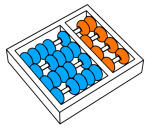
Universidade Estadual de Campinas
Instituto de Computação



Eder Maicol Gomez Zegarra

Support for Parallel Scan in OpenMP

Suporte de Parallel Scan em OpenMP

CAMPINAS
2018

Eder Maicol Gomez Zegarra

Support for Parallel Scan in OpenMP

Suporte de Parallel Scan em OpenMP

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

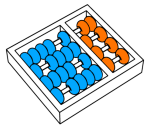
Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Guido Costa Souza de Araujo
Co-supervisor/Coorientador: Dr. Marcio Machado Pereira

Este exemplar corresponde à versão final da Dissertação defendida por Eder Maicol Gomez Zegarra e orientada pelo Prof. Dr. Guido Costa Souza de Araujo.

CAMPINAS
2018

Universidade Estadual de Campinas
Instituto de Computação



Eder Maicol Gomez Zegarra

Support for Parallel Scan in OpenMP

Suporte de Parallel Scan em OpenMP

Banca Examinadora:

- Prof. Dr. Guido Costa Souza de Araujo (Supervisor/*Orientador*)
  IC/UNICAMP
- Prof. Dr. Renato Antônio Celso Ferreira
  Universidade Federal de Minas Gerais (UFMG)
- Prof. Dr. Guilherme Pimentel Telles
  Institute of Computing - UNICAMP

A ata da defesa com as respectivas assinaturas dos membros da banca encontra-se no processo de vida acadêmica do aluno.

Campinas, 24 de abril de 2018

Acknowledgements

First of all, I want to thank God, my family for their constant and unconditional support. To my adviser Prof. Guido and co-adviser Prof. Marcio for the continuous help and advice during these two years to be able to finish this work. To the panel to accept the invitation and for their time provided.

Along over this two years, I found the AClang project that allowed me to add my work to its structure, thanks to AClang project's creators.
Thanks to CAPES for supporting in all this time (Scholarship number : 1590935).

Resumo

Prefix Scan (ou simplesmente scan) é um operador que computa todas as somas parciais de um vetor. A operação scan retorna um vetor onde cada elemento é a soma de todos os elementos precedentes até a posição correspondente. Scan é uma operação fundamental para muitos problemas relevantes, tais como: algoritmos de ordenação, análise léxica, comparação de cadeias de caracteres, filtragem de imagens, dentre outros. Embora existam bibliotecas que fornecem versões paralelizadas de scan em CUDA e OpenCL, não existe uma implementação paralela do operador scan em OpenMP. Este trabalho propõe uma nova clausula que permite o uso automático do scan paralelo. Ao usar a cláusula proposta, um programador pode reduzir consideravelmente a complexidade dos algoritmos, permitindo que ele concentre a atenção no problema, e não em aprender novos modelos de programação paralela ou linguagens de programação. Scan foi projetado em ACLang (`www.aclang.org`), um framework de código aberto baseado no compilador LLVM/Clang, que recentemente implementou o *OpenMP 4.X Accelerator Programming Model*. AClang converte regiões do programa de OpenMP 4.X para OpenCL. Experimentos com um conjunto de algoritmos baseados em Scan foram executados nas GPUs da NVIDIA, Intel e ARM, e mostraram que o desempenho da clausula proposta é equivalente ao alcançado pela biblioteca de OpenCL, mas com a vantagem de uma menor complexidade para escrever o código.

Abstract

Prefix Scan (or simply scan) is an operator that computes all the partial sums of a vector. A scan operation results in a vector where each element is the sum of the preceding elements in the original vector up to the corresponding position. Scan is a key operation in many relevant problems like sorting, lexical analysis, string comparison, image filtering among others. Although there are libraries that provide hand-parallelized implementations of the scan in CUDA and OpenCL, no automatic parallelization solution exists for this operator in OpenMP. This work proposes a new clause to OpenMP which enables the automatic synthesis of the parallel scan. By using the proposed clause a programmer can considerably reduce the complexity of designing scan based algorithms, thus allowing he/she to focus the attention on the problem and not on learning new parallel programming models or languages. Scan was designed in AClang (`www.aclang.org`), an open-source LLVM/Clang compiler framework that implements the recently released OpenMP 4.X Accelerator Programming Model. AClang automatically converts OpenMP 4.X annotated program regions to OpenCL. Experiments running a set of typical scan based algorithms on NVIDIA, Intel, and ARM GPUs reveal that the performance of the proposed OpenMP clause is equivalent to that achieved when using OpenCL library calls, with the advantage of a simpler programming complexity.

List of Figures

Contents

Chapter 1

Introduction

Parallelizing loops is a well-known research problem that has been extensively studied. The most common approach to this problem uses DOALL [27] algorithms to parallelize the iterations of loops which do not have loop-carried dependencies. Although there are approaches such as DOACROSS [15], DSWP [34] and BDX [16] that can be used to parallelize loop-carried dependent loops, these algorithms can not be directly applied to loops that are sequential in nature. One example of such loop can found in the implementation of the scan operation.

Cumulative sum, inclusive scan, or simply *scan* [9] is a key operation that has for goal to compute the partial sums of the elements of a vector. The scan operation results in a new vector where each element is the sum of the preceding elements of the input vector up to its corresponding position. Scan is a central operation in many relevant problems like sorting, lexical analysis, string comparison, image filtering, stream compaction, histogram construction as well as in many data structure transformations [10].

Scan is a very simple operation that can be generalized in two flavors (*inclusive* and *exclusive*) as follows. Given a binary associative operator $\oplus$ and a vector of n elements $x = [x_0, x_1, ..., x_{n-1}]$, an *inclusive scan* produces the vector $y = [x_0, x_0 \oplus x_1, ..., x_0 \oplus x_1 \oplus ... \oplus x_{n-1}]$.

$$\begin{aligned} y[0] &= 0 \\ y[1] &= x[0] \\ y[2] &= x[0] + x[1] \\ y[3] &= x[0] + x[1] + x[2] \\ &\dots \\ y[i] &= \sum_{j=0}^{i-1} x[j] \end{aligned} \tag{1.1}$$

Similarly, the *exclusive scan* operation results in vector $y = [I, x_0, x_0 \oplus x_1, ..., x_0 \oplus x_1 \oplus ... \oplus x_{n-2}]$, where I is the identity element in the binary associative operator $\oplus$. The parallel scan clause proposed in this work is based on the exclusive scan operation which will be called *scan* from now on. It is trivial to compute inclusive scan from the result of its exclusive version. This can be done by: (i) computing the exclusive scan of y; (ii)

Listing 1.1: The prefix sum implementation

```
1  (a) Sequential implementation
2
3  y[0] = 0;
4  for(int i = 1; i < n; i++)
5    y[i] = y[i-1] + x[i-1];
6
7  (b) Parallel implementation using the new clause
8
9  y[0] = 0;
10 #pragma omp parallel for scan(+: y)
11 for(int i = 1; i < n; i++)
12   y[i] = y[i-1] + x[i-1];
13
```

shifting the elements of y to the left; and (iii) storing the operation $y[n-2] \oplus x[n-1]$ into $y[n-1]$.

The scan of a sequence is trivial to compute using an $\mathcal{O}(n)$ algorithm that sequentially applies the recurrence formula $y[i] = y[i-1] \oplus x[i-1]$ to the n elements of x. For example, when the binary operator $\oplus$ is the addition (Equation 1.1), the scan operation stores in y all partial sums of array x, an algorithm named *Prefix Sum*. As shown in Listing 1.1a, the loop that implements prefix sum is intrinsically sequential due to the loop-carried dependence on y which makes the value of $y[i]$ depend on the value of $y[i-1]$ from the previous iteration. Hence, the loop body in Listing 1.1a forms a single *strongly connected component* in the program control-flow graph [7] and thus typical DOACROSS based algorithms like [40, 13] cannot be used to parallelize the loop iterations of prefix sum.

