanos.

Nanos++ lowering: Main lowering phase. Basically it uses built IR from first phase and rebuild almost final IR. All OmpSs and OpenMP parallelization are performed in this step. In the other words this phase lowers from Mercurium parallel IR into real code involving Nanos++ runtime interface. The purpose of involving Nanos++ runtime interface is that in the OmpSs everything is managed and orchestrated by Nanos++ runtime such as tasking, dependency management etc.

On the other hand, there is a important thing that OmpSs heterogeneous device support is not involved by this step. Because after this phase, modified IR is passed into device phases. Since many kind of device can be used, rest of phases can be changed depend on defined device at input source code. MACC CUDA code generator phase didn't being involved with this step.

Main modification in this step is code generation differences when some directives are used.
Chapter 4. MACC: Mercurium ACCelerator Compiler

with OpenMP 4.0 accelerator model. The effects of directives may be slightly change. For instance omp for is a most common directive will effects CUDA threads instead of CPU threads if it’s used inside of the teams / distribute directive. In this case MACC compiler leaves these as they are. They will be handled at last step.

**Nanox SMP support**: SMP is a kind of device and used in order to define to run task on SMP. This phase is used by Nanox phases to implement SMP device support.

**MACC Phase**: Thus far every phases are already involved by Mercurium. This phase is defined in the MACC compiler that allows applying code generation for accelerator by high level directives. A new directive lowerer and code generator are also implemented for this step.

In addition, CUDA and OpenCL tasks can be used in the MACC as well by existed Mercurium phases. In this case MACC will invoke their phases after Nanos++ lowering phase as Nanox SMP support is invoked.

### 4.1.4 MACC Configuration

As mentioned above, Mercurium is a dynamically configurable compiler. Compiler pipeline phases, back end compilers, linker, environment variables, compilation parameters and etc. can be configurable in the special format configuration file. By this mean, compiler developers can manage dynamically phases and generate new compiler work flow with existed phases without the need for any developing. Moreover the configuration files are inheritable each other.

Since MACC is a extension of OpenMP 4.0 accelerator model, it should support existed directives. Besides MACC involve OmpSs task facility. Hence it can be said that MACC is a new phase to generate CUDA codes based on current OmpSs platform.

Figure 4.4 shows MACC compiler configuration file. The mac which is compiler executable name at the same time, is inherited from omp-base at Line 2 in order to support existed OmpSs directives and previous OpenMP version than 4.0 accelerator model. In addition back end compiler and preprocessor are defined as nvcc with some special parameters. And input

---

2SMP is abbreviation of Shared Memory Processor.
4.2 ACCelerator Code Generation Phase

A new compilation phase MACC has been included in the Mercurium compiler supporting the OpenMP 4.0 accelerator model in an integrated way with OmpSs. MACC takes care of kernel configuration, loop scheduling and use of shared memory for those tasks whose target device is set to acc, leveraging the functionalities already available in the NANOS++[3] runtime and. Therefore in this chapter, first of all is going to mention OmpSs platform and its compiler and runtime.

4.2.1 MACC Accelerator Directives

The main intention of MACC is to provide code generation for accelerator by CUDA. While it’s doing this, OpenMP 4.0 directives is used. OpenMP is chose because it’s quite standard among programmers. Moreover many programmers already used them. On the other hand, OmpSs suite has already supported OpenMP directives as it explained Section 3.3. Therefore directives of OpenMP 4.0 is used in this thesis in order to allow to programmers to express own code scopes and offloaded codes.

Some of the OpenMP 4.0 directives for accelerators are simply ignored because OmpSs can delegate their functionality by the NANOS++ runtime, mostly those related with data copying. Figure 4.5 identifies them (extended, implemented, ignored). The main differences and implementation decisions are explained in the following subsections.

Figure 4.4 – MACC configuration file

```c
# OpenMP 4.0 HWA support
language = C
options = --cuda --arch=sm_20
preprocessor_name = /usr/local/cuda/bin/nvcc
preprocessor_options = -D__CUDA__ -I/usr/local/cuda/include -Iinclude nanos-gpu.h
compiler_name = /usr/local/cuda/bin/nvcc
compiler_options = -arch sm_20
linker_name = /usr/local/cuda/bin/nvcc
linker_options = -lnanox-gpu-api -Xlinker -lnanox-ss
compiler_phase = /lib/thnalnax-acc.so
```

language is indicated as C. It will run C parser inside of the compiler. And what’s more in this configuration file, special MACC compilation phase is indicated with a dynamic library at Line 11.
Chapter 4. MACC: Mercurium ACCelerator Compiler

<table>
<thead>
<tr>
<th>OpenMP 4.0</th>
<th>MACC</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>extended (implements ndrange for CUDA and OpenCL kernels)</td>
</tr>
<tr>
<td>map(to/from/tofrom)</td>
<td>implemented but different names (copy_in/out/inout)</td>
</tr>
<tr>
<td>map(alloc)</td>
<td>ignored</td>
</tr>
<tr>
<td>target data</td>
<td>currently ignored</td>
</tr>
<tr>
<td>target update</td>
<td>ignored</td>
</tr>
<tr>
<td>first_/private</td>
<td>implemented</td>
</tr>
<tr>
<td>device(int)</td>
<td>extended</td>
</tr>
<tr>
<td>teams</td>
<td>implemented</td>
</tr>
<tr>
<td>distribute</td>
<td>new clause dist_private</td>
</tr>
<tr>
<td>parallel for</td>
<td>implemented</td>
</tr>
<tr>
<td>distribute parallel for</td>
<td>implemented</td>
</tr>
<tr>
<td>dist_first/last_/private</td>
<td>added</td>
</tr>
</tbody>
</table>

