UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
BACHELOR OF COMPUTER SCIENCE

LEONARDO PILETTI CHATAIN

Hybrid Parallel Programming - Evaluation of OpenACC

Final Report presented in partial fulfillment of the requirements for the degree of Bachelor of Computer Science

Prof. Dr. Nicolas Maillard
Advisor

Porto Alegre, December 2012

CIP – CATALOGING-IN-PUBLICATION

Chatain, Leonardo Piletti

Hybrid Parallel Programming - Evaluation of OpenACC / Leonardo Piletti Chatain. – Porto Alegre: COMGRAD CIC UFRGS, 2012.

43 f.: il.

Final Report (Bachelor) – Universidade Federal do Rio Grande do Sul. Bachelor of Computer Science, Porto Alegre, BR–, 2012. Advisor: Nicolas Maillard.

1. OpenACC. 2. GPGPU. 3. GPU programming. 4. CUDA. I. Maillard, Nicolas. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitora de Graduação: Prof^{a}. Valquiria Link Bassani
Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb
Coordenador do CIC: Prof. Raul Fernando Weber
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“Experience is what you get when you didn’t get what you wanted.”
— RANDY PAUSCH

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GPGPU	General Purpose GPU
GPU	Graphics Processing Unit
SGEMM	Single-precision General Matrix Multiplication
SIMT	Single Instruction Multiple Thread

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ABSTRACT

OpenACC is a new specification for a hybrid (CPU + GPU) parallel programming API, in which the programmer uses compiler directives to distribute the computation between the GPU and the CPU.

With a similar paradigm to OpenMP, OpenACC presents clear advantages in terms of ease of programming. Regarding performance, however, a comparison between OpenACC and CUDA has not yet been made.

This study aims to evaluate OpenACC, establishing a comparison with CUDA. Furthermore, this work aims to identify the main limitations of OpenACC, analyzing its impact on performance.

The evaluation is made using three different benchmarks (*matrix transpose*, *dot product* and *matrix multiplication*), each one comprising several implementations.

Our results show that, although being in some cases notably slower than optimized CUDA, OpenACC implementations can still benefit from significant performance improvements over serial programs executed on the CPU. Moreover, when compared with less optimized CUDA implementations, OpenACC is shown to provide competitive performance with a much simpler programming model.

Keywords: OpenACC, GPGPU, GPU programming, CUDA.

Programação Paralela Híbrida - Avaliação de OpenACC

RESUMO

OpenACC é uma nova especificação para uma API de programação paralela híbrida (CPU + GPU), na qual o programador utiliza diretivas de compilação para distribuir a execução do programa entre a CPU e a GPU.

Em comparação a CUDA (programação GPU direta) OpenACC apresenta vantagens claras em termos de facilidade de programação (à semelhança de OpenMP). Com relação à performance, entretanto, ainda não existe uma comparação entre OpenACC e CUDA.

Este trabalho visa fazer uma avaliação de OpenACC, estabelecendo uma comparação com CUDA. Adicionalmente, este estudo visa identificar as principais limitações de OpenACC, estudando seu impacto no desempenho.

A avaliação é feita usando três diferentes benchmarks (*matrix transpose*, *dot product* and *matrix multiplication*), cada um composto de diversas implementações.

Nossos resultados mostram que, ainda que em alguns casos notavelmente mais lentas que CUDA, implementações em OpenACC podem apresentar melhorias significativas de desempenho se comparadas a programas seriais executados na CPU. Em comparação a implementações CUDA menos otimizadas, OpenACC apresenta uma performance competitiva com um modelo de programação muito mais simples.

Palavras-chave: OpenACC, GPGPU, CUDA, Programação em GPUs.

1 INTRODUCTION

For many years programmers relied primarily on single core processors to perform virtually any general purpose task. GPUs, on the other hand, were used almost exclusively for graphic applications.

Following the demands of the entertainment industry, GPUs evolved to become massively parallel devices, capable of processing the thousands of triangles per second required to render the latest generation of games, a level of arithmetic throughput CPUs were not able achieve. Figure 1.1 (NVIDIA, 2012) shows the recent evolution of CPUs and GPUs in terms of GFLOPS (Floating Point Operations per Second).

In 2001, with the introduction of programmable shaders, developers were given control over the exact computations the GPU would perform. The programming model was, however, still too focused on graphics to be efficient and accessible when solving other kinds of tasks.

In 2006 NVIDIA introduced CUDA, a parallel computing platform that enabled programmers to write general purpose code that would run on GPUs. In 2007 Tesla was announced: the first GPU specifically designed for general purpose computing. Since then, the CUDA platform has been steadily improved in terms of software (with additions to the API and introduction of higher level libraries) and hardware (with the loosening of constraints and introduction of features such as caching).

As of November 2012, CUDA enabled GPUs are present in 62 supercomputers of the *Top500 list*, including *Titan*, the current leader of the ranking (TOP500, 2012).

However, despite the significant improvement over previous GPGPU alternatives, CUDA programming still presents some challenges, such as:

Portability CUDA is a proprietary platform, with hardware and software provided and supported only by NVIDIA.

Ease to code/Maintainability CUDA code is still arguably hard to write and maintain.

An alternative to minimizing portability issues is *OpenCL*. Released in 2008, OpenCL is an open standard initially developed by Apple and now maintained by the Khronos Group (GROUP, 2008).

OpenACC, another open standard, was released in 2011 and aims to solve the second issue (OPENACC, 2011a). Similarly to OpenMP (CHAPMAN; JOST; VAN DER PAS,

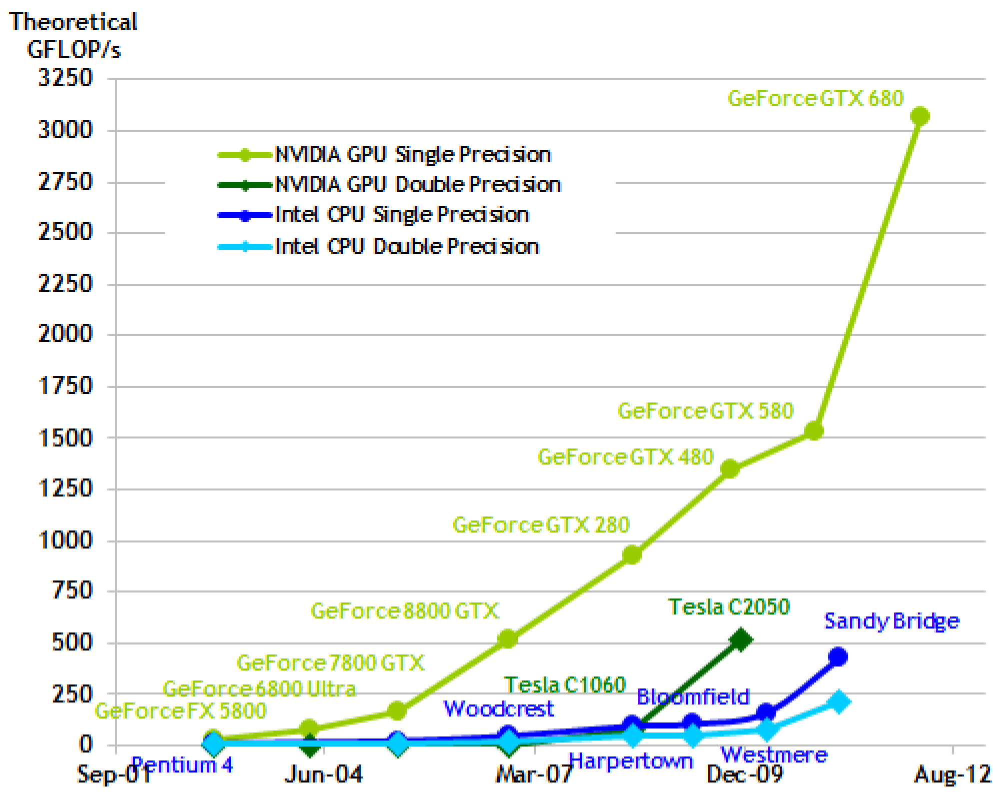


Figure 1.1: GPU performance vs CPU performance.

2007), with OpenACC programmers can use directives to determine that areas of the code should be parallelized and executed on the GPU.

1.1 Objective

OpenACC is a fairly recent standard that has not yet been evaluated. The goal of this study is to provide an objective evaluation of OpenACC and a comparison with CUDA in terms of features, performance and ease of programming.

Moreover, this work aims to answer the question: *how much of CUDA's performance can be matched by OpenACC, and what exactly are the advantages in terms of ease of programming?*

Finally, we seek to identify the main limitations of OpenACC and what is their impact on performance.

1.2 Structure of this document

Chapter 2 provides an overview of the current GPGPU programming models, discussing the OpenACC standard. Chapter 2 also introduces the NVIDIA architecture, explaining the performance bottlenecks and which CUDA mechanisms can be used to achieve maximum performance.

Chapter 3 explains the benchmarks that were used, the CUDA solutions and how OpenACC was used. Chapter 4 shows the experimental results.

2 HYBRID PARALLEL PROGRAMMING

The main idea of hybrid parallel programming is to have architecturally different computing devices working together to achieve performance. Furthermore, the goal is to have each part of the code running on the architecture that suits it better.