There are many other scan based operations that use various associative binary operators like the product, maximum, minimum, and logical `AND`, `OR`, and `XOR` to parallelize some very relevant algorithms [31, 14, 8]. Given the relevance of scan in computing, library-based parallel implementations of scan have been proposed in the past [19, 9] and designed as library calls in languages like OpenCL and CUDA [36, 12]. Unfortunately, most of these implementations are problem specific leaving the programmer with the task of mastering the complexity of OpenCL and CUDA to handle the design of a scan based operation to a specific problem.

This work proposes a new OpenMP `scan` clause that enables the automatic synthesis of parallel scan. The programmer can use the new clause to design algorithms in OpenMP C/C++ code thus eliminating the need to deal with the complexity of OpenCL or CUDA. The new scan clause was integrated into *ACLang*, an open-source LLVM/Clang compiler framework (www.aclang.org) that implements the recently released *OpenMP 4.X Accelerator Programming Model* [29]. AClang automatically converts OpenMP 4.X annotated program regions to OpenCL/SPIR kernels, including those regions containing the new scan clause.

A careful reader might think that such new scan clause is a trivial extension of the reduction clause already available in OpenMP. As a matter of fact, the reduction of the elements of a sequence x can be obtained by computing the scan of x into y as shown in Listing 1.1b and returning the value of $y[n-1] + x[n-1]$. In other words, reduction is a simpler version of scan in which the values of all intermediate partial sums are not exposed, and only the total sum of the elements of x is returned. Reduction can be performed in $\mathcal{O}(log\ n)$ complexity using a tree-based [6] or a butterfly-based [23] parallel

algorithm. Moreover, both reduction and scan are operations that handle loop-carried dependent variables. In the reduction case, a single variable accumulates the value from the previous iterations, while in scan the accumulation occurs for all elements of $y[i]$ each depending on elements from the previous iterations. This makes the implementation of parallel scan much harder than the implementation of reduction.

The rest of the work is organized as follows. Chapter 2 details some concepts of programming to GPU, and the AClang compiler. Chapter 3 describes the state-of-art of the scan algorithm used to design and implement the new OpenMP scan clause. Also, this Chapter gives an outline of the structure of the AClang compiler and describes the details of the implementation of the OpenMP scan clause into the AClang Compiler. Chapter 4 describes some examples of the use of the scan clause. Chapter 5 discusses related work, and Chapter 6 provides performance numbers and analyzes the results when programs are compiled with the new scan clause. Finally, Chapter 7 concludes the work.

The contribution of this work was published in the following conference paper [17].

- M. Gomez, M. Pereira, X. Martorell, and G. Araujo. *Automatic scan parallelization in openmp*. In 2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW), pages 85–90, Oct 2017.

Chapter 2

Background

2.1 Introduction to GPUs

2.1.1 A Brief History of GPUs

In the early 1990s, users began purchasing 2D display accelerators for their computers. These display accelerators offered hardware-assisted bitmap operations to assist in the display and usability of graphical operating systems.

Around the same time, the company Silicon Graphics popularized the use of three-dimensional graphics. In 1992, Silicon Graphics opened the programming interface for its hardware, launching the OpenGL library, so that it became a de facto standard.

By the mid-1990s, the demand for applications that were using 3D graphics increases considerably, growing one stage of development. PC gaming was affected by those devices creating progressively, more realistic 3D environments. At the same time, companies such as NVIDIA, ATI Technologies, and 3dfx Interactive, began releasing graphics accelerators that were affordable enough to attract widespread attention. These developments cemented 3D graphics as a technology that would become prominently for years to come.

In 1999 was created the first GPU GeForce 256 that was marketed as "the world's first GPU" or Graphic Processing Unit, enhancing the potential for even more visually exciting applications. Since transform and lighting were already integral parts of the OpenGL graphics pipeline, the GeForce 256 marked the beginning of a natural progression where increasingly more of the graphics pipeline would be implemented directly on the graphics processor.

In 2001 was introduced the GeForce 3, the first programmable GPU. For the first time, developers had some control over the exact computations that would be performed on their GPUs.

2.1.2 GPU Overview

Graphics processor units are designed to allow to handle massive computations, required to render the graphics that are created and displayed by a computer. Commonly, this requires the execution of the same operation on a large data set. In addition, this processing should be done in real time and must be completed as fast as can be done. The

GPU was the answer to this.

GPUs were designed for graphics processing in mind, and this was the main application for GPUs for a long time. Lately, programmers discovered an opportunity to explore GPUs to achieve high levels of parallelization for general-purpose application beyond the graphics domain. That is how started what is known as general purpose graphics processing (or GPGPU) programming. With the steady growth interest in GPGPU programming, GPU vendors started building GPU designs that were more flexible and had an open programming model. Hence, modern GPUs began been designed consisting of many programmable cores. These cores are capable of executing threads of computation, where each thread operates on a slice of a large input data set.

Therefore, over time the use of GPUs has been improved the performance of programs. The downside to using GPU is the fact that having a partner CPU is necessary to enable GPU execution. The GPU by itself can not be a standalone unit. To be able to operate on a GPU, the presence of the CPU is necessary to manage the execution of the program. The CPU is responsible for determining which portions of the application are completed by the GPU and defining which parameters will be used in this operation. Also, the CPU is responsible for the memory management of the data which is delivered and received from the GPU. Hence, for the operations that need to be performed in the GPU, the data must be copied from the CPU memory. Similarly, when the GPU finishes its work, it is necessary to pass the data from the GPU to the CPU. Such data transfer operations are typically very expensive and often limits the applications that can benefit from the GPU usage.

In addition to the above discussion, GPUs have several other disadvantages. Similarly to Digital Signal Processors (DSPs), a GPU has a slower clock rate than the CPU. It also does not have the same cache sizes. It does not implement branch prediction or any similar optimizations. For these reasons a GPU cannot keep up with the CPU in serial execution. Therefore, it is very important to define which portions of the program can be done serially and executed on the CPU, and which other portions could be parallelized and executed on the GPU. If the proportion of the code to be executed on the GPU compensates the disadvantages mentioned above and produces overall faster code than its serial version, it is worth to use the GPU to accelerate that fragment of the code.

With the goal of enabling general-purpose applications, GPU manufacturers started offering programming toolkits. In particular, NVIDIA designed a toolkit based on the CUDA language that allows programmers to create applications that can run on GPUs. A major advantage of CUDA is its similarity with the C language. This allows the implementation of many applications as well as the increase in the number of parallel algorithms that can harness the potential of GPUs. As was mentioned before, GPUs can execute a program in parallel.

By seeking to have a broader programming model that could span parallel execution for a range of accelerator devices, and not only CPU to GPUs, a large share of the industry proposed a new language called OpenCL. The OpenCL standard is the first open, royalty-free, unified programming model for accelerating algorithms on heterogeneous systems. OpenCL allows the use of a C-based programming language for developing code across different platforms, such as CPUs, GPUs, DSPs, and field-programmable

gate arrays (FPGAs). OpenCL is a programming model for software engineers and a methodology for system architects. It is based on standard ANSI C (C99) with extensions to extract parallelism. OpenCL also includes an application program interface (API) for the host to communicate, using a *kernel* code, with the hardware accelerator (mainly GPU), traditionally over PCI Express. It also allows one kernel to communicate with another without host interaction. In the OpenCL model, the user schedules *kernels* to command queues, of which there is at least one for each device. The OpenCL run-time then breaks the data-parallel tasks into pieces and sends them to the processing elements in the device. This is the method for a host to communicate with any hardware accelerator. It is up to the individual hardware accelerator vendors to abstract away the vendor-specific implementation. Summing up what was said before, OpenCL is a framework that allows the use of several devices from different vendors. Most developers agree that CUDA has a better performance than OpenCL in NVIDIA devices. However, not all the users have NVIDIA cards, and therefore OpenCL is the preferred choice instead of CUDA. Clearly if NVIDIA card is an option, CUDA will always be chosen. There has been a huge amount of research work on GPU architectures and code optimization. Chapter 5 discusses the most relevant research related to this dissertation. Nevertheless, it is important to highlight the importance of users understanding the process of evaluating if it is worth or not the usage of a GPU to accelerate a specific fragment of code. As an example, Trancoso *et al.* [38] analyzed a very simple application when implemented on a GPU, a low-end CPU, and a high-end CPU. They discussed the application's performance on the GPU relative to the CPU and also looked at several of the different variables that can be changed to improve the GPU's performance. They also looked at what factors make a program more likely to be better suited for a GPU than a CPU. Owens *et al.* [33] also studied how the GPU can handle applications that were previously implemented on a CPU. That study looks at the GPU design and discusses the possible performance improvements offered by the GPU. Also, they analyzed how the GPU was used for specific applications such as in-game physics and computational biophysics.