Figure 4.5 – ComparingPragma Directives: MACC vs OpenMP 4.0

4.2.2 Kernel Configuration

When generating kernel code MACC needs to decide: 1) the dimensionality of the resources hierarchy (one-, two- or three-dimension team and thread) and 2) the size in each dimension (number of teams and threads in each dimension). In order to support the organization of the threads in two- and three- dimensions MACC allows the nesting of parallel for directives inside a target region (dimensionality equals the nesting degree). Other proposals considered the use of collapse which includes an integer to specify the number of nested loops with the same purpose. As i will see in the experimental section, deciding which loop in the nest is mapped to each dimension of the thread hierarchy has an impact, and this seems to be out of control from the programmer in the current OpenMP 4.0 specification (OpenACC has three clauses such as gang, worker and vector in order to directly assign iteration), being the responsibility of the compiler. In our current implementation i do not support nesting of distribute directives, which could be useful to decide the dimensionality of the blocks organization.

In order to decide size, MACC takes into account the restrictions of the device (for example maximum number of blocks and threads warp size in the CUDA computing capability) and the information provided by the programmer in the num_teams or max_threads clauses; if not specified, MACC will try to generate as much of possible (e.g. one iteration per block and one iteration per thread), depending on the iteration count of the associated loops. It starts to association with first dimension of threads or blocks. By this means primarily first dimensions are created during kernel configuration by MACC. However starting with second dimension of threads is also experimented as it is seen in the evaluation section.
4.2. ACCELERATOR Code Generation Phase

Example 3 – 1D thread blocks

```c
#pragma omp target device(acc)
#pragma omp teams thread_limit(512)
#pragma omp distribute
for (i = 0; i < n; i++)
#pragma omp parallel for
for (j = 0; j < m; j++)
b[i][j] = a[i][j] * NX;
```

Example 4 – 2D thread blocks

```c
#pragma omp target device(acc)
#pragma omp teams thread_limit(512)
#pragma omp distribute parallel for
for (i = 0; i < n; i++)
#pragma omp parallel for
for (j = 0; j < m; j++)
b[i][j] = a[i][j] * NX;
```

Example 5 – 2D thread blocks

```c
#define nX 4
#define nelem 12000

#pragma omp target device(acc)
#pragma omp teams thread_limit(64)
#pragma omp distribute parallel for
for (i = 0; i < nelem; i++)
#pragma omp parallel for private(k)
for (j = 0; j < nX*nX; j++)
    // < calculations >
#pragma omp parallel for
for (j = 0; j < nX*nX; j++)
    // < calculations >
```

Figure 4.8 shows three simple examples that are mapped into one- and two-dimensional thread organizations. In particular, for the third code MACC decides to generate a two-dimensional thread organization because of the nesting of `parallel for` directives in Lines 8 and 13 inside Line 6. The total number of threads is 64 threads (as specified in the `thread_limit` clause in Line 5), divided in 4 and 16 threads in the first and second dimension, respectively (this is a compiler decision based on the iteration count for the loops). Teams are organized in a single dimension, generating in total 3000 teams in order to execute the 12000 iterations in the loop in Line 7.

4.2.3 Loop Scheduling & Thread Mapping

MACC compiler follows one iteration for one block/thread rule while it’s generating CUDA kernel configurator which is called as `kerneler`. In CUDA accessing global memory is often exploited by warps with coalesced access. In order to exploit data locality, MACC uses cyclic distribution rather than in a block manner. While MACC is doing cyclic loop scheduling, it takes into account conditional and increment expressions.
Chapter 4. MACC: Mercurium ACCelerator Compiler

In order to exemplifying, DG application is used. DG which is a kernel version of a climate benchmark[25] developed by National Center for Atmospheric Research. Moreover this thesis evaluates different kind of DG application at Section 5.3. Before consideration of evaluation results, DG in MACC way and generated CUDA codes are shown. Figure 4.7 shows original DG application with MACC directives. As is seen, only number of team clause is used at line 11. However since MACC maps each iteration with one threads, it generates kerneler. It can be seen at Figure 4.8. Line 7–10 shows MACC calculation in order to find proper number of threads and blocks. since teams size is indicated as 5625 at Line 5 Figure 4.7, MACC generates 16 threads at the first dimension of threads. Because iteration at Line 13 is mapped with one thread/block. It corresponds with a formula at Line 9 Figure 4.8. Besides, other iterations at Line 15 & 39 which are used with `omp parallel for` has same iteration count. Therefore MACC compiler generates nX*nX threads for second dimension as it seen at Line 10 Figure 4.8.