With the rise of GPU computing many hybrid computing platforms consist of CPUs and GPUs. In the configuration targeted by this study, hybrid programs comprise two parts: one executing in the *host* (CPU) and one executed in the *device*, or *accelerator* (GPU).

2.1 CUDA

CUDA provides a relatively low-level programming model, in which developers are directly responsible for the use of the resources available on the GPU, such as different memories and thread organization.

This section describes CUDA in detail, providing a basic understanding of the GPU architecture and explaining the mechanisms in CUDA which provide high performance.

NVIDIA architecture is constantly evolving, and this evolution usually affects the programmability, of CUDA devices. A device "programability" is represented by its *compute capability* (c.c.). As of the date of this study, three major architectures have been released: *Tesla* (c.c. 1.x), *Fermi* (c.c. 2.x) and Kepler (c.c. 3.x).

This study was made using a *Fermi* card, and some of the numbers and characteristics presented are specific to this architecture.

2.1.1 Execution Model

In CUDA, GPU code is written in special functions called *kenels*. Kernels are functions that, when launched, are executed in parallel across several *threads* on the GPU.

It is important to note that, in comparison with their CPU counterparts, GPU threads are extremely lightweight, with the cost of context-switching being of only a few cycles (KIRK; HWU, 2010). Because of such low scheduling overhead and of the massively parallel nature of the GPU architecture, launching a high number of threads is often required in order to hide I/O latency with computation.

Threads are organized into *blocks*, which can have one, two or three dimensions, as defined by the programmer. Blocks, in turn, are organized into one-dimensional, two-

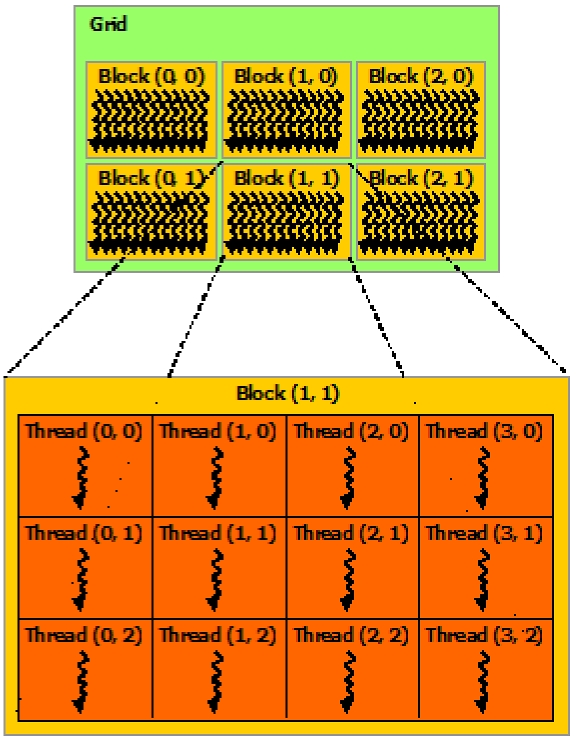


Figure 2.1: Grid of thread blocks.

dimensional or three-dimensional *grids*. The thread hierarchy can be seen in Figure 2.1.1 (NVIDIA, 2012).

Furthermore, CUDA executes instructions in a SIMT (Single Instruction Multiple Thread) fashion. In the GPU, threads are divided into groups called *warps*, which execute in lock-step (all threads in a warp execute the exact same instruction at the same time).

2.1.2 Memory Hierarchy

There are 4 different kinds of GPU memory available to programmers:

Registers Registers, also known as *local memory*, are private to *threads* and represent the fastest available memory. Registers also represent a very small portion of the total memory.

A *GTX 480* has 32768 available registers *per block*. A kernel launched with 256 threads per block, for example, contains only 128 registers per thread.

Shared memory Shared memory is local to *blocks*, representing the second fastest memory available. As the name suggests, shared memory can be shared across all threads inside the block. Shared memory is also a relatively scarce resource.

A *GTX 480* has 48 KiB of shared memory available *per block*. Previous generation graphic cards had only 16 KiB.

Global memory Global memory is the largest and slowest memory available, and is accessible to all threads. This is also the memory the *host* can read and write.

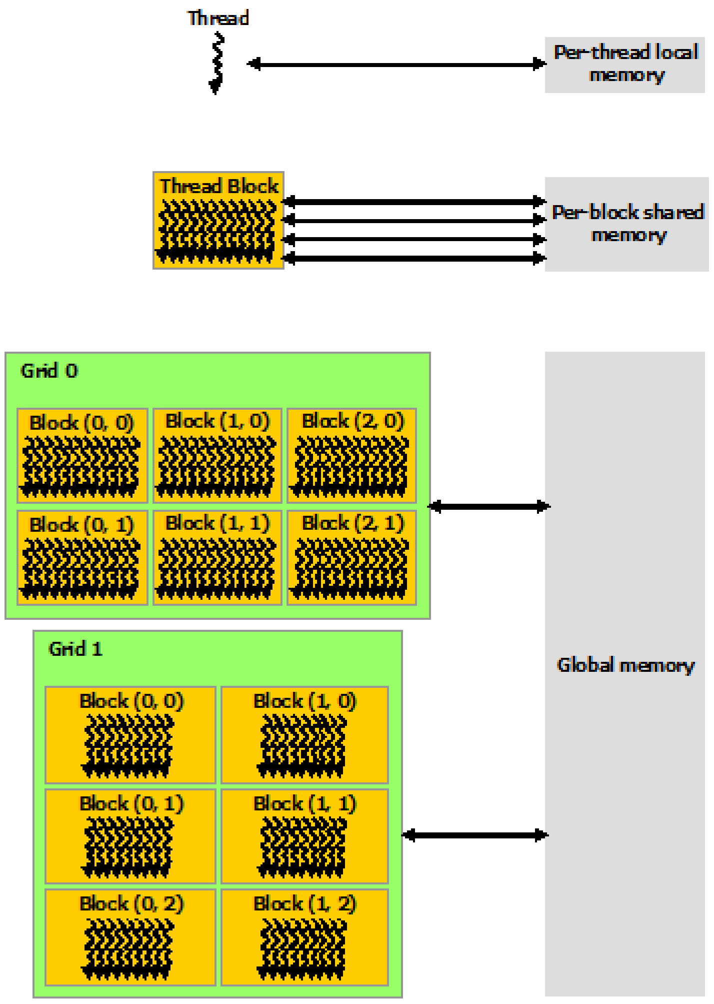


Figure 2.2: CUDA memory hierarchy.

A *GTX 480* has approximately 1.5 GiB of global memory.

Constant memory Constant memory is small and globally accessible. It is faster than the global memory, but slower than the shared memory.

A *GTX 480* has 64 KiB of constant memory.

Figure 2.1.2 (NVIDIA, 2012) shows the different memories and their relation with threads, blocks and grids.

As stated before, the programmer is responsible for the use of each of these memories.

Additionally, the Fermi architecture also features L1 and L2 caches to the global memory, which cannot be directly controlled by the programmer[1] (NVIDIA, 2011).

2.1.2.1 *Global memory accesses and coalescing*

In c.c. 1.x GPUs (Tesla architecture), when a *warp* (a group of threads executing in lock-step) fetched data from the global memory, one of the following would happen:

[1]The Fermi architecture features a 64 KiB shared-memory, that can be configured into 16 KiB of L1 cache and 48 KiB of shared-memory or the other way around.

- If the requests were aligned[2], memory would be accessed in a single transaction[3].
- Otherwise, one transaction would be issued per request.

Because global memory accesses are much slower than computations or that other memory accesses, ensuring coalescing is a top priority when developing CUDA programs.

The Fermi architecture, on the other hand, introduces a cached global memory access scheme which greatly relaxes the constraints for high efficiency memory utilization.

Due to the design of the cache, however, some access patterns will still degrade performance, for example:

All threads requesting the same element this will cause all requests to be serialized.

Strided accesses this will cause accesses to fall into different cache-lines. The effects of this will be seen in practice in Chapter 3.

Overall, even though the caching system is present, the general performance guideline is to still strieve for coalescing whenever possible.

2.1.3 Wrap up

CUDA is a powerful platform for programming GPUs, and offers great control to the programmer. It also leaves on the hands of the programmer most of the resposibility for achieving performance.

Some of the main goals when designing CUDA algorithms are:

- Ensuring that there is work available for all threads. This is done by chosing massively parallel problems and tuning the number of threads and blocks.
- Ensuring that there are enough threads to hide memory latency.
- Using registers and shared memory as caches to the global memory.
- Ensuring that global memory accesses are efficient.

2.2 Alternatives to “direct” GPU computing

As an attempt to make CUDA and general purpose GPU programming easier, several alternatives have arised. This is a short, non extensive, description of the main available options:

Thrust Thrust is a high level CUDA library of algorithms and data structures, similar to the C++ Standard Template Library (STL). Thrust lets programmers transparently manage vectors on the GPU and features a comprehensive set of algorithms such as sort, prefix-sum, transform and reduce (HWU, 2011).

[2]Depending on the exact architecture the requirements for alignment would be more or less strict, but it would usually involve that consecutive threads loaded consecutive regions of the memory.