2.1.3 GPU hardware

Before going into the programming of GPUs (see Section 2.1.4), it is important to have some background on GPU architecture. The GPU programming model exposed by CUDA very much mirrors the underlying hardware. Some of the details that make GPU programming hard are more apparent when looking at the underlying hardware. A CUDA GPU is built around a single kind of processor (as opposed to the different kinds of processors found in earlier GPUs). The processors in the GPU (called MPs, MultiProcessors) all contain some cores called SPs (Streaming Processors). Each MP also contains local memory, called shared memory since it can be accessed by all of the SPs in that MP. The number of MPs varies over the available GPU; cheaper GPUs have as few as one MP, and as you go up in price as the number of MPs increases. Each MP of the GPU can manage a large number of threads; on today's GPUs up to 2048 threads can run on a single MP. The GPU schedules threads in groups of 32, called *Warps* that are executed in lock-step (SIMD style execution). Threads are also divided into Blocks; the threads within a block

Table 2.1: Kind of CUDA functions

	Executed on the:	Only callable from the:
device int DeviceFunction()	device	device
global void KernelFunction()	host	device
host int HostFunction()	host	host

can communicate using the shared memory. The maximum number of threads per block is 1024 [5]. Threads within a warp can communicate via the shared memory without using any synchronization primitive. However, if communication takes place across warps synchronization is necessary. A barrier synchronization mechanism exists to ensure that all threads within a block have reached the same position in the code. Blocks are also grouped into a grid, that is the collection of blocks executing the same program.

2.1.4 Programming for GPUs

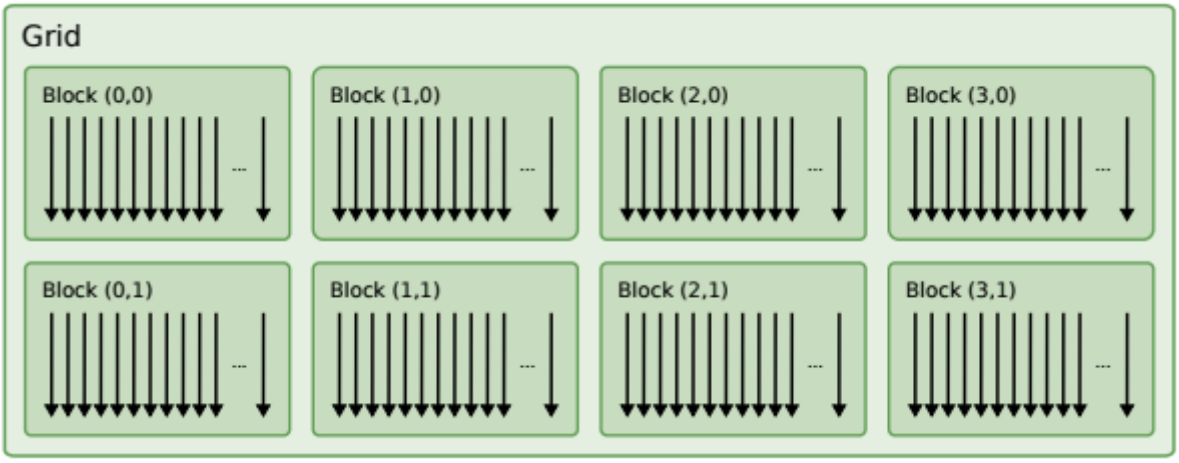
The GPGPU programming landscape has rapidly evolved over the past several years. Nowadays there are several approaches to programming GPUs. For this section, CUDA language will be taken as a reference.

CUDA is a parallel computing platform and programming model developed by NVIDIA for GPU computing. CUDA computing system has two parts: The host and device. The host part is one or many traditional CPU(s) like Intel or AMD CPUs. The device part consists of one or several GPU(s), which are used as co-processors. Since GPUs can enable much parallelism, CUDA devices could help to accelerate those applications that have a lot of data to parallelize. Thus, parallelism is the critical factor in deciding if the use is appropriate for a CPU-GPU system.

CUDA Function Declaration

As stated above, a complete CUDA Program is a mixed code with both GPU and CPU code. Function declaration keywords are designed to support this kind of mix coding. As shown in Table 2.1 functions in CUDA can be declared as `global`, `host` or `device`. A kernel function is a function that will generate a large number threads and is declared as global. During the compilation, the NVCC compiler will generate thousands of threads for the kernel function and map them to the GPU. The keyword `device` is used to declare a CUDA device function that can only be executed on GPU. Also, a CUDA `device` function can only be called from a kernel function. The last keyword, `host`, is designed for declaration for a host function which is run on CPU. The keywords `host` and `device` can be used together to instruct the compiler to generate two versions of the kernel, one running on the CPU and another on the GPU.

Figure 2.1: A two-dimensional arrangement of 8 thread blocks within a grid.



CUDA Thread Organization

When a kernel is executed, the execution is distributed over a grid of thread blocks as shown in Figure 2.1. Since all threads are performed in the kernel function, some mechanisms are necessary to define in which data area the threads must work. In CUDA, all threads are organized in two-level hierarchy-block and grid, as shown in Figure 2.2. Some threads compose a block and use the threadIdx to index them in a block. A grid is organized in the same way and uses blockIdx to index each block in a grid. Both threadIdx and blockIdx are pre-defined variables of CUDA. Besides, there are another two pre-defined variables blockDim and gridDim, which are used to indicate the block and grid dimensions, respectively the number of threads in a block and the number of blocks in a grid. An example is showed in Figure 2.3.

Figure 2.2: CUDA Thread Organization

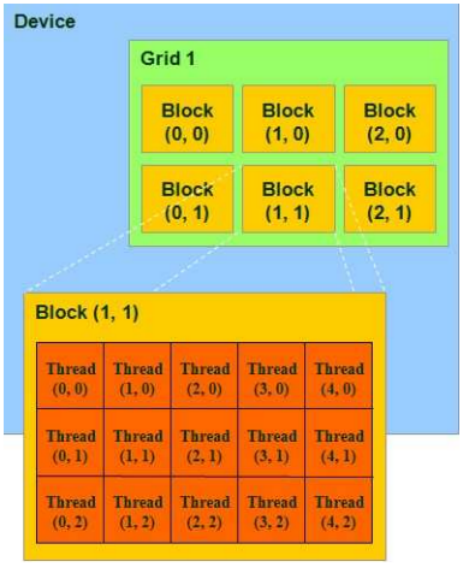
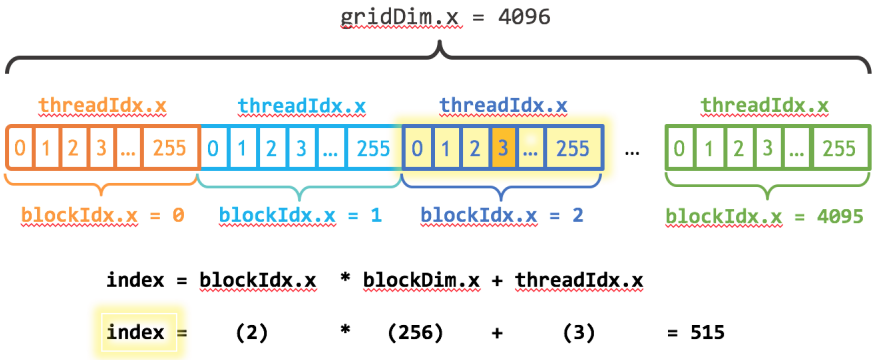