On the other hand, if the iteration count is equal to the number of team/thread among which it is distributed, MACC removes the for-iteration and directly assigns the current dimension of threadIdx or blockIdx to an induction variable. As is seen at Figure 4.7 there are 3 iterations
4.2. ACCELERATOR CODE GENERATION PHASE

```c
void macc_acc_ol_main_0_unpacked( int * const nelem, int * const nXa, int * const j, int * const _macc_nXa, \
    int * const ii, int * const ie, double ** const delta, double ** const grad)
{
    /* Mercurium ACCelerator Compiler - KERNELER */

dim3 gridDim, blockDim;

gridDim.x = MIN(_CUDA_MAX_TEAM,5625);
gridDim.y = 1;
blockDim.x = MIN(_CUDA_MAX_THREAD, ceil (nelem / gridDim.x));
blockDim.y = MIN(_CUDA_MAX_THREAD, nX *nX);
int _macc_dynamic_shared_memory_size
    = (8) * (nX + (0) + 1) + (8) * (nX * nX + (0) + 1) + (8) * (CHUNK + 1) + (8) * (CHUNK + 1);

macc_acc_ol_main_0_unpacked_kernel<<<gridDim, blockDim, _macc_dynamic_shared_memory_size,
        nanos_get_kernel_execution_stream()>>>(nelem, dt, nXa, j, ii, ie, delta, grad);
}
```

Figure 4.8 – Generated MACC Kerneler to configure kernel from Figure 4.7

which are used with OpenMP accelerator model pragma directives at respectively at Line 13, 16 and 40. As mentioned at Section 2.1.1 NVidia GPUs don't have higher capability branch predictor. Therefore MACC compiler has removing redundant for iteration optimization. Figure 4.10 is a generated CUDA kernel code by MACC from Figure 4.7. MACC could map each iteration with block/thread since thread and block size is enough. As is seen at Figure 4.10 there is only one iteration which corresponds actually with nested for iteration at the Line 23. The other iteration at Line 83 is used for `dist_last_private`. It will be explained next section. In the generated CUDA kernel, induction variables are privatized with a prefix as `textbf_macc_`. Starting from here, it can be said that in fact `_macc_ie` and two `_macc_ii` are induction variable. They are mapped directly CUDA threads/blocks and MACC could remove for iteration which will be redundant.

4.2.4 DEVICE LOCAL MEMORY

MACC is able to decide the use of shared memory for threads in a team based on the specification of `private` and `firstprivate` data structures in the `teams` and `distribute` directive, so that each team will have a private copy in its own shared memory. MACC compiler analyzes the size of the data structure to be privatized and generates code for its allocation and copying from global memory in the device to shared memory in each team.
4.2.5 New data sharing attributes for *distribute* directive

MACC extends the current OpenMP 4.0 specification with three data sharing clauses in order to exploit further shared memory. These directives can be used with two ways which are with *chunk_size* or *dist_schedule*(*static,chunk_size*). Because i wanted to use shared memory partially because of limitation and thus i made indication *chunk_size* in two way obligatory to allow to divide each of array into chunks and send each chunk to different team. While programmers are using shared memory, they should make sure that part of array which each team will access must be equal with *chunk_size* each team has.

- *dist_private(list)*: only allocates shared memory up to indicated *chunk_size* in the beginning of the kernel.
- *dist_firstprivate(list)*: allocates shared memory up to indicated *chunk_size* and fills it own part of array from global memory in the beginning of the kernel.
- *dist_lastprivate(list)*: allocates shared memory up to indicated *chunk_size* and fills own part of global memory from own shared memory at the end of the kernel.

The main difference about using data sharing attributes with between *distribute* and *team* is that MACC allocates and copies all size of array when clauses are used with *teams*. If clauses are used with *distribute*, MACC has a chunk size and it can allocate and copy only size of chunk from shared memory. As i mentioned, using *lastprivate* is available.

Figure 4.7 shows *dist_lastprivate* proposal and use of team memory with an example. In this example, *delta* and *grad* are arrays specified as *private* and *firstprivate*, respectively, with *teams* directive at Line 11 in Figure 4.7. Besides, *flx-fly* are specified as *dist_first_lastprivate* with a chunk size at Line 12.

MACC compiler need analysis size of data shared variables since shared memory size is indicated before kernel execution. Figure 4.8 at Line 11–16 shows calculation. Right after that, it is passed as kernel invocation parameter inside of the "«... »" at Line 18.

On the kernel side, compiler generated a code started with identifying pointers for team level values at Line 8, 11 and 14 Figure 4.10. Then after these pointers are filled from global memory. In addition to every thread in the same team has a same pointer since CUDA block uses same
4.2. ACCelerator Code Generation Phase

```c
#pragma omp target device(acc) copyin(key_buff)
#pragma omp task in(nk, key_buff2[0:nk]) inout(key_buff1[0:nk])
#pragma omp teams num_teams(16) thread_limit(32)
#pragma omp distribute parallel for
for (i = 0; i < nk; i++)
#pragma omp atomic
key_buff1[key_buff2[i]]++;
```

Figure 4.9 – Using Atomic (NAS - IS)

shared memory. After the kernel execution MACC compiler generated iteration in order to fill global memory from shared memory. These codes are run at Line 83–87 Figure 4.10 for `dist_last_private` clause.