[3]Technically, one transaction would be issued per half-warp.

cuBLAS, cuFFT, cuSPARSE, NPP These are CUDA versions of numeric libraries. cuBLAS (NVIDIA, 2008) is the CUDA version of the standard BLAS library. cuFFT (NVIDIA, 2010) provides a FFT implementation. cuSPARSE is the CUDA Sparse Matrix library, and NPP (Performance Primitives library) is a collection of image, video and signal processing algorithms.

Matlab/Mathematica both Matlab and Mathematica either directly support or provide CUDA plugins that enable some of the numerical computations to be performed on the GPU.

These are some of the alternatives that avoid coding directly in CUDA, also being extremely optimized and well maintained. On the other hand, these options provide a limited set of functionality.

2.3 OpenACC

As opposed to the previous alternatives, OpenACC is not restricted to a limited set of algorithms. With a paradigm similar to OpenMP, OpenACC defines a group of directives that work as hints to the compiler, enabling it to offload compute-intensive regions to an external accelerator (GPU) without any additional work of the developer.

Furthermore, OpenACC provides a generic programming model in which the same code can be compiled to target different architectures, or even run sequentially in the absence of a GPU.

OpenACC is an open standard, maintained by a group of four enterprises: *CAPS Enterprise*, *CRAY Inc*, *The Portland Group Inc (PGI)* and *NVIDIA*.

2.3.1 Available compilers

Despite being an open standard, there are currently no free implementations of OpenACC. By the date of this study the only compilers supporting OpenACC are offered by *PGI*, *CAPS* and *Cray*. For this work the *PGI Accelerator Compiler v12.9* was used (GROUP, 2012).

OpenACC is available for C and Fortran. This study uses only the C version, but there are no reasons to believe there would be any significant difference if Fortran was used.

2.3.2 Programming model

OpenMP programmers will be somewhat familiar with the OpenACC paradigm: both use directives to generate parallel code out of `for` loops. However, OpenACC's target architecture, the GPU, is fundamentally different from OpenMP's. Specific details such as CPU-GPU data movement and kernel launch configuration are some of the key differences between OpenMP and OpenACC.

The following sections aims to provide a short introduction to the main features in OpenACC, especially the ones used in this work. More information about the OpenACC specification can be found in (OPENACC, 2011b).

2.3.3 Directives

The general syntax of an OpenACC directive is:

```
#pragma acc <directive-name> [clauses]
```

There are basically two kinds of directives/clauses: execution directives and data directives.

2.3.3.1 Execution directives/clauses

The main directive is `kernels`, that specifies a region of the code to be translated into a GPU kernel.

Inside a *kernel* area, loops can be annotated with the `loop` directive, defining one dimension of the kernel.

The following clauses can be used:

vector defines the number of threads per block.

gang defines the number of blocks launched. Threads may operate on more than one element if the total number of threads is smaller than number of elements. The total number of threads is defined as the number threads per block times the number of blocks.

independent Dependency occurs when computations inside a loop depend on data external to the loop. If dependent code is wrongly parallelized it might lead to errors on the result. Because of such errors, the compiler is very conservative, and might not parallelize code, even if it does not present dependencies. This directive tells the compiler to assume the loop is independent.

2.3.3.2 Data directives/clauses

OpenACC data directives define how data is copied across host and device. In the absence of these directives, the compiler may generate unnecessary transfers. As seen in Subsection 2.3.4, optimizing data transfers is a key detail for improving performance.

A data region can be defined with the `data` directive. The main clauses are:

copy Copies the data specified to the device in the beginning of the region, and back to the host at the end.

copyin Copies data to the device, and does not copy it back at the end.

copyout Creates uninitialized data on the device and copies it to the host at the end of the region.

create Creates uninitialized data on the device. Never copies it to the host.

present Tells the compiler that the data required had already been copied/created on the device by a previous directive and there is no need to copy anything.

pcopy / pcreate / pcopyin / pcopyout The initial *p* is a shortcut for *present_or*. These clauses do exactly the same as their counterparts, but the copy will only happen if the data is not present.

2.3.3.3 Example

```
void sum(float * restrict a, float * restrict b, int n) {
#pragma acc kernels loop vector(256) copy(a[0:n]) copyin(b[0:n])
  for (int i = 0; i < n; i++) {
    a[i] = a[i] + b[i];
  }
}
```

Listing 2.1: Vector sum example.

Listing 2.1 shows an example code for the sum of two vectors. This code launches a one-dimensional grid of one-dimensional thread blocks, each block is composed of 256 threads. The amount of work per thread, as well as the amount of thread blocks is defined at launch time.

The data clauses in Listing 2.1 tell the compiler to copy the a vector back and forth, and to only copy b from the host to the device (and not back).

2.3.4 Performance guidelines

There are a few generic peformance guidelines for OpenACC. These guidelines are based mostly on how to ensure that the compiler has enough information to perform the best parallelization.

Pointer arithmetic In a low-level programming language such as C, arrays are represented as contiguous areas of memory. With this representation, accesses to elements are done with simple pointer arithmetic, and there are no guarantees that the memory position being accessed is within the limits of the array. Moreover, from the compiler's point of view there are no guarantees that a pointer from one array will never "step over" other's data.

Pointer arithmetic is not allowed inside OpenACC compute regions (only accesses through indices, using the `element[i]` syntax). Moreover, the `restrict` keyword can be used in array declarations to tell the compiler that arrays will never overlap.

Additionally, irregular memory access patterns should be avoided at all cost. Each iteration of the loop will be mapped to a different thread and, as discussed in Section 2.1, complex memory access patterns incur in a significant overhead in CUDA.

Data movement data movement between the host and the device is very expensive, and should be avoided whenever possible. Eliminating useless data transfers can be done using the *data* directives as seen previously.

Other important aspects of minimizing data transfers include to allocate multidimensional arrays in contiguous area of memory, enabling data to be copied over much faster.

Minimize dependencies Dependencies between loops should be minimized whenever possible. The key for this is to try to keep data inside loops as private as possible. Keywords such as `restrict` and `independent` help the compiler to generate the best possible code.

2.3.5 Runtime API

The runtime API specifies routines to querying device information or chose which device to be used. During this study no runtime function was used, except for *acc_ init(device)*, which tells OpenACC to initialize the device, theoretically saving initialization time from the first call.

However, in practice we found that the performance of the first execution was always poorer than the rest of executions. To work around such limitation we execute all kernels once before start measuring time.

2.4 Wrap up

This chapter introduced the concept of hybrid programming and provided an introduction to CUDA and OpenACC.

3 BENCHMARKS

Three different benchmarks were evaluated on this study of OpenACC: *matrix transpose*, *dot product* and *matrix multiplication*. Each benchmark involves one or more OpenACC implementations that are compared against one or more CUDA implementations.

This chapter introduces the experiments and explains the objectives of each one, as well as the challenges involved and the solutions adopted. All CUDA implementations and ideas were taken from literature, while all OpenACC solutions were developed during this study. Results are shown in Chapter 4.

3.1 Matrix Transpose

In linear algebra, a matrix can be transposed by being reflected over its main diagonal. The rows of the transposed matrix are therefore the columns of the original matrix and vice versa. Figure 3.1 shows a matrix and its transpose.

Implementation-wise, matrices are internally represented as contiguous regions of memory (organized in row-major order in this study). As it can be seen in Figure 3.2, this layout causes memory to be sometimes accessed in a non-contiguous way when performing the transpose.

In a GPU programming context, benchmarking different matrix transpose methods has no other purpose but to measure the impact of different memory access patterns on the overall performance.

This benchmark will be structured around the comparison of a simple matrix copy kernel with several transpose implementations. For the sake of simplicity, the matrices used on all benchmarks are square.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \qquad A' = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Figure 3.1: Matrix A and its transpose A'.

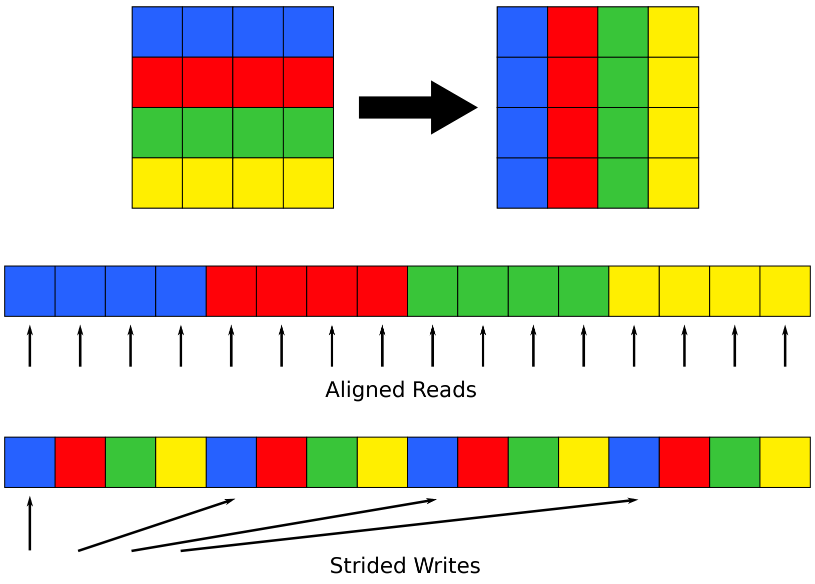


Figure 3.2: Non-coalesced write accesses on row-major matrix transpose.