Figure 2.3: CUDA parallel thread hierarchy



CUDA Device Memory

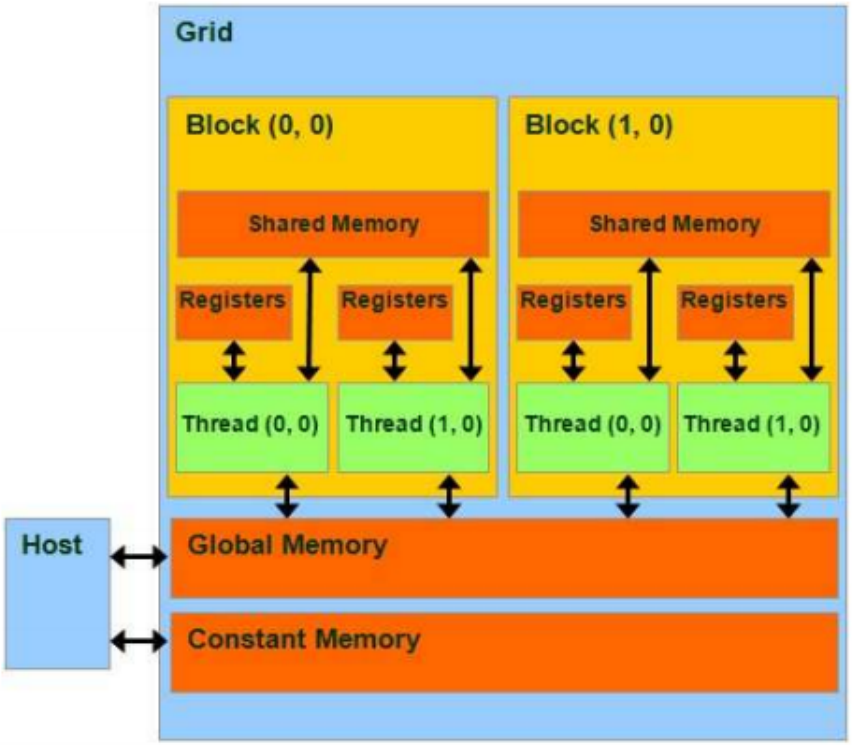
The memory hierarchy is one of the leading factors in a system. Figure 2.4 shows the overview of the CUDA memory hierarchy. There exist four kinds of memory in CUDA: global memory, constant memory, shared memory and register. As shown in Figure 2.4, the global memory and constant memory are used to communicate with the host device.

Global memory can be read and written in a kernel while constant memory can only be read. However, access to the constant memory is much faster than the global memory. Shared memory is designed for the data communication for threads within a block. This is a fast memory but is also very limited in its size that is very smaller than the global memory. Moreover, each thread has several private registers which are the fastest storage elements in a GPU device; these registers are frequently used by their corresponding threads. Since memory access contributes a lot to the computation time of the program, developers should take advantage of these different kinds of memories. The critical rule is that registers and shared memory should be used as much as possible and the data that are not modified in the execution should be stored in the constant memory to have faster access.

OpenCL

After the release of CUDA, an alternative open standard general-purpose programming API was released under the name OpenCL (Open Computing Language) [25]. Initially developed by Apple and subsequently by the Khronos Group, OpenCL allows developers to harness the GPU and multi-core CPUs for general-purpose parallel computation. However, unlike CUDA, OpenCL has multi-vendor and multiplatform support thus enabling parallel code to be executed on AMD and NVIDIA GPUs as well as on x86 CPUs. Whereby, this gives OpenCL the advantage of portability between platforms. However, as Kirk and Hwu [25] noted, OpenCL programs can be inherently more complex if they choose to accommodate multiple platforms and vendors. Developers must use different features from each platform to maximize performance ,and so multiple execution paths dependent on the platform and vendor must be included. This can result in each platform

Figure 2.4: CUDA Memory Hierarchy



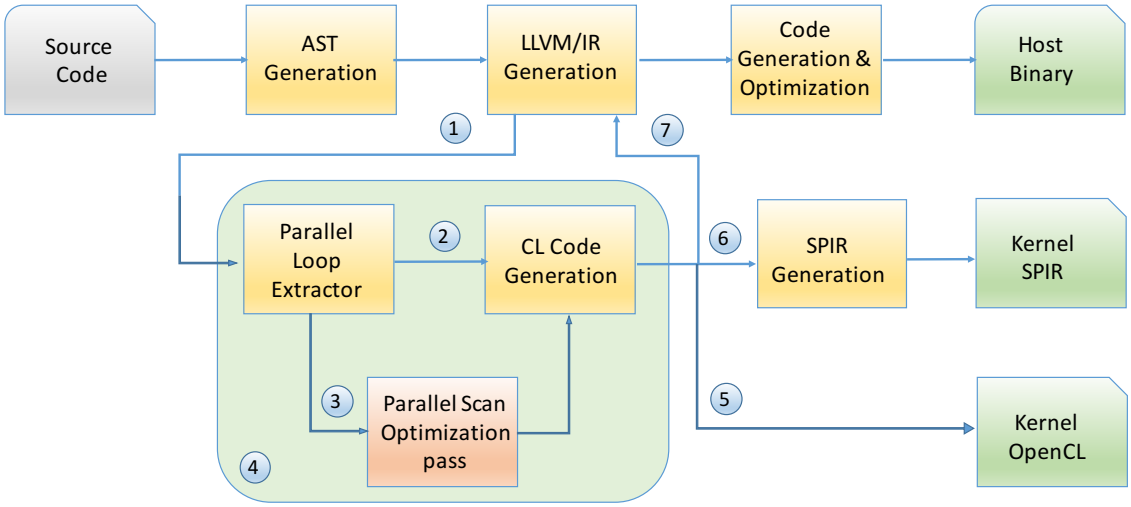
achieving a different execution time depending on the input algorithm, mapping, and usage of platform-specific APIs that may give an advantage to that specific platform. Kirk and Hwu also note that the design of OpenCL is influenced heavily by that of CUDA and as a result working with OpenCL can be very similar to CUDA. As with CUDA, regions of the application that execute in parallel are encapsulated in kernels. OpenCL also has a similar concept of CUDA blocks and threads which have been renamed to *Work group* and *Work item* respectively. The current index of the block within the grid of all blocks has also been renamed as the NDRange. To facilitate support for multiple devices across platforms and vendors, OpenCL introduces the concept of an OpenCL context. Each device is assigned to a context and work is scheduled for execution in a queue of that context [25]. For additional information regarding OpenCL, we direct the reader to the Khronos Group OpenCL specification [2].

2.2 The AClang Compiler

Although OpenCL provides a library that eases the task of offloading kernels to devices, its function calls are complex, have many parameters and require the programmer to have some knowledge of the device architecture's features (e.g, block size, memory model, etc.) in order to enable a correct and effective usage of the device. Hence, OpenCL can still be considered a somehow low-level language for heterogeneous computing.

Introduced through OpenMP 4.0 the new *OpenMP Accelerator Model* [28] proposes

Figure 2.5: AClang compiler pipeline.



a number of new clauses aimed at speeding up the task of programming heterogeneous architectures. This model extends the concept of offloading and enables the programmer to use dedicated directives to define offloading target regions that control data movement between host and devices. Although most OpenMP directives used for multicore hosts can also be used inside the target regions, the new accelerator model easies the tasks of identifying data-parallel computation.