4.2.6 Reductions and Atomic Operations

Another OpenMP 4.0 features are the reductions and atomic operations. Macc compiler supports two level reduction such as thread and block level as specified at OpenMP 4.0 standard. Besides, I implemented 2 way for atomic operations because of computation capability insufficiency. If the capability of device is compatible to use atomic functions which are provided by CUDA, our compiler generates kernel with these functions. Otherwise it generates its own atomic function with mutex on software level. In addition CUDA doesn’t has atomic functions for every data type. In these cases, I also generates own atomic function. Figure 4.9 shows an example about usage of atomic directive. In this example compiler generates code depends on data type of `key_buff1`.

43
```c
__global__
void macc_acc_ol_main_0_unpacked_kernel (int *nelem, double *dt, int *nXa, int *j, int *i, int *ie, double *grad, double *iebs, double *gw, double *der, double *delta, double *flx, double *fly) {

__int32_t _macc_sh_iter = 0;
__int32_t _macc_sh_var_size = 0;
__int32_t _macc_sh_offset = 0;

double *_macc_grad = (double*) get_shared_memory(_macc_sh_offset);
_macc_sh_offset += (8) * ((nXa * nXa) - (0) + 1) / 8;

double *_macc_flx = (double*) get_shared_memory(32);
_macc_sh_offset += (8) * (BS) - (0) + 1) / 8;

double *_macc_fly = (double*) get_shared_memory(32 + BS);
_macc_sh_offset += (8) * (BS) - (0) + 1) / 8;

for (_macc_sh_iter = threadIdx.y * blockDim.x + threadIdx.x; _macc_sh_iter < BS; _macc_sh_iter += blockDim.x * blockDim.y) {

_macc_flx[_macc_sh_iter] = flx[_macc_sh_iter + BS * blockIdx.x];
_macc_fly[_macc_sh_iter] = fly[_macc_sh_iter + BS * blockIdx.x];
}

macc_sync();

int _macc_ie = 0;
int _macc_ii = 0;

for (_macc_ii = blockIdx.x * blockDim.y + threadIdx.y; _macc_ii < blockDim.y * threadIdx.y; _macc_ii++) {

int _macc_j = 0;
int k,
int l;

k = _macc_ii % 4;
l = _macc_ii / 4;

double s2 = 0.0;
double s1 = 0.0;

#pragma unroll
for (_macc_j = 0; _macc_j < 4; _macc_j++) {

int i = 0;

s1 = (delta[_macc_j * 4 + l] * _macc_flx[(j * 4 * 4 + i + _macc_j * 4) - BS * blockIdx.x] * der[k * 4 + i] + delta[k * 4 + i] * _macc_fly[(j * 4 * 4 + i + _macc_j * 4) - BS * blockIdx.x] * der[l * 4 + _macc_j]) * gw[i];
i ++;
s1 = s1 + (delta[_macc_j * 4 + l] * _macc_flx[(j * 4 * 4 + i + _macc_j * 4) - BS * blockIdx.x] * der[k * 4 + i] + delta[k * 4 + i] * _macc_fly[(j * 4 * 4 + i + _macc_j * 4) - BS * blockIdx.x] * der[l * 4 + _macc_j]) * gw[i];
i ++;
s1 = s1 + (delta[_macc_j * 4 + l] * _macc_flx[(j * 4 * 4 + i + _macc_j * 4) - BS * blockIdx.x] * der[k * 4 + i] + delta[k * 4 + i] * _macc_fly[(j * 4 * 4 + i + _macc_j * 4) - BS * blockIdx.x] * der[l * 4 + _macc_j]) * gw[i];
i ++;
s1 = s1 + (delta[_macc_j * 4 + l] * _macc_flx[(j * 4 * 4 + i + _macc_j * 4) - BS * blockIdx.x] * der[k * 4 + i] + delta[k * 4 + i] * _macc_fly[(j * 4 * 4 + i + _macc_j * 4) - BS * blockIdx.x] * der[l * 4 + _macc_j]) * gw[i];
s2 = s2 + s1 * gw[_macc_j];
}

_macc_grad[_macc_ii] = s2;

_macc_flx = threadIdx.x;
_macc_fly = threadIdx.x;

for (_macc_ii = blockIdx.y * blockDim.x + threadIdx.y; _macc_ii < blockDim.x * threadIdx.y; _macc_ii++) {

_macc_flx[_macc_ii] = flx[_macc_ii + blockIdx.x] * _macc_grad[_macc_ii];
_macc_fly[_macc_ii] = fly[_macc_ii + blockIdx.x] * _macc_grad[_macc_ii];
}
}
```

Figure 4.10 – Generated MACC CUDA Kernel from Figure 4.7
5 Evaluations

The objective of the performance evaluation in this section is to show how the OmpSs proposal to program accelerators behaves, which just integrates those directives from OpenMP 4.0 that are used to specify the kernel computations. For the evaluation I use three simple codes: Jacobi, DG_kernel [25] and CG from NAS Parallel Benchmark [7].