3.1.1 Transpose in CUDA

The CUDA benchmark features 4 kernels: *copy*, *naive transpose*, *coalesced transpose* and *bank conflict free transpose*. All implementations are described in (RUETSCH; MICIKEVICIUS, 2009).

All kernels are launched with blocks of size *32x8*, and each thread block operates on a *32x32* block of the matrix. With this setup, each thread is responsible for transposing/-copying 4 elements.

The host code for all kernels is the same. It allocates and initializes the matrices, launches the kernels and checks for the correctness of the output.

Unlike the rest of the benchmarks, the measured time only includes actual kernel time, excluding setup and host-device memory copies as they would dominate the measurements.

3.1.1.1 Copy

In this kernel, each thread is responsible for copying four elements from the input matrix to the output matrix.

Each thread reads from the input matrix and writes to the output matrix in the exact same position, ensuring coalesced accesses to the global memory.

Given its simplicity and regularity, this kernel achieves the maximum prectical throughput, and has therefore been chosen as the main comparison point.

3.1.1.2 Naive transpose

The naive transpose is also a simple kernel. It is similar to the copy kernel, and simply swaps the output matrix indices.

From Figure 3.2 it is easy to see that such approach will not produce coalesced accesses to the global memory: threads perform coalesced reads but *strided* writes.

3.1.1.3 Coalesced transpose

This implementation makes use of the shared memory to avoid the strided global memory accesses. The basic idea is to decompose the transposition in two steps:

1. *Read global memory in a coalesced way and store it in the shared memory*: Thread (i, j) reads $M_{i,j}$ from the global memory and stores it in position (i, j) of the shared memory.

2. *Read from the shared memory (already in transposed order) and write to the global memory in a coalesced way*: Thread (i, j) reads position (j, i) from the shared memory and stores it in $M_{i,j}$ in the global memory.

The reason why this approach works is that there is no penality for non-aligned accesses to the shared memory. This makes it a perfect intermediate buffer to ensure global memory aligned accesses.

3.1.1.4 Bank conflict free transpose

The bank conflict free transpose is even more interesting, even though it translates into a very small change to the coalesced transpose source code.

Shared memory is implemented as a series of memory modules, named *banks*. Accessing memory in different banks can be done in parallel, but concurrent accesses to the same bank must be serialized.

Banks are organized in such a way that consecutive words are assigned to consecutive banks. Bank conflicts occur because in the *coalesced transpose* the size of the shared memory cache is a multiple of the size of a warp, and because of the scheme of accesses.

While this problem may seem complicated, the solution is surprisingly simple: instead of allocating shared memory caches of 32x32, we allocate caches of *32x33*. The extra column is never used by the program, but the padding causes the accesses to be distributed in a better way across the banks.

3.1.2 Transpose in OpenACC

As seen in the previous section, all efforts to improve the performance of the transpose are based on the use of shared memory to improve the efficiency of accesses to the global memory.

OpenACC provides, however, a much more abstract programming model in which details such as shared memory utilization are out of reach.

Benchmarks copy and transpose feature a few implementations, ranging from the most naive one (the result of simply adding OpenACC directives) to more elaborate ones.

Rather than to optimize the use of memory, the main focus of optimizations in OpenACC is the tuning the amount of work per thread.

Similarly to CUDA, all OpenACC benchmarks define TILE_DIM as 32 and BLOCK_ROWS as 8.

3.1.2.1 Copy

The initial implementation is shown in Listing 3.1 and is the result of adding the `#pragma acc kernels loop` directive to the copy.

```
#pragma acc kernels loop independent
  for (int i = 0; i < n*n; i++) {
    out[i] = in[i];
  }
```

Listing 3.1: Simple OpenACC copy.

The first modification to this kernel is to make it 2D. This can be seen in Listing 3.2. Moreover, this second implementation uses the idea of forcing the compiler to generate code that does more work per thread.

The kernel in Listing 3.2 launches a *grid* of $n/TILE\_DIM$ by $n/TILE\_DIM$ blocks (as specified by the two *gang* parameters). Each block, however, has dimensions $TILE\_DIM$ (32) by $BLOCK\_ROWS$ (8). Therefore, in order to cover the whole matrix each thread will have to operate over 4 elements, exactly like in the CUDA version.

```
#pragma acc kernels loop independent gang(n/TILE_DIM) vector(↩
    TILE_DIM)
  for (int i = 0; i < n; i++) {
#pragma acc loop independent gang(n/TILE_DIM) vector(BLOCK_ROWS)
    for (int j = 0; j < n; j++) {
      out[j + i * n] = in[j + i * n];
    }
  }
```

Listing 3.2: 2D OpenACC copy.

The last copy kernel family attempts to "manually" force each thread to operate over more elements. This is done by adding an extra for loop inside each thread, making it iterate over $n/BLOCK\_ROWS$ elements. The code can be seen in Listing 3.3.

```
#pragma acc kernels loop independent vector(TILE_DIM)
  for (int j = 0; j < n; j++) {
#pragma acc loop independent vector(BLOCK_ROWS)
    for (int i = 0; i < BLOCK_ROWS; i++) {

      for (int k = 0; k < n; k+=BLOCK_ROWS) {
        out[j + (i + k) * n] = in[j + (i + k) * n];
      }

    }
  }
```

Listing 3.3: OpenACC copy with extra inner loop.

3.1.2.2 *Transpose*

The OpenACC transpose kernels follow the same ideas introduced in the *copy* benchmark. The first implementation can be seen in Listing 3.4 and is the result of the simple addition of OpenACC directives to a transpose code.

```
#pragma acc kernels loop independent
  for (int j = 0; j < n; j++) {
#pragma acc loop independent
    for (int i = 0; i < n; i++) {
      out[j + i * n] = in[i + j * n];
    }
  }
```

Listing 3.4: Naive OpenACC transpose.

Another version of Listing 3.4 uses the same idea of Listing 3.2 to force more work per thread.

Finally, Listing 3.5 uses an inner loop to force threads to do more work, similarly to Listing 3.3.

```
#pragma acc kernels loop independent vector(TILE_DIM)
  for (int j = 0; j < n; j++) {
#pragma acc loop independent vector(BLOCK_ROWS)
    for (int i = 0; i < BLOCK_ROWS; i++) {

      for (int k = 0; k < n; k+=BLOCK_ROWS) {
        out[j + (i + k) * n] = in[(i + k) + j * n];
      }
    }
  }
```

Listing 3.5: OpenACC transpose with extra inner loop.

3.1.2.3 *Attempts to use shared memory*

OpenACC does not offer support to direct management of shared memory, but it generates code that uses shared memory whenever it detects it. As shown in Chapter 4, none of the implementations above generated code that used the shared memory.

There were, however, unsuccessful attempts to force the compiler into using shared memory, from which we can highlight:

- Dividing the two main for loops into several small ones, and hoping that each one of them would be mapped to one block and make use of shared memory.
- Allocating another matrix (using the `create` directive) with same dimensions as the main matrix and using blocks of it as the blocks of the shared memory.

3.2 Dot Product

In mathematics, the *dot product* between two equal-lenghted vectors is defined as follows:

$$(x_1, x_2, \ldots, x_n) \cdot (y_1, y_2, \ldots, y_n) = x_1y_1 + x_2y_2 + \cdots + x_ny_n$$

In computing, the dot product can be seen as a *reduction* over the pairwise multiplication of the elements of two vectors.

This benchmark was chosen to evaluate whether OpenACC is able to automatically generate code for a reduction, and if so, how performant it is.

3.2.1 Dot Product in CUDA

CUDA is not the ideal platform to perform reductions. Reductions may not be considered massive data parallel problems, having the amount of computations (n) equal to the amount of data (n).

Nevertheless, reductions are interesting problems to analyze, and are often used as part of a larger problem. It usually better to use sub-optimal algorithms in the GPU than to transfer data back and forth from the host.

The CUDA dot product algorithm analyzed in this study is presented in (SANDERS; KANDROT, 2010), and consists of 4 main steps. Assuming vectors X, Y of size n, n threads are launched, organized in blocks of size B:

1. Each block declares a shared-memory cache of size B (one “slot” for each thread in the block).
2. Each thread i computes $X[i] * Y[i]$ and stores it in its slot of the cache. At this point the two vectors have been pairwise multiplied and the result is stored in shared memory.
3. Each block performs a sum reduction on its elements. This is done in $O(\log B)$ steps, as seen in Figure 3.3.
4. In the CPU, the partial results are summed.

In practice, instead of launching one thread per element, a much smaller number is launched, causing each thread to operate on multiple elements. Therefore, in step 2 instead of computing one product, each thread computes several products and stores their sum in the cache.

Although all accesses to the global memory are made in a coalesced way, resources utilization is not optimal in step 3.