ACLang is an open source (`www.aclang.org`) LLVM/Clang based compiler that implements the OpenMP Accelerator Model. It adds a *OpenCL runtime library* to LLVM/-CLang that supports OpenMP offloading to accelerator devices like GPUs and FGPAs. The kernel functions are extracted from the OpenMP region and are dispatched as OpenCL [2] or SPIR [3] code to be loaded and compiled by OpenCL drivers, before being executed by the device. This whole process is transparent and does not require any programmer intervention.

Figure 2.5 shows the AClang execution flow pipeline with emphasis on the *Parallel Scan Optimization* pass. The LLVM IR generation phase handles the conversion of the AST nodes generated by the Semantic phase into LLVM Intermediate Representation[1]. In this phase, the annotated loops are extracted from the AST ❶, optimized ❸, and/or transformed ❷ into OpenCL kernels in source code format ❺ (see Section 3.2 for more details on the Parallel Scan optimization pass). Kernels can also go through the SPIR generation pass ❻ to produce kernel bit codes in SPIR format. AClang's transformation engine ❹ provides information to the LLVM IR generation phase ❼ to produce intermediate code that calls AClang runtime library functions. These functions are used to perform data offloading and kernel dispatch to the OpenCL driver.

[1]Historically, this was referred to as *codegen*

Chapter 3

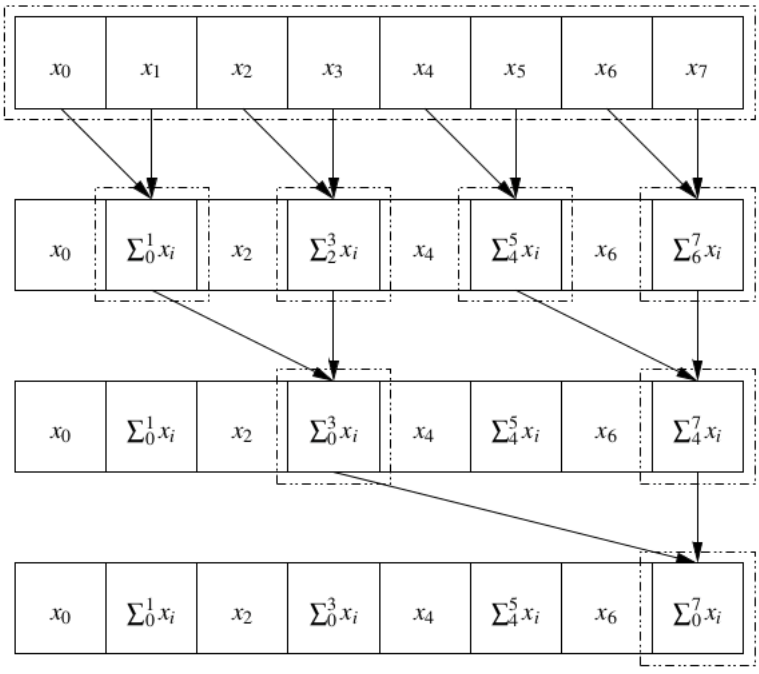
The Parallel Scan

3.1 The Scan algortihm

In the 80's Hillis and Stelle [19] presented several algorithms to enable parallelization of common computing problems. To execute these algorithms, they used the Connection Machine System [20] (CM), a large message-passing based parallel architecture. A crucial observation from those days was that Hillis and Stelle considered that given the almost unlimited number of processors in the CM there was no need to worry about what would happen if the size of the problem they were dealing with surpassed the CM limits. Hence, most of the algorithms were evaluated with at most 65536 elements (the maximum number of processors of the CM).

One of the algorithm proposed by Hillis and Stelle [19] aimed at performing the sum of an array of numbers (see Figure 3.1), an operation which nowadays is calleed *reduction*.

Figure 3.1: Hillis and Steele reduction algorithm

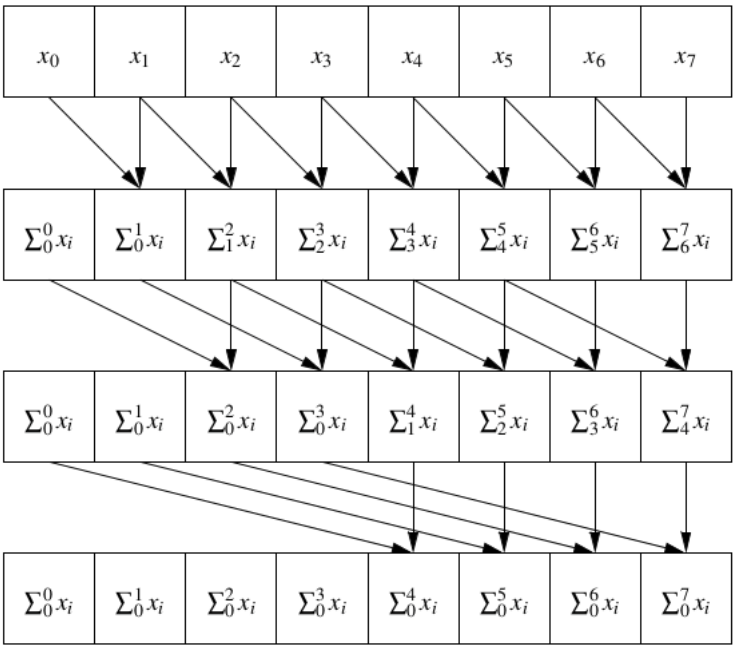


Listing 3.1: The parallel scan and reduction proposed by Hillis and Steele

```
1  (a) Parallel Reduction
2
3  for(int d = 1; d <= log2(n); d++)
4      for(int k = 0 ; k < n ; k++){ //In parallel
5          if( (k+1)%2^d == 0 ){
6              x[k] = x[k - 2^(d-1)] + x[k];
7          }
8      }
9
10 (b) Parallel Scan
11
12 for(int d = 1; d <= log2(n); d++)
13     for(int k = 0 ; k < n ; k++){ //In parallel
14         if( k >= 2^d ){
15           x[k] = x[k - 2^(d-1)] + x[k];
16         }
17     }
18
19
```

The main idea behind their work was to organize the array data at the leaves of a binary tree and perform the sums at each level of the tree in parallel. There are several ways to organize an array onto a binary tree, as shown in Figure 3.1. As an example, Figure 3.1 presents an array of 8 elements named x_0 through x_7. In this algorithm, for the sake of simplicity, the number of elements to be summed is assumed to be an integral power of two. In their solution, there are many processors as elements. Line 4 (in Listing 3.1a) causes all processors to execute lines 5 and 6 (Listing 3.1a) synchronously, but variable k has a different value for each processor, namely, the index of that processor within the array. At the end of the process, x_{n-1} contains the sum of the n elements.

Figure 3.2: Hillis and Steele scan algorithm



As described in Listing 3.1a at each level (iteration) d of the tree, if the processor k

meets the property of line 5 stores the sum of the neighbors at a distance 2^{d-1} on his position. For example, at level $d = 1$ of Figure 3.1 the processor $k = 1$ stores the sum of the neighbors at a distance $2^{1-1} = 1$ (elements of 1 and 0) into the element at index $k = 1$, at the next level of the tree $d = 2$. The distance of the neighbors, in this case, is $2^{2-1} = 2$ and this continues until the algorithm reaches the last level where the root of the tree stores the sum of all nodes in the array. The algorithm is computed in $O(log\ n)$ time, it traverses $log\ n$ levels and at each level, it performs operations in parallel in constant time.