For the experimental evaluation I have used a node with 2 Intel Xeon E5649 sockets (12 cores in total) running at 2.53 GHz and with 24 GB of main memory, and two Nvidia Tesla M2090 GPU devices (512 CUDA cores, compute capability 2.0) running at 1.3GHZ and with 6GB of memory per device. For the compilation of OpenACC codes I have used the HMPP (version 3.2.3) compiler from CAPS [4]. For the compilation of OmpSs codes I have used the Mercurium/Nanos environment [3]. GCC 4.6.1 has been used as back-end compiler for CPU code generation and the CUDA 5.0 toolkit for device code generation.

Performance is reported in terms of execution time for the kernels generated and speed-up, with respect to sequential execution on a single core, for the complete application. In order to obtain these performance metrics I have used nvprof from CUDA 5.0 toolkit.
Chapter 5. Evaluations

```c
int n=512;
int i, begin;
float a = 0.2f, x, y, z;
x = (float*) malloc(n * sizeof(float));
y = (float*) malloc(n * sizeof(float));
z = (float*) malloc(n * sizeof(float));

int stride=128;
for(begin=0 ; begin < n ; begin+=stride)
{
    int end=begin+stride-1;
#pragma omp target device(acc) copy_in(begin, end, a, stride) copy_deps
#pragma omp task inout(z[begin:end]) in(y[begin:end], x[begin:end])
#pragma omp teams
#pragma omp distribute parallel for
for(i = 0 ; i < stride ; ++i)
    z[i] = a * x[i] + y[i];
}
#pragma omp taskwait
```

Figure 5.1 – SAXPY Application with MACC directives

5.1 SAXPY Application

SAXPY stands for “Single-Precision A·X Plus Y”. SAXPY is a combination of scalar multiplication and vector addition, and it’s very simple: it takes as input two vectors of 32-bit floats X and Y with N elements each, and a scalar value A. It multiplies each element x[i] by A and adds the result to y[i]. A simple C implementation looks like this.

Given this basic example code, MACC ways to SAXPY with directives is shows at Figure 5.1. Note that I chose SAXPY because it is a really short and simple code, but it shows enough of the syntax of each programming approach to compare them. Because it’s so simple, and does very little computation, SAXPY is not really a great computation to use for comparing the performance of the different approaches, but that’s not my intent. My goal is to demonstrate how to use multiple gpu with MACC.

Figure 5.1 illustrates basic SAXPY example with directives. In this example arrays which will be calculated are divided by stride and are distributed automatically GPU.

The another important thing is that even if i have one GPU, MACC are going to use CUDA streaming by this means the SAXPY application works concurrent in any case.
5.2 Jacobi Application

Jacobi is a simple iterative program to get an approximate solution of a linear system \( A \times x = b \). In each iteration of an outer `while` loop two nested loops are executed, the second one performing the main computation including a reduction operation. The structure of the code is shown in Figure 5.2, with three different annotations that correspond to three different versions:

- OpenACC baseline version in which each loop is a `target` region with the individual specification of data copying. This version is representative of an OpenMP 4.0 version just changing the names in the directives.

- OpenACC optimized version in which a `target data` region is defined, which includes the two `target` regions mentioned in the previous version. The `target data` directive is used by the programmer to define the data environment for the execution of all kernel instances, minimizing data copying between `target` regions. This version is also representative of an OpenMP 4.0 version just changing the names in the directives.

- The OmpSs/MACC version, equivalent to the baseline OpenACC version in terms of `target` regions but written in OpenMP 4.0. In this version the programmer relies on the runtime system to do all data allocations and copying when necessary. Observe that all `target` regions are `tasks`. This is because the current OmpSs implementation just supports asynchronous `target` regions (not yet in OpenMP 4.0 specification); in this code this does not have any influence due to the serialization caused by data dependences. Two additional tasks are defined for the initialization and computation of the scalar `error` variable, both of them to be executed on the host according to the `smp` type in the `device` clause.

The left plot in Figure 5.3 shows the total execution time of the kernels generated by HMPP and MACC compilers for a data size of 2048 elements. For this code there are no significant differences in the quality of the CUDA kernels automatically generated. The right plot in the same figure shows the speed-up that is obtained for the three versions mentioned above for three different problem sizes: 512, 1024 and 2048. First of all, observe that in OpenACC (and in OpenMP 4.0) the programmer needs to define an external `target data` region to minimize data copying between consecutive `target` regions. This achieves a relative speed-up of 25 between the OpenACC optimized and baseline versions. And second, the performance plot also shows that the runtime system in OmpSs is able to achieve a slightly better performance even with the overheads incurred by keeping track of memory allocations, data copying and orchestration of kernel execution.
Chapter 5. Evaluations

OpenACC baseline

```c
while ((k <= mits) && (error > tol)) {
    error = 0.0;
    #pragma acc kernels copyin(u)
    copyout(uold)
    #pragma acc loop
    for (i = 0; i < n; i++)
        // <. computation.>
    #pragma acc kernels copyin(uold) copyin(u)
    copy(error)
    #pragma acc loop reduction(+ : error)
    for (i = 1; i < (n - 1); i++)
        // <. computation.>
    error = sqrt(error) / (n * m);
    k++;
}
```