3.2.2 Dot Product in OpenACC

The OpenACC Dot Product is a straightforward implementation as shown in Listing 3.6.

```
  ff res = 0;
#pragma acc kernels loop
  for (int i = 0; i < n; i++) {
    res += v1[i] * v2[i];
  }
```

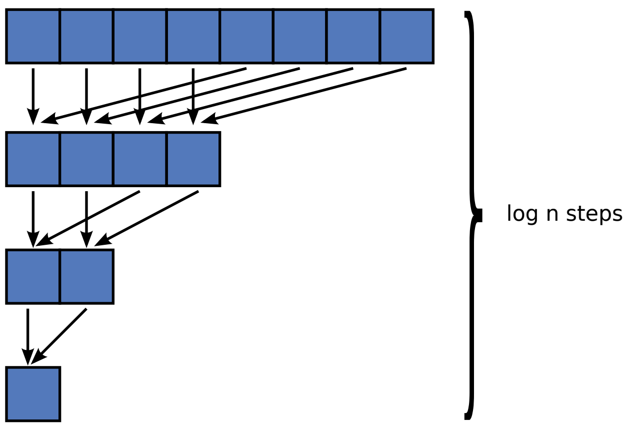


Figure 3.3: Parallel reduction inside each block.

```
    return res;
```

Listing 3.6: Dot product in OpenACC.

The interesting part of this implementation is that the compiler is actually able to detect the reduction and generate specific code for it. The developer is informed a reduction has been generated by the compiler output.

3.3 Matrix Multiplication (sgemm)

In mathematics, matrix multiplication is a binary operation that takes a pair of matrices and produces another matrix.

This study refers to the matrix multiplication as *sgemm* (Single-precision General Matrix Multiplication). All matrices used are square, and for the sake of simplicity matrices are referred as A, B and C, where:

$$C = A * B$$

On the computation of a matrix multiplication, the element (i, j) in the resulting matrix is the result of the dot product of the ith line of the first matrix with the jth column of the second matrix.

Matrix multiplication is a classic example of a parallel benchmark, featuring an interesting scheme with n^2 *independent* operations of cost n.

3.3.1 sgemm in CUDA

The CUDA implementations are described in (KIRK; HWU, 2010).

Following the idea of the matrix transpose benchmark, there are three matrix multiplication implementations: *naive*, *shared-memory* and *cuBLAS*.

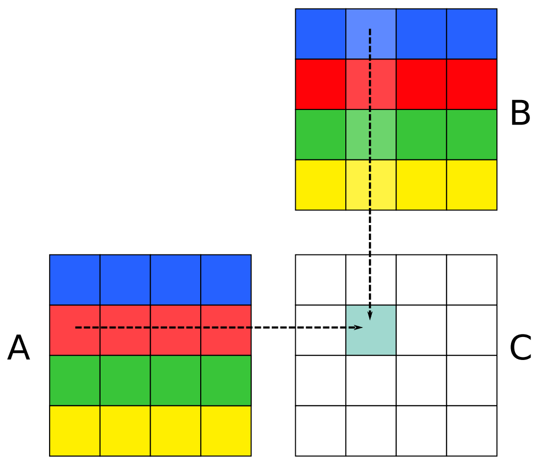


Figure 3.4: Matrix multiplication access patterns.

3.3.1.1 Naive implementation

In the naive implementation, n^2 threads are launched, each thread (i, j) being responsible for computing its respective element in C (thread (i, j) computes element (i, j)). Each one of the n^2 threads performs n operations, as shown in Figure 3.4.

3.3.1.2 Shared-memory implementation

Much like the naive matrix transpose algorithm, the multiplication described above generates non aligned accesses, as can be seen in Figure 3.4.

Moreover, many accesses are redundant. When computing the ith row of C, each thread loads different columns from B, but all threads read the exact same row from A.

Once again, *shared memory* is used to cache global memory accesses and increase performance.

3.3.1.3 cuBLAS

cuBLAS is the BLAS library for CUDA developed by NVIDIA. The implementation of the matrix multiplication is highly optimized. cuBLAS is closed-sourced, and no implementation details are provided.

In terms of programmability, using cuBLAS is as simple as using a regular library.

3.3.2 sgemm in OpenACC

Listing 3.7 shows the first version of the OpenACC sgemm, which results of the simple addition of the OpenACC directives.

It is interesting to notice that, in this code, the intermediate sum of the products have been stored on a local variable (`sum`) rather than on the final vector (`c`). This not only avoids several writes to the global memory, but also ensures that the loops are fully parallelizable.

```
#pragma acc loop gang vector(32) independent
  for (int j = 0; j < n; ++j) {
```

```
#pragma acc loop gang vector(32) independent
    for (int i = 0; i < n; ++i) {
      float sum = 0;
      for (int k = 0; k < n; ++k) {
        sum += a[j + n * k] * b[k + n * i];
      }
      c[i + n * j] = sum;
    }
  }
```

Listing 3.7: Matrix multiplication with OpenACC.

Two other similar implementations were made: one in which the *gang* parameter is tuned to force more work per thread (as in Listing 3.2) and another where the inner loop is manually unrolled into blocks of 32.

Typical CPU sgemm implementations swap the indices of the three loops to maximize *cache-hits* when iterating over the matrices. This is not a good idea in OpenACC implementation, since matrices A and B are not really being "iterated over". In the OpenACC solution, a bi-dimensional grid of threads is being launched to cover each one of the cells in C. Swapping the order of the indices would cause each thread to be responsible for computing one multiplication of each of the elements in a row/column, rather than an entire element of the C matrix.

3.3.2.1 Attempts to use shared memory

Like in the transposition, there were attempts to make the compiler use shared memory as caching, mainly trying to decompose the multiplication in small blocks. As in the transposition, such attempts did not generate shared memory usage.

3.4 Wrap Up

This chapter introduced the benchmarks proposed, as well as the several solutions studied. A common focus of optimization of the CUDA benchmarks was, as previously mentioned in Section 2.1, the use of shared memory to cache memory accesses and to ensure global memory efficiency. OpenACC optimizations, on the other hand, focused on tuning the amount of work per thread.

4 EXPERIMENTAL RESULTS

This chapter describes the results from each of the benchmarks introduced in Chapter 3.

4.1 Metrics

The following metrics were used to evaluate the benchmarks:

Throughput Throughput is defined as the amount of processed (copied/transposed) memory in a time interval, and is measured in Gb/s. The throughput T can be calculated with this formula:

$$T = \frac{B}{10^9 * time}$$

Where B is the amount of data in bytes read and written by the kernel and $time$ is the execution time in seconds.

Speedup we will define *speedup* as the execution time of the sequential algorithm running on the CPU divided by the execution time of the parallel algorithm running on the GPU. The speedup S is defined by the following formula:

$$S = \frac{T_{CPU}}{T_{GPU}}$$

Tests of the *Matrix multiplication* and *Dot product* benchmarks were run 10 times. *Matrix transpose* tests were run 100 times. The average of all executions was measured. These numbers were sufficient to get reliable results with negligible standard deviations.

4.2 Machine used

All benchmarks were executed on a *GeForce GTX 480*, with compute capability 2.0, 480 cores and 1.5 GB of total global memory. The CUDA driver version was 5.0. The compiler used was the *PGI Accelerator Compiler 12.9*.

The CPU is an Intel® Core™ i7 930, with 4 cores at 2.80GHz, 8MB of cache and 12 GB of total RAM memory.

	Registers	**Shared Mem.**	**Constant Mem.**	**Thread block size**	**Grid size**
acc_1d_naive	12	0	60	128	*unknown*
acc_1d_par	12	0	60	128	*
acc_2d_inner	22	0	64	32x8	*unknown*
acc_2d_naive	14	0	64	64x4	*unknown*
acc_2d_par	14	0	64	32x8	*

Table 4.1: Compilation results for copy.

4.3 Matrix Transpose

This section presents the results for the matrix transpose implementations.

4.3.1 Theoretical maximum throughput

Measuring the theoretical maximum throughput of the used GPU is important in order to have a main point of comparison for the copy/transpose benchmarks.

With a memory clock rate of 1848 MHz ($1848 * 10^6$Hz) and a memory bus width of 384 bits (48 bytes), the theoretical maximum throughput achievable by the GTX 480:

$$(1848 * 10^6)Hz * (48 * 2)B = 1774086\ Bytes/s = 177.4\ GB/s$$

In practice, other factors such as kernel initialization and scheduling overhead make it impossible to obtain this throughput.

4.3.2 Copy

Recalling the *copy* implementations:

acc_1d_naive simple 1D copy with an OpenACC *loop* pragma.

acc_1d_par 1D copy with parameter tuning (threads per block times number of blocks is smaller than the size of the matrix).

acc_2d_inner 2D copy with inner loop.

acc_2d_naive simple 2D copy with just the *loop* pragma.

acc_2d_par 2D copy with parameter tuning.

Table 4.1 shows the compilation results for the copy OpenACC implementations. Notice that the grid size is either *unknown* - in which case the compiler is free to choose it upon launch (typically chosing a scheme such that each thread works on one piece of the problem) - or is a configuration that, upon launch, will respect the work ratio specified before (noted as “$*$” on the table).

Figure 4.1 shows the performance of all the copy benchmarks. As expected, not even CUDA is able to reach the maximum theoretical throughput, approaching on average *0.75%* of the theoretical 177.4 GB/s.

From Figure 4.1, the best performing version is *acc_2d_inner*. The 1D approaches (*acc_1d_naive* and *acc_1d_par*) are very inneficient for small instances, but become as efficient as *acc_2d_inner* with matrices as large as 10240x10240.

The intriguing result is the relation between *acc_2d_inner* and *acc_2d_par*. One would expect the two of these to have similar performances, since they should perform the same operations. The parameter tuned version of copy is, however, significantly slower than its “manually” tuned counterpart.