Another algorithm presented by Hillis and Steele [19] was the *scan*. They noticed that by modifying only one line, they could get all the partial sums of an array. Looking at the simple summation algorithm explained above (Listing 3.1a), one can notice that most of the processors are idle most of the time. Line 5 shows that during iteration j, only $n/2^j$ processors are active, and, indeed, half of the processors are never used. Hence, by making the idle processors do useful work one could also compute all the partial sums of the array.

This could be done by means of a variation of the algorithm explained above, and in the same amount of time that took to compute reduction, i.e, $log\ n$. Figure 3.2 shows this process for an array of 8 elements.

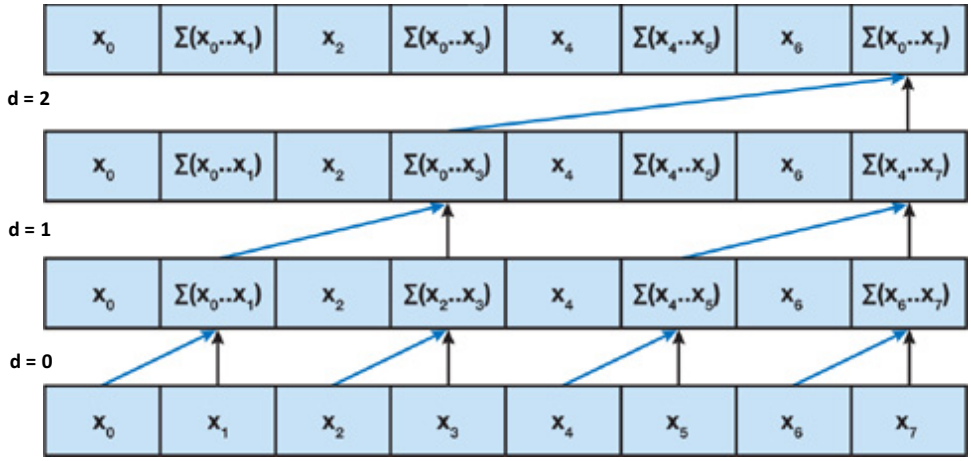
The only difference between this algorithm and the earlier one on reduction is the test in the if statement on the partial-sums that determines whether a processor will perform the assignment (line 5 to 14 in Listing 3.1). This algorithm keeps more processors active: During step j, $n - 2^j$ processors are in use; after step j, element number k has become $\sum_a^k$ where $a = max(0, k - 2^j + 1)$. For example, at level $d = 1$ of Figure 3.2 the processor $k = 1$ stores the sum of the neighbors at distance $2^{1-1} = 1$ (elements of 1 and 0) into the element at index $k = 1$, processor $k = 2$ stores the sum of the neighbors with the same distance 1 (elements of 2 and 1) into the element at index $k = 2$ and its son at the next level of the tree $d = 2$. The distance of the neighbors become $2^{2-1} = 2$ and the processor at $k = 2$ stores the sum of the neighbors 2 and 0 into the element at index $k = 2$ (notice that this element stores the sum from $\sum_0^0$ to $\sum_1^2$, that is $\sum_0^2$). This continues until the algorithm reaches the last level where all nodes have the sum of all its preceding elements, and thus element k stores the sum $\sum_0^k$. The algorithm is computed in $O(log\ n)$ time, it traverses $log\ n$ levels and at each level it performs operations in parallel, i.e, in constant time due to the numbers of processors.

In 1990, Guy E. Blelloch [9] proposed a new method, also based on balanced trees, to perform the scan parallel algorithm. His idea was to build a balanced binary tree on the input data and sweep it from the leaves to the root to compute all the partial sums. A binary tree with n leaves has $d = logn$ levels, and each level d has 2^d nodes. If one addition is performed at each node, the algorithm performs $O(n)$ additions on a single traversal of the tree.

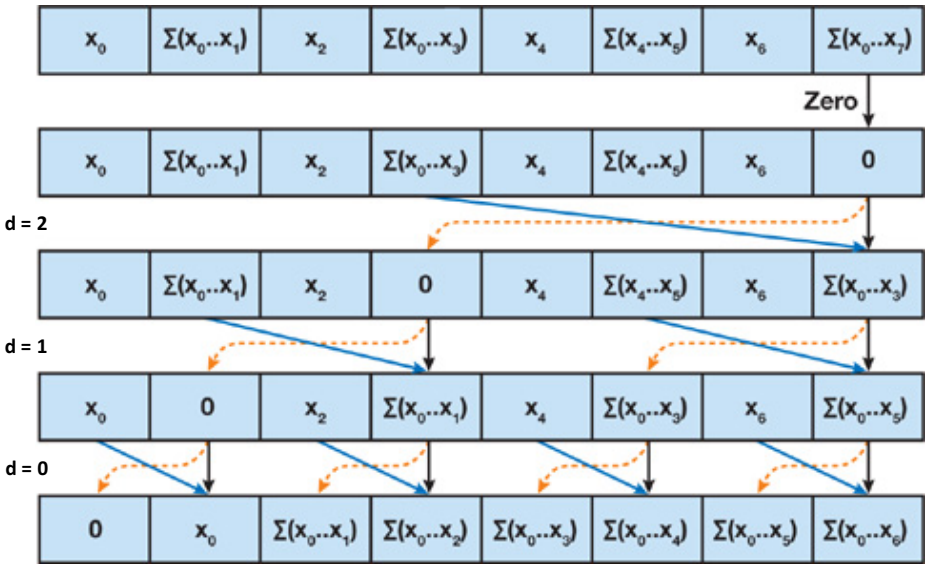
The key idea in [9] is to build a balanced binary tree on the input data x and sweep it to and from the root, scanning at each phase half of the elements of the array. The tree is not an actual data structure, but a concept used to determine what each thread does at each one of the two phases of the traversal. The tree representation is shown in Figures 3.3a – 3.3b where blue and black arrows represent read operations of the elements of x that

Figure 3.3: Parallel scan in $\mathcal{O}(log\ n)$

(a) Up-sweep phase



(b) Down-sweep phase



will be added, and orange arrows represent copy statements.

As shown in Listings 3.2a – 3.2b, the algorithm consists of two phases: *up-sweep* and *down-sweep*. In the up-sweep phase, described in Listing 3.2a the tree is traversed bottom-up computing the scan of half of the internal nodes of the tree in Figure 3.3a. As described in Listing 3.2a at each level (iteration) d of the tree neighbors at distance 2^d are accumulated into the elements at index $k + 2^{(d+1)} - 1, k = 0 \ldots \lceil n/2 \rceil$ of the array (line 18). For example, at level $d = 0$ of Figure 3.3a neighbors at distance $2^{(0+1)} - 1 = 1$ are accumulated into the elements at index $k + 1$, at the next level of the tree. The distance of the neighbors that are accumulated doubles as the tree level is incremented (e.g. the distance is 2 at level $d = 1$) until the partial sum at $x[i - 1]$ is computed. This phase is also known as *parallel reduction*, because after this phase, the root node (the last node in the array) holds the sum of all nodes in the array.