OpenACC optimized

```c
#pragma acc data copy(u)
copyout(error)
copy(uold, error)
while ((k <= mits) && (error > tol)) {
    error = 0.0;
    #pragma acc kernels loop
    for (i = 0; i < n; i++)
        // <. computation.>
    #pragma acc update device(error)
    #pragma acc kernels loop
    reduction(+ : error)
    for (i = 1; i < (n - 1); i++)
        // <. computation.>
    error = sqrt(error) / (n * m);
    k++;
}
```

OmpSs/MACC

```c
while ((k <= mits) && (error > tol)) {
    error = 0.0;
    #pragma omp target device(acc) copy, deps
    #pragma omp task in(u) out(uold)
    #pragma omp teams distribute parallel for
    for (i = 0; i < n; i++)
        // <. computation.>
    #pragma omp target device(acc) copy, deps
    #pragma omp task in(uold) out(u) inout(error)
    #pragma omp teams distribute parallel for reduction(+ : error)
    for (i = 1; i < (n - 1); i++)
        // <. computation.>
    #pragma omp taskwait
    error = sqrt(error) / (n * m);
    k++;
}
```

Figure 5.2 – Annotated codes for Jacobi application

Figure 5.3 – Performance evaluation of Jacobi application
5.3 DG-Kernel Application

DG which is a kernel version of a climate benchmark developed by National Center for Atmospheric Research [25]. The structure of the code is shown in Figure 5.4. The code consists of a single target region that is executed inside an iterative time step loop that is repeated for a fixed number of iterations. Inside the target region the iterations of two nested loops (\(ii\) and \(ie\)) are mapped to the teams/thread hierarchy as specified by the programmer.

Figure 5.5 plots the performance that is achieved by different versions of the code, described in the following bullet points:

- CUDA: hand-optimized CUDA version of the application (with host and kernel code written in CUDA).
- OmpSs/CUDA: OmpSs version of the application leveraging (only) the computational kernels written in CUDA.
- HMPP: OpenACC version compiled with the HMPP compiler.
Chapter 5. Evaluations

- MACC: different versions of our OpenMP 4.0 implementation in the MACC compiler, including additional clauses to influence kernel code generation by the compiler.

Comparing bars labelled CUDA and OmpSs/CUDA in the Figure 5.5 one can extract a first conclusion: OmpSs is able to leverage existing CUDA kernels with similar performance as full host/device CUDA codes. In this case i observe a small performance degradation probably due to overheads of the runtime in generating tasks in each iteration of the time step loop that are unnecessary.

The second conclusion from this evaluation is the important role of the compiler in generating efficient kernel codes for the target device. The first 3 bars at Figure 5.6 show the execution time for the original CUDA kernel, the kernel generated by the HMPP compiler and the initial kernel generated by the MACC compiler. As one can observe, the manually programmed CUDA kernel clearly outperforms the kernels generated by the two compilers, which directly translate into significant performance degradation in terms of speed–up for the whole application (first, third and fourth bar, in Figure 5.5).

Thanks to the previous observation and to the study of the kernels available and generated by the compilers, i have been investigating alternative code generation schemes and proposed a new clause for the distribute directive (dist_private explained in Section 4.2.5). The impact of this optimizations is shows in the performance plot at the Figure 5.6: opt1 corresponds to the use of dist_private to force the distribution of a private data structure among the different private memories of the devices; opt2 to simply remove redundant loops when blocks and threads just execute one iteration; and opt3 to change the mapping of loops in a nest to the
5.4 CG Application from NASA Parallel Benchmark Suite

The last code I have selected for the experimental evaluation in this paper is NAS CG. The main computational part of the application contains several loops that can be made tasks and offloaded to a device or executed on the host. The loop that contributes the most to the execution time performs a reduction operation. To execute this loop I want to use the two GPUs available in the node.

With this code I want to show that the `device` clause in OmpSs provides much more flexibility than the current one in OpenMP 4.0 (specifying type of device instead an integer with the device identifier). And I also want to show that breaking the sequential equivalence when using the OpenMP 4.0 `target` construct forces the programmer to change the code in a cumbersome way when moving for one GPU to more than one.
Chapter 5. Evaluations

OmpSs/MACC for a single device

```
#pragma omp target device(acc) copy_deps
#pragma omp task
in (a[0:nz],p[0:na],colidx[0:nz],rowstr[0:na],sum) out(q[0:na], w[0:na])
#pragma omp teams
#pragma omp distribute private(sum)
for (j = 1; j <= lastrow-firstrow+1; j++) |
  sum = 0.0;
#pragma omp parallel for reduction(+:sum)
for (k = rowstr[j]; k < rowstr[j+1]; k++)
  sum = sum + a[k] * p[colidx[k]];
#endif w[j] = sum;
#else q[j] = sum;
```