It is also possible to see that for smaller matrices (for which the processing time is less significant) the startup time for OpenACC is much higher than for CUDA. Overall this is also true, as can be seen that OpenACC’s running time tends to differ by a constant offset from CUDA’s. The higher launch overhead is understandable, and is unlikely to represent a big problem on real tests.

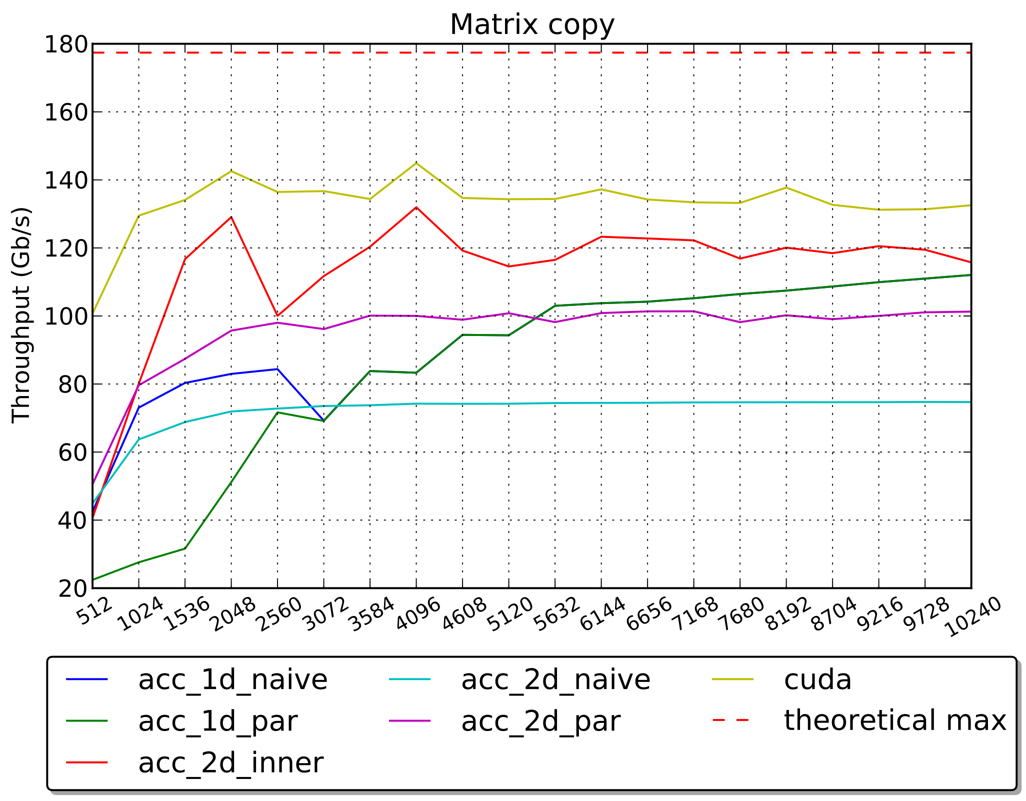


Figure 4.1: Matrix copy results.

4.3.3 Transpose

Recalling the *transpose* implementations from Chapter 3:

acc_naive simple kernel with no tuning.

acc_inner_loop kernel with inner loop

acc_par_tuning kernel with parameter tuning.

	Registers	**Shared Mem.**	**Constant Mem.**	**Thread block size**	**Grid size**
acc_naive	17	0	64	64x4	*unknown*
acc_inner_loop	22	0	64	32x8	*unknown*
acc_par_tuning	27	0	64	32x8	*

Table 4.2: Compilation results for transpose.

	% theoretical maximum	**% CUDA copy**
CUDA copy	76.0	100.0
CUDA transpose	66.7	87.7
OpenACC copy	66.7	87.7
OpenACC transpose	45.6	59.9

Table 4.3: Percentage of the theoretical and pratical optimal solutions.

cuda_naive naive CUDA.

cuda_coal coalesced CUDA.

cuda_opt bank conflict free CUDA.

The performance of the matrix transposes can be seen in Figure 4.2. The results of the compilation can be seen in Table 4.2.

It is interesting that, for this set of benchmarks, the best performance from OpenACC is observed with the naive implementation with parameter tuning (*acc_par_tuning*), rather than the one with an inner loop (*acc_inner_loop*). This contrasts with the results obtained from the copy kernel and gives us an impression that getting the most performing OpenACC implementation may require some experimentation. It was not possible to further investigate why these results were produced, since none of the codes generated in the experiments are human-readable.

Another interesting fact is that an OpenACC transpose (*acc_par_tuning*) can be faster than a CUDA coalesced transpose. In fact, it is in average 18% faster, even though it does not use shared memory. This pattern (OpenACC faster than naive CUDA) is found in the next benchmarks as well, and can be explained by the fact that the Fermi cache amortizes the cost of global memory accesses. Moreover, unlike the CUDA implementations, OpenACC makes use of constant memory.

4.3.4 Summary

Figure 4.3 shows the aggregated results for both copy and transpose benchmarks. Table 4.3 shows the percentual difference from each one of the best implementations to the theoretical maximum and to the pratical maximum. In this measurement matrices smaller than 2048x2048 were discarded in order to exclude the initialization overhead.

4.4 Dot Product

The dot product benchmark is interesting for its comparison with CUDA, given that both implementations perform significantly worse than the serial version, as seen in Fig-

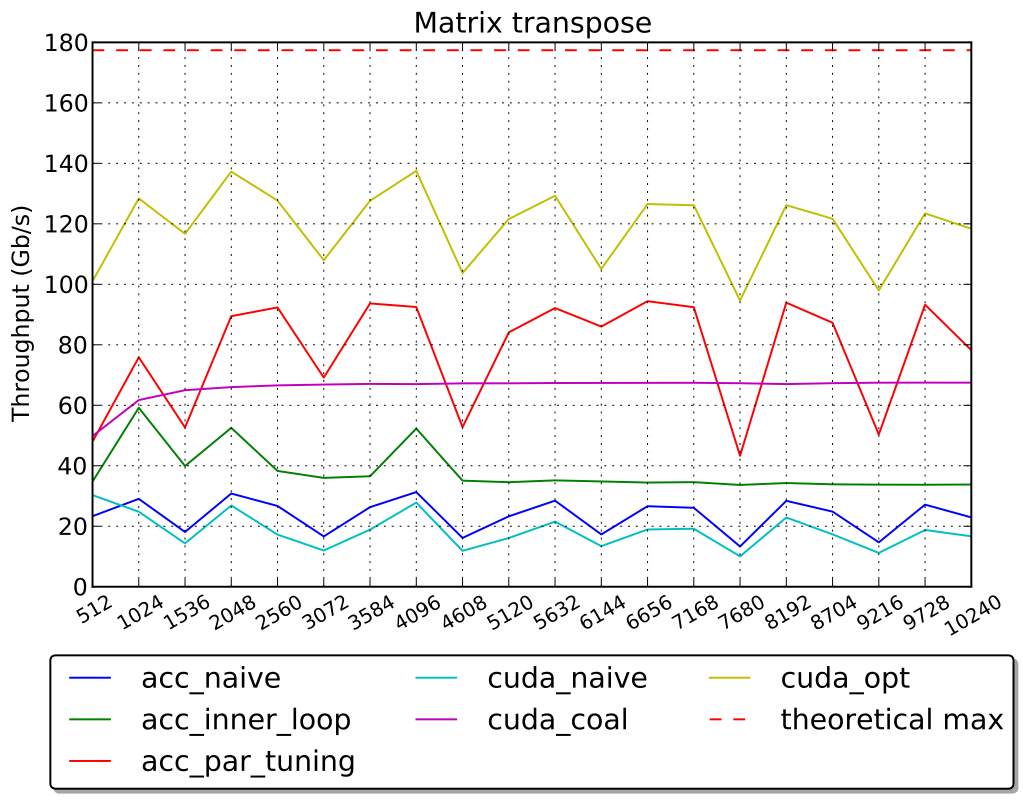


Figure 4.2: Matrix transpose results.

ure 4.4.

The interesting detail is that both implementations have almost the same results, what shows us that OpenACC actually generated CUDA-equivalent code. Given the relative complexity of the CUDA code involved, this is quite impressive.

4.5 Matrix Multiplication (sgemm)

Recalling matrix multiplication implementations:

acc the naive OpenACC implementation with blocks of 32x32.

acc_par each thread operates on 4 cells of the result matrix.

acc_loop features a manually unrolled loop over 32 elements.

cuda_naive simple naive implementation with CUDA.

cuda_opt shared memory aware implementation.

cublas cuBLAS sgemm.

Matrix multiplication results are shown in Figure 4.5.