In the down-sweep phase Listing 3.2b, the tree is traversed top-down and the partial

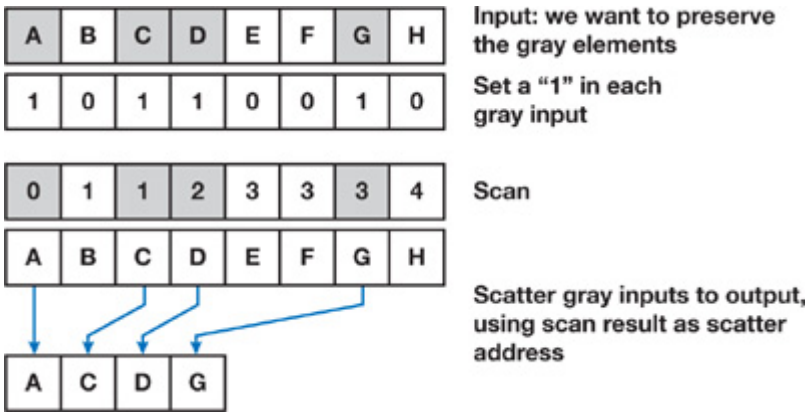
Listing 3.2: The parallel scan implementation proposed by Blelloch

```
1
2   (a) Up-sweep phase of scan parallel implementation
3
4   x[0] = 0;
5   for(d = n >> 1; d > 0; d >>= 1){
6     // We parallelize this section
7     for(k = 0 ; k < n ; k += (1<<(d+1)) ){
8       x[k + (1<<(d+1)) - 1] = x[k + (1<<d) -1] +
9       x[k + (1<<(d+1)) - 1];
10    }
11  }
12
13  (b) Down-sweep phase of scan parallel implementation
14
15  x[n-1] = 0;
16  for(d = log2(n); d >= 0 ; d--){
17    // We parallelize this section
18    for(k = 0 ; k < n ; k += (1<<(d+1))){
19      t = x[k + (1<<d) - 1];
20      x[k + (1<<d) - 1] = x[k + (1<<(d+1)) - 1];
21      x[k + (1<<(d+1)) - 1] = t + x[k + (1<<(d+1)) - 1];
22    }
23  }
24
```

sums computed in the previous phase are propagated downward to accumulate with the entries which did not have their partial sums computed previously in the up-sweep phase. The phase starts by inserting zero at the root of the tree. Then at each step, each node at the current tree level will: (i) sum its value to the former value of its left child and store the result into its right child; and (ii) copy its value to its left child. For example, consider the node at index 7 level $d = 1$ of the tree in Figure 3.3a. That node has two children, a left child at index 5 and a right child at index 7. Hence, during the down-sweep phase two operations will occur: (i) the value at index 7 is summed to the value at index 5 (left child index 7) and is stored into the right child of index 7 (index 7 itself); and (ii) the value at index 7 is copied to index 5 to be used in the next level $d = 0$ (orange arrow to left child of index 7). The algorithm performs $\mathcal{O}(n)$ operations in the first phase (*up-sweep*) and for every level of this phase ($log\ n$ levels) is computed in $\mathcal{O}(1)$ (because it is done in parallel) hence the total time is computed in $\mathcal{O}(log\ n)$, similarly for the second phase (*down-sweep*) the total of operations is $\mathcal{O}(n)$ between adds $(n-1)$ and swaps $(n-1)$ moreover the computed time is $\mathcal{O}(log\ n)$. So the total number of operation of parallel scan is $\mathcal{O}(n)$ and computed time is $\mathcal{O}(log\ n)$ time.

In 2005, Horn [21] proposed the first version of scan parallel for GPUs. Their proposal was based on [19]. Although this implementation is more realistic, it is also limited. As mentioned before in [19], they even did not worry about the number of processors. They considered that it is possible to have a number of processors equal to the size of the input (a not real situation nowadays). So Horn's proposal was limited by the block size of the GPU device (today, this size could be up to 2048). Horn proposed this method by using it as the solution to the problem *StreamCompaction*. This problem is defined as follow: given an input vector, and a key value, it is necessary to reorder the input so that elements with the same key are moved to the end of the vector. On the other hand, the rest of the elements are moved towards the beginning, thus keeping the original order between

Figure 3.4: Example of scan application

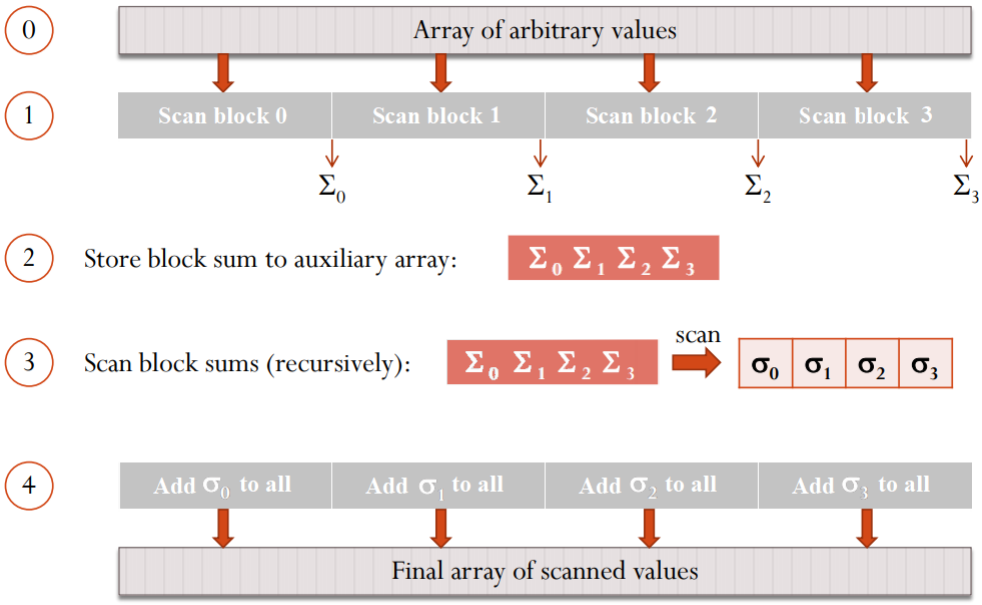


them (See Figure 3.4). Recall that for this version of scan, the complexity is $logn$, and the order of the operation is $nlog\ n$.

In 2007 Mark Harris et al. [18] proposed a new implementation of the scan parallel algorithm. This version was based on the work of Blelloch [9]. In the method proposed by Horn the number of operations is in order of $O(nlog\ n)$ meanwhile the simple serial version of scan performs in order of $O(n)$ operations, as well as Blelloch's version. That was one of Harris's main motivations: the possibility of at least achieving the same order of operations than the serial version. So, he implemented a method to solve the scan operation based on the work of Blelloch for GPUs. However, that implementation had the same problem of Horn's implementation, the algorithm only works for small arrays, because it is limited by a thread block. Thereby, the main contribution of Mark Harris was to design a new algorithm capable of executing on large arrays. His basic idea is simple: the large array is divided into blocks, each of which can be scanned by a single thread block, and then the scan operations are computed for the blocks, and the total sum of each block is written to another array of sums of blocks. Next, the block sums are examined, generating an array of block increments that are added to all the elements in their respective blocks.

For example (see Figure 3.5), let N be the number of items in the input array, and B be the number of elements processed in a block. In this case, N/B thread blocks of $B/2$ threads each allocated. Here, it is assumed that N is a multiple of B, which is dependent on the architecture of the GPU. The scan algorithm of the previous section is used to scan each block i independently, storing the resulting scans into sequential locations of the output array ①. In this case, one minor modification to the scan algorithm is performed. Before zeroing the last element of block i (the block of code labeled B in line 15 in Listing 3.2), the value (the total sum of block i) is stored into an auxiliary array represented by $\sum$ ②. Then scan $\sum$ is done as before, and the result is written into an array represented by σ ③. Then $\sigma[i]$ is added to all elements of block i ④. After doing all the previous steps, the final array of scanned values is obtained.

Figure 3.5: Scan for large arrays by Mark Harris et al.



3.2 Scan clause implementation in AClang

The Parallel Scan Optimization pass ❸ shown in Figure 2.5 is responsible for implementing the scan clause. This implementation is based on the best parallel scan algorithm known today [36] and is detailed in Section 3.1. Nevertheless, the effectiveness of that algorithm is increased when it runs within a thread block within which it can leverage on data locality.