OmpSs/MACC for 2 devices using unrolling

```
static double *q_1= &q[0];
static double *q_2= &q[mid];
static double *w_1= &w[0];
static double *w_2= &w[mid];
#pragma omp target device(acc) copy_deps
#pragma omp task in (a[0:nz],p[0:na],colidx[0:nz],rowstr[0:na],sum) out(q_1[0:mid], w_1[0:mid])
#pragma omp distribute private(sum)
#pragma omp parallel for reduction(+:sum)
for (k = rowstr[j]; k < rowstr[j+1]; k++)
  sum = sum + a[k] * p[colidx[k]];
```

```
static double *q_2= &q[0];
static double *q_3= &q[mid];
static double *w_2= &w[0];
static double *w_3= &w[mid];
#pragma omp target device(acc) copy_deps
#pragma omp task in (a[0:nz],p[0:na],colidx[0:nz],rowstr[0:na],sum) out(q_2[0:mid], w_2[0:mid])
#pragma omp distribute private(sum)
#pragma omp parallel for reduction(+:sum)
for (k = rowstr[j+smid]; k < rowstr[j+smid+1]; k++)
  sum = sum + a[k] * p[colidx[k]];
```

Figure 5.7 – Annotated codes for NAS CG benchmark

The code in the top part of Figure 5.7 corresponds to the main computational loop, annotated with the OmpSs/MACC directives. This code is written to run in a single device because there is a single task. By simply applying blocking to the j loop, the code would be ready to be executed on multiple devices (code not included due to page limits, to be included in final version if accepted). However, the fact that kernels in OpenMP 4.0 always iterate from 0 forced us to apply unrolling instead, making the code transformation dependent of the number of devices. The resulting code is shown in the lower part of the same Figure.

By using paraver[14] profiler application, MACC compiler generated traces in order to show task distribution when multi gpus are used for same program. The Upper side of the Figure 5.8 shows how to MACC do task scheduling by NANOS++ runtime. Tasks at last 2 rows are run on GPU, rest of them are run on CPU. In these figures, meaning of light blue is idle. So density of tasks for GPUs seems quite good. When is zoomed at selection area, the below figure is caught. In this figure, big green and red are shows most expensive tasks. In fact they are at OmpSs/MACC for 2 devices using unrolling at Figure 5.7. When the original task is separated into 2 tasks at the Figure 5.7, MACC compiler can distribute concurrently on multi GPUs using NANOS++ runtime.
The performance plot in Figure 5.9 shows the speed-up of the GPU accelerated version of NAS CG (bars HMPP, MACC and MACC/2 GPU) and the speed-up using 8 processors in the host, for three different classes of NAS CG. The speed-up with 2 GPU is significant although i only refined one of the loops given the very simple source code transformation. Note that data transfers between GPUs will take place, automatically handled by the runtime.

**5.5 EP Application from NASA Parallel Benchmark Suite**

In the EP which is embarrassingly parallel benchmark, two-dimensional statistics are accumulated from a large number of Gaussian pseudo random numbers, which are generated according to a particular scheme that is well suited for parallel computation. EP is a simple and small kernel that generates pairs of Gaussian random numbers according to a specific scheme. EP is characterized by the fact that no inner-communication is required among all processors. EP gives a good estimate to the maximum floating point performance can be achieved.

EP kernel is a communication-less application. What i se in the table is a pure execution time. By this reason, it can be get quite well speedup. The EP program which is developed by MACC
Figure 5.9 – CG - NAS Application

pragma can see at Figure 5.10. As it seen this figure, there is one main iteration inside of main computation area. It uses 3 different arrays at different time. There are no data dependencies between these arrays inside main iteration. Therefore this application is called Embrassingly Parallel Application. Its speedup can reach up to number of the computation cores.

The performance plot in Figure 5.11 shows the speed-up of the GPU accelerated version of NAS EP (bars HMPP, MACC). As it seen in this figure, speedup of MACC and HMPP are almost same. Because the application is already parallelised maximum.
5.5. EP Application from NASA Parallel Benchmark Suite

```c
#pragma omp target device(acc) copy_deps
#pragma omp task in(qq) out(x)
#pragma omp team reduction(+:sx, sy)
#pragma omp distribute private(kk, t1, t2, t3, k_offset, an, ik)
for (k = 1; k <= np; k++) {
    kk = k_offset + k;
    t1 = S;
    t2 = an;
    for (i = 1; i <= 100; i++) {
        ik = kk / 2;
        if (i) t3 = randlc(&t1, t2);
        if (ik == 0) break;
        t3 = randlc(&t1, t2);
        kk = ik;
    }
}

double vx1, vx2, vx2, vx3, vx4, vt1, val1, a2;
val1 = (int) v1;
a2 = A - t23 * val1;
double loc_x = t1;

#pragma omp parallel for private(vx1, vx2, vx2, vx3, vx4, vt1, val1, a2)
for (i = 0; i < 2 * NK; i++) {
    vt1 = r23 * (loc_x);
    vx1 = (int) vt1;
    vx2 = loc_x - t23 * vx1;
    vt2 = (int) vt1;
    vx2 = loc_x - t23 * vx1;
    vt3 = t23 * vx3 + a2 * vx1;
    vt4 = (int) vt3;
    loc_x = vt3 - t46 * vt4;
    x[i] = r46 * (loc_x);
}

double psx, psy;
psx = psy = 0.0;

#pragma omp parallel for reduction(+:psx, psy) private(x1, x2, x1, x2, x3, x4, l)
for (i = 0; i < NK; i++) {
    x1 = 2.0 * x[2 * i] - 1.0;
    x2 = 2.0 * x[2 * i + 1] - 1.0;
    t1 = (x1 * x1) + (x2 * x2);
    if (t1 < 1.0) {
        t2 = sqrt(-2.0 * log(t1) / t1);
        t3 = (x1 * t2);
        x = max(fabs(t3), fabs(t4));
        psx = psx + t3;
        psy = psy + t4;
    }
}
sx += psx;
sy += psy;
```

Figure 5.10 – Annotated codes for NAS EP benchmark
Figure 5.11 – Performance evaluation for NAS EP
6 Conclusion

6.1 Summary

Source–to–source translation for accelerators are already reviewed at the other studies\cite{22, 23, 24}. Moreover many companies which are leading HPC area such as PGI\cite{15}, CAPS\cite{4} and CRAY present own OpenACC compiler. In fact HPC scientists or engineers are using directive based programming for heterogeneous architectures in the real life. In addition Intel also published own OpenMP 4.0 compiler for Xeon PHI co processors.

When they are compared each other, most of them are focused only for code generation of the accelerator. Only early experiences OpenMP model\cite{24} studied current OpenMP directives

<table>
<thead>
<tr>
<th></th>
<th>HiCUDA</th>
<th>CETUS</th>
<th>ROSE</th>
<th>OpenACC</th>
<th>SYCL</th>
<th>OmpSs</th>
<th>MACC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Knowledge</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>-</td>
<td>no</td>
</tr>
<tr>
<td>Offload</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Code Gen</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Runtime</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>Depends</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Annotation</td>
<td>pragma</td>
<td>pragma</td>
<td>pragma</td>
<td>pragma</td>
<td>C++11</td>
<td>pragma</td>
<td>pragma</td>
</tr>
<tr>
<td>Thread Map</td>
<td>explicit</td>
<td>implicit</td>
<td>implicit</td>
<td>explicit</td>
<td>explicit</td>
<td>-</td>
<td>explicit</td>
</tr>
<tr>
<td>Shmem</td>
<td>explicit</td>
<td>implicit</td>
<td>implicit</td>
<td>implicit</td>
<td>explicit</td>
<td>-</td>
<td>Both</td>
</tr>
</tbody>
</table>

Figure 6.1 – Comparing Discussed Compilers and MACC
but there is no real compiler that currently supports task model of 4.0 and memory model, they only analysed particular area of the accelerator. This thesis got interested about that what happened when new task model and accelerator code generation model are integrated. Because currently, accelerator programming have to been managed by host. Also everybody knows about this handwritten CUDA program usually better if it is developed optimized. Therefore i wanted to exploit OmpSs advantage to get close CUDA performance.

In order to summarize all accelerator model approaches which are mentioned in this thesis, Figure 6.1 shows main differencies which were discussed in the thesis between accelerator programming environments.

6.2 Conclusion

After long years of existence, the CUDA programming framework has become a significant part of HPC programming. Each version of CUDA brings important features and support for the new available products, but is still in a constant evolution. Therefore developing logic of CUDA program is usually changing. Moreover, writing CUDA code can gain good performance from the GPU acceleration remains a painstaking job for the programmer. Therefore MACC compiler is presented as source–to–source compiler model in order to deal with this kind of problems. MACC compiler model for translating standard OpenMP 4.0 directives onto OmpSs into CUDA based GPGPU programs is explained. In addition, the compiler model of this thesis has some advantages when compared OpenMP 4.0 standard (e.g. more flexibility in the use of multiple GPU, shared memory, data managing, ...).

The OpenMP accelerator model is based on CPU likely accelerator. This thesis is working with different architectures and different memory areas. Therefore assumption of the user about the output code is becoming more difficult day by day. However, in any case in order to develop high performance applications, user should know the target architecture. Because this issue is same on the CPU. Programmers sometimes need to change their own algorithm to leverage parallel multiprocessor performance when they are using OpenMP. Due to the these reasons this thesis is going to work on new directives to allow editing transformation to user.

In addition, generally difference about usage shared memory is encountered. According to the CUDA model, shared memory is used in order to fill from global memory or to fill to global memory. By this means, user gains a significant performance as speed up. In the MACC compiler model already presented using shared memory to fill from global memory. And
the effect of using shared memory is analysed. In fact, this kind of data variable usage is like \textit{lastprivate} clause logic. However OpenMP 4.0 doesn’t have \textit{lastprivate} clause for the \textit{team} directive, therefore this kind of usage haven’t been implemented yet.

In conclusion briefly, in this thesis presents the main design considerations that are embedded in our current implementation of the OpenMP 4.0 [12] accelerator model in OmpSs, making emphasizing on the roles of the programmer, compiler and runtime system in the whole picture. The compiler plays a key role and for this reason previous efforts have been devoted to the automatic generation of device-specific programs from high-level programs annotations such as OpenMP and OpenACC [11], including both research efforts at academia [22, 23, 24] as well as commercial implementations [4, 15]. Our compiler implementation in Mercurium [2] has been useful to experiment with different code generation strategies, trying to foresee the need for new clauses in current OpenMP 4.0 specification. OmpSs [19] is strongly rooted on the assumption that the runtime system should play a key role, making appropriate use of the information that can be gathered at execution time. In this thesis emphasizing this aspect supported by an experimental evaluation on three application kernels is tried.
Bibliography