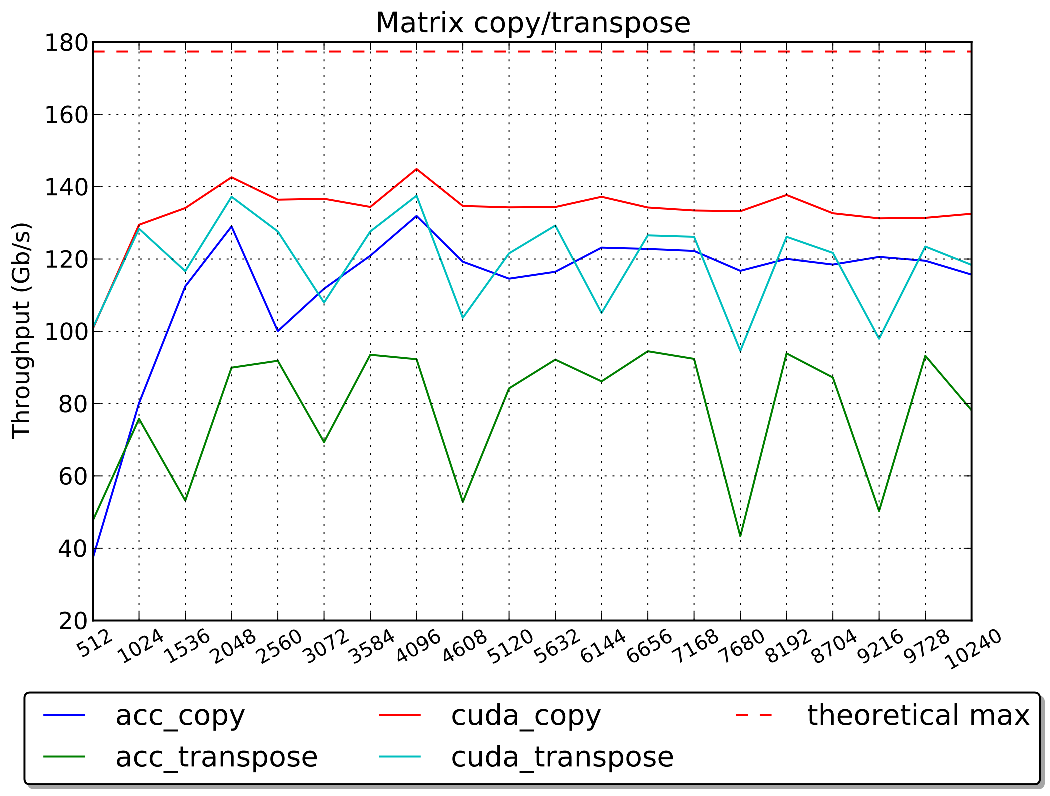


Figure 4.3: Comparison of the best OpenACC/CUDA copies and transposes.

	Registers	**Shared Mem.**	**Constant Mem.**	**Thread block size**	**Grid size**
acc	24	0	80	128	*unknown*
acc_par	22	0	80	32x8	*
acc_loop	63	0	80	128	*unknown*

Table 4.4: Compilation results for sgemm.

It is possible to see that all OpenACC implementations have the peformance, being relatively better than the naive CUDA and worse than the optimized CUDA.

OpenACC implementations are better than naive CUDA. This shows that if one is not familiar with CUDA, or is not willing to optimize his code, OpenACC might be a better option.

The cuBLAS implementation shows the maximum performance achievable, which is significantly faster than any other implementation. This kind of performance can only be achieved with thorough optimizations and fine-grained tuning. As seen in the previous sections, this is far from possible with OpenACC.

From a developers standpoint, the main advantage of a GPU solution is the comparison with a serial solution. Even though OpenACC shows a poor results when compared to a thoughtful CUDA implementation, the speedup relative to the serial implementation is around 45x for matrices of size 2048x2048, as shown in Table 4.5.

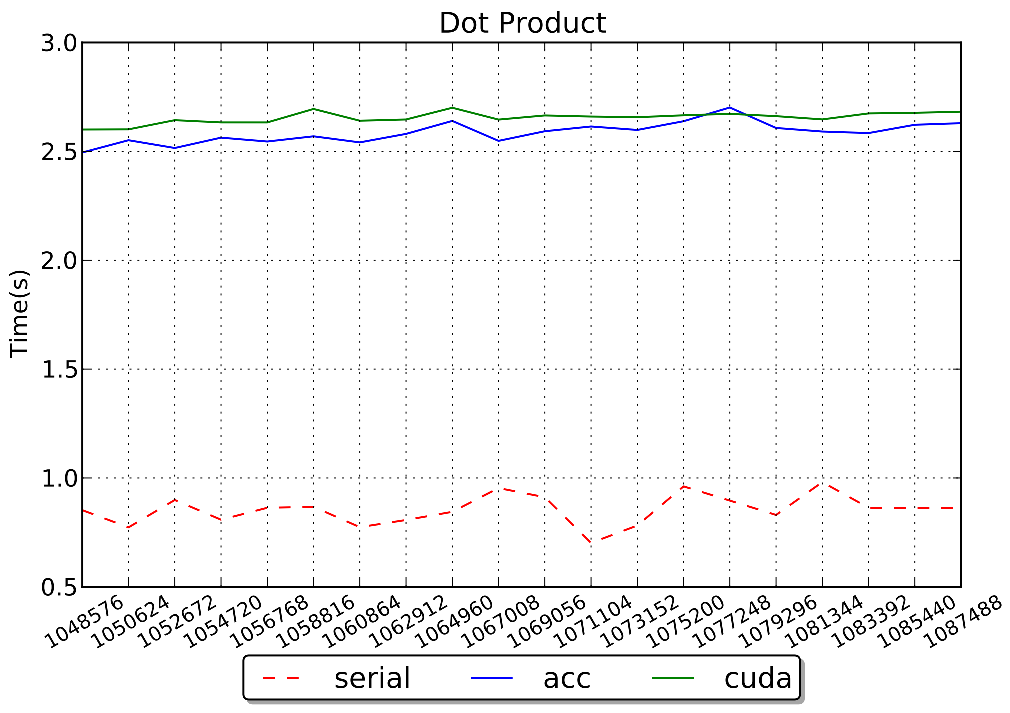


Figure 4.4: Dot product results.

	1024x1024	**2048x2048**	**4096x4096**
acc	36.5	45.2	45.9
cuda_naive	26.4	29.3	29.4
cuda_opt	85.9	98.8	100.4
cublas	143.0	242.4	301.4

Table 4.5: GPU speedups for sgemm.

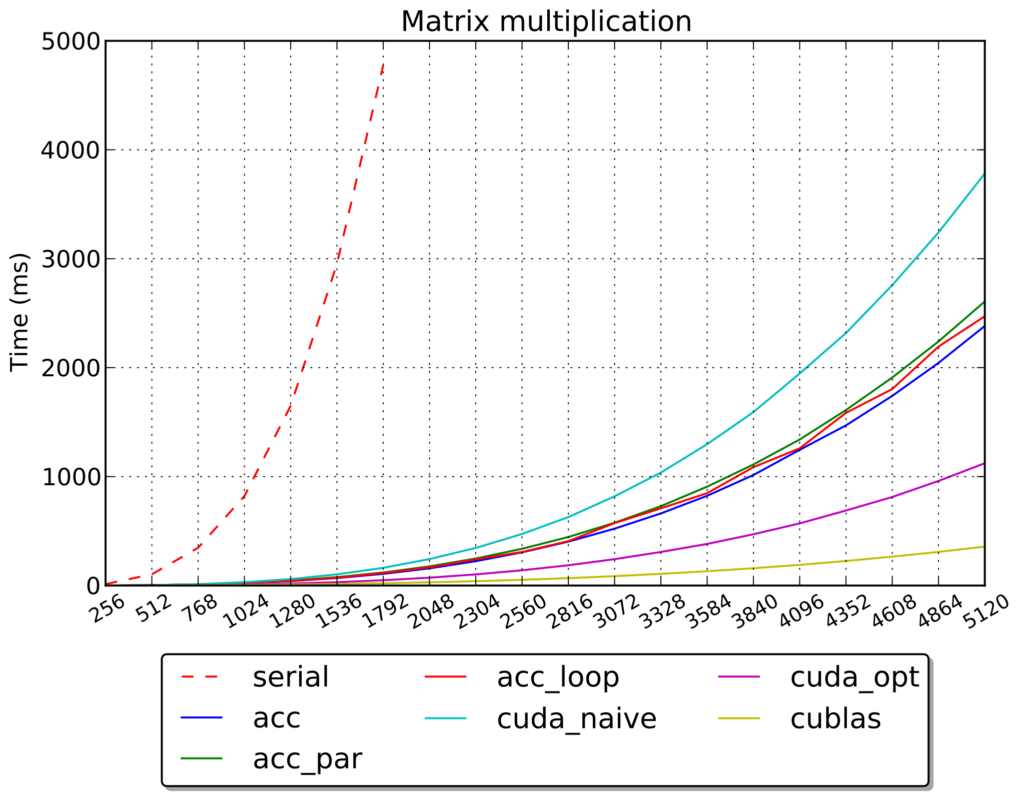


Figure 4.5: Matrix multiplication results.

5 CONCLUSION

This work experimentaly evaluated OpenACC and compared it to CUDA using three main benchmarks, each one comprising several implementations.

This chapter summarizes the results and tries to answer the initial question: *how much of CUDA's performance can be matched by OpenACC, and what exactly are the advantages in terms of ease of programming?*

Programmability/Maintainability OpenACC provides a *much* simpler programming paradigm than CUDA.

Also, although OpenACC and OpenMP share many common principles, OpenACC still targets a completely different architecture, which must be taken into account in order to attain good results.

That said, OpenACC provides a great way to accelerating legacy code, as the modifications required are quite small and the parallelization would not be made otherwise. It also provides a starting point for programmers with little GPGPU experience, as they can profit from some of the performance of GPUs without sacrificing productivity.