To apply the parallelized scan from Section 3.1 to data sets larger than a single thread block, an extended four step method was proposed [18]. This method applies twice the scan algorithm described in Listings 3.2a – 3.2b to spread the scan of each block to all blocks of the array. Such method is shown in the block diagram of Figure 3.6 where each number corresponds to one step of the method. In the first step, the method divides the large input array into blocks that are scanned each by a single thread block ❶ using the algorithm from Section 3.1. During the second step, the total sum of all elements of each block (i.e, the value in the last element of the scanned block) is transfered to the corresponding entry of an auxiliary array ❷. In the third step, using again the algorithm in Section 3.1, the method scans the auxiliary array, and writes the output at another array of block sums ❸. At the end of this third step each entry of the array of block sums contains the partial sums of all elements of the blocks up to that entry (inclusive). Finally in the fourth step, for each block, the method adds the previous block sums to the elements of the current block ❹.

The following paragraph describes how AClang implements the parallel scan algorithm. This process makes the necessary calls to the AClang runtime library and populates a template that will execute the scan algorithm with the program data (e.g., type of data to be carried out the scan, the scan operation, the output vector).

In the first step, the algorithm obtains the pieces of information of the omp scan

Listing 3.3: Pseudocode of Scan Parallel implementation in AClang

```
(a) Get information from omp scan clause

I = S.clauses().begin(), E = S.clauses().end();
OpenMPClauseKind ckind = ((*I)->getClauseKind());
if (ckind == OMPC_scan){ //Checking if the clause extracted is our Scan clause.
 OMPVarListClause<OMPScanClause> *list = cast<OMPVarListClause>cast<OMPScanClause>(*I);
   for (auto l = list->varlist_begin(); l != list->varlist_end(); l++) {
     DeclRefExpr *scanVar = cast<DeclRefExpr>(*l);
     const std::string scanVarType = scanVar->getType().getAsString();
     OpenMPScanClauseOperator op = cast<OMPScanClause>(*I)->getOperator();
     ...
   }

(b) Preparing the scan algorithm parameters and openCL environment

ThreadBytes = EmitRuntimeCall(CGM.getMPtoGPURuntime().cl_get_threads_blocks(), KArg);

EmitRuntimeCall(CGM.getMPtoGPURuntime().cl_create_read_write(), Size);
EmitRuntimeCall(CGM.getMPtoGPURuntime().cl_create_program(), FileStrScan);

EmitRuntimeCall(CGM.getMPtoGPURuntime().cl_create_kernel(), FunctionKernel_0);
EmitRuntimeCall(CGM.getMPtoGPURuntime().cl_create_kernel(), FunctionKernel_1);
EmitRuntimeCall(CGM.getMPtoGPURuntime().cl_create_kernel(), FunctionKernel_2);

  EmitRuntimeCall(CGM.getMPtoGPURuntime().cl_set_kernel_arg(), Args);
  EmitRuntimeCall(CGM.getMPtoGPURuntime().cl_execute_kernel(), GroupSize);

EmitRuntimeCall(CGM.getMPtoGPURuntime().cl_release_buffer(), Aux);

(c) Customing the scan generator

 CLOS << "#pragma OPENCL EXTENSION cl_khr_fp64 : enable\n\n";
 std::string includeContents = CGM.OpenMPSupport.getIncludeStr();
 if (includeContents != "") {
   CLOS << includeContents;
 }
 switch (op) {
   case OMPC_SCAN_add:
   case OMPC_SCAN_sub:
     initializer = "0";
  ...
 }
 if (initializer == "") {
 } else {
   CLOS << "\n#define _initializer " << initializer;
 }

CLOS << "\n#define _dataType_ " << scanVarType.substr(0, scanVarType.find_last_of(' '))
   << "\n";
CLOS.close();
```

Listing 3.4: Template to generate the final kernel of scan parallel algorithm

```
1  (a) Header for every kernel
2  Header_kernel = """
3    __kernel void kernel_0 (__global _dataType_ *input,
4    __global _dataType_ *S,
5    const int n) {
6  """
7
8  (b) Kind of Operation
9
10   Oper_0_basic = """  block[bi] = block[bi] _operation_ block[ai];  """
11   Oper_0_user  = """  block[bi] = _operation_(block[bi], block[ai]);"""
12
13
14 (c) Vector result
15
16   Tail_input_basic  = """ input[gid]  = input[gid] _operation_ S[bid];  """
17   Tail_output_basic = """ output[gid] = input[gid] _operation_ S[bid]; """
18
19
```

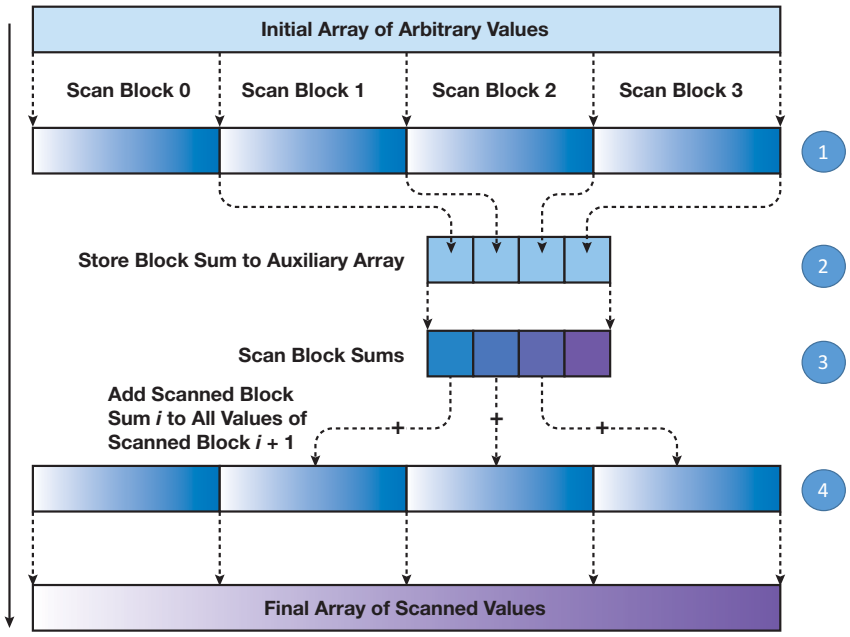
clause, as detailed in Listing 3.3a. For instance, Lines 3 to 12 get the list of variables and the kind of operations associated with the scan clause.

AClang provides a series of methods in the *CodeGenModule* class for calling the runtime library responsible for interfacing with the OpenCL drivers. Those functions have the following structure: $CGM.operation1.MPtoGPURuntime().operation2$. The algorithm retrieves the scan parameters and makes an initial configuration of OpenCL. For instance, Line 16 computes the number of threads per block and the number of blocks. That will be used to call the sub-routines of the parallel scan algorithm. This is a very important step because, as mentioned before, the algorithm only works for input sizes that are powers of 2 (2^k). So, when the size is not a power of two, it is impossible to solve the problem. Hence, to fix it, the higher number that is a product of two powers of two is founded. Let us represent the number of blocks as B and the number of threads per block as T. For example, if the size of the input data is 12, the closest number with the previously mentioned properties is 16. However, the algorithm produces more than one solution. In the example, the answers are (B: 1 - T: 16), (B: 2 - T: 8), and (B: 4 - T: 4). This is important because, in some cases, it is possible to find some results with T greater than the threads provided by the architecture used. Thus, T must be limited according to the characteristics of the architecture used; similarly for B.

Therefore, the maximum size to compute the scan algorithm is limited by the resources provided in the architecture. Later in Chapter 7 it will be shown a proposal to extend that limit. Line 9 retrieves the type of variables in which the program is performing the scan operation. This variable can be of type `int`, `double`, `float`, but can also be a new user-defined type. In this case, it is mandatory for this new type to be defined within the `omp declare target` clauses (see example in Listing 4.6).

Line 10 finds the operator that the programmer defined to use in the scan algorithm. This operator should be a binary associative operator to be used by the scan algorithm; the most common operators are: (+, $*$, &, ||, max, min). In the case that a new variable type was defined, it is also mandatory to define an associative operator for this type (See example in Listing