Overall, OpenACC is very maintainable and easy to develop. Code produced is portable across different platforms.

From a performance standpoint, however, relying on existing libraries such as *cuBLAS* (used in the matrix multiplication benchmark) or the ones mentioned in Chapter 2 may provide much better results with virtually the same added complexity.

Performance OpenACC's higher level model comes at the cost of performance and expressivity. OpenACC can provide a lot of improvement over serial code, but results are still far from what is observed in handwritten CUDA.

The main disadvantage arises from the significant loss of control over the generated code. This is particularly impactful when it comes to managing the use of the different memories, a key point for GPU programming optimization.

Performance is, however, good enough for OpenACC to be considered against a naive or non-optimized CUDA implementation.

Expressivity OpenACC represent a much more rigid and limited model than CUDA, only allowing regular `for` loops to be parallelized.

Moreover, OpenACC does not provide other interesting features, such as *atomics*, which are available in CUDA.

Overall, OpenACC's abstract model allows a smaller set of problems to be parallelized efficiently.

5.1 Future of GPUs and OpenACC

Right now, achieving a good speedup with CUDA depends deeply on fine-grained optimizations, but this picture is gradually changing with the evolution of GPUs.

The introduction of the Fermi architecture in 2010 brougth transparent caching to GPUs and to CUDA, making it a much friendlier platform. Furthermore, at each new CUDA device architecture, the constraints for memory efficiency are relaxed and GPUs become overall more flexible.

If it gets easier for programmers to write efficient code, the same holds true for compilers. Future graphic cards, together with newer versions of the OpenACC standard and improved compilers are likely to reduce the gap between OpenACC and CUDA.

Handwritten CUDA performance, however, is unlikely to be matched anytime soon, and OpenACC will probably become a tool such as OpenMP, in which the simplicity comes at the expense of performance.

REFERENCES

CHAPMAN, B.; JOST, G.; VAN DER PAS, R. **Using OpenMP**: portable shared memory parallel programming. [S.l.]: MIT press, 2007. v.10.

GROUP, T. K. **The Khronos Group Releases OpenCL 1.0 Specification**. Available from: http://www.khronos.org/news/press/the_khronos_group_releases_opencl_1.0_specification Accessed in Dec. 2012.

GROUP, T. P. **PGI Accelerator Compilers**. Available from: www.pgroup.com/resources/accel.htm Accessed in Dec. 2012.

HWU, W. **GPU Computing Gems Jade Edition**. [S.l.]: Elsevier Science, 2011. (Applications of GPU Computing Series).

KIRK, D. B.; HWU, W.-m. W. **Programming Massively Parallel Processors**: a hands-on approach. 1st.ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

NVIDIA. **Fermi Compute Architecture Whitepaper**. [S.l.: s.n.], 2011.

NVIDIA. **NVIDIA CUDA Programming Guide 5.0**. [S.l.: s.n.], 2012.

NVIDIA, C. Cublas Library. **NVIDIA Corporation, Santa Clara, California**, [S.l.], v.15, 2008.

NVIDIA, C. **CUFFT library**. [S.l.]: Version, 2010.

OPENACC. **NVIDIA, Cray, PGI, CAPS Unveil 'OpenACC' Programming Standard for Parallel Computing**. Available from: http://www.openacc.org/node/93 Accessed in Dec. 2012.

OPENACC. **The OpenACC Application Programming Interface 1.0**. [S.l.: s.n.], 2011.

RUETSCH, G.; MICIKEVICIUS, P. Optimizing matrix transpose in CUDA. **NVIDIA CUDA SDK Application Note**, [S.l.], 2009.

SANDERS, J.; KANDROT, E. **CUDA by Example**: an introduction to general-purpose gpu programming. 1st.ed. [S.l.]: Addison-Wesley Professional, 2010.

TOP500. **Top500 List**. Avaliable from: http://www.top500.org/list/2012/11 Accessed in Dec. 2012.

APPENDIX A COMPILING AND PROFILING OPENACC

Compiling OpenACC code using the PGI Accelerator compiler can be done with the following command:

```
pgcc -acc -o <binary name> <sources>
```

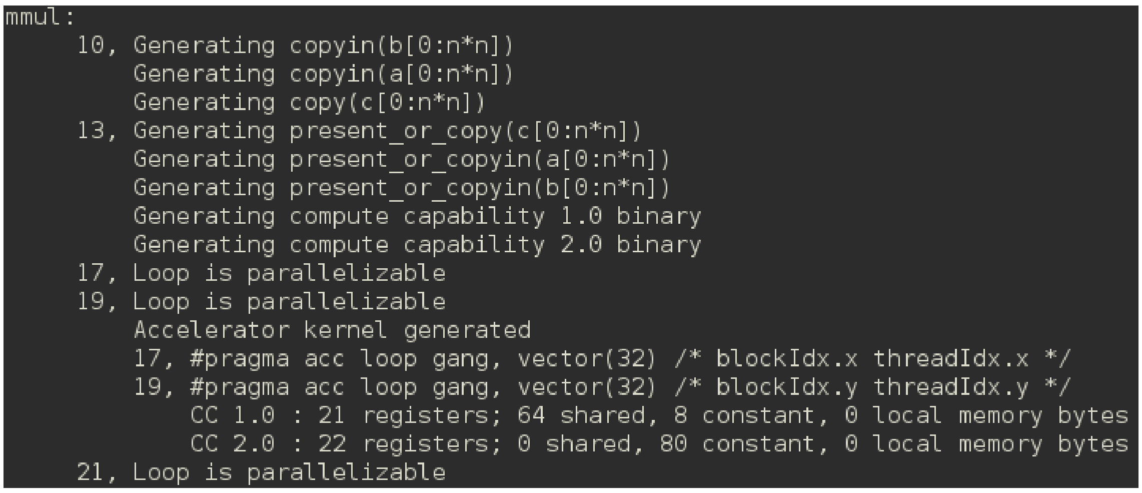
Moreover, the compiler can display useful information about the compilation when the option *-Minfo=acc* is added. Figure A.1 shows the compiling output from a simple kernel.

The most important information to be extracted from the compilation output is the usage of *registers*, *shared memory* and *constant memory*, as well as the kernel launch settings (number of threads per block, number of blocks...). Moreover, the compiler output shows information about *memory transfers*, which as mentioned in 2.3.4 is a fundamental part of the OpenACC parallelization.

The compiler will also display here information about issues with parallelization, such as inter-loop dependencies.

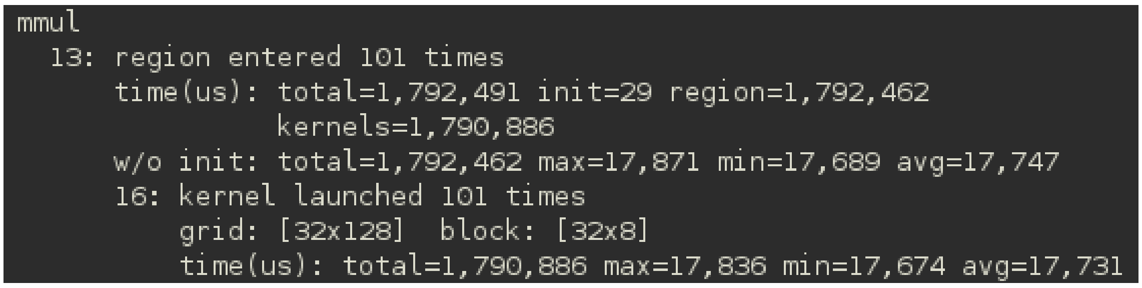
Enabling the profiler in OpenACC can be done either by adding the option `-Minfo=time` to the `pgcc` compiler.

Additional options can enable features such as storing the resulting CUDA bytecode (*PTX*) or even the generated CUDA code (which is not meant to be human-unreadable



```
mmul:
     10, Generating copyin(b[0:n*n])
         Generating copyin(a[0:n*n])
         Generating copy(c[0:n*n])
     13, Generating present_or_copy(c[0:n*n])
         Generating present_or_copyin(a[0:n*n])
         Generating present_or_copyin(b[0:n*n])
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
     17, Loop is parallelizable
     19, Loop is parallelizable
         Accelerator kernel generated
         17, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
         19, #pragma acc loop gang, vector(32) /* blockIdx.y threadIdx.y */
             CC 1.0 : 21 registers; 64 shared, 8 constant, 0 local memory bytes
             CC 2.0 : 22 registers; 0 shared, 80 constant, 0 local memory bytes
     21, Loop is parallelizable
```

Figure A.1: PGCC compiler output showing information about the generated GPU code.



```
mmul
  13: region entered 101 times
      time(us): total=1,792,491 init=29 region=1,792,462
                kernels=1,790,886
      w/o init: total=1,792,462 max=17,871 min=17,689 avg=17,747
      16: kernel launched 101 times
          grid: [32x128]  block: [32x8]
          time(us): total=1,790,886 max=17,836 min=17,674 avg=17,731
```

Figure A.2: OpenACC profiler output.

and is not very useful in practice).

Profiling information can be seen in Figure A.2, and includes information about host-device memory transfers, initialization and kernel execution times. This information is particularly useful to detect regions where memory is being unnecessarily transfered and to visualize in which parts of the program time is being spent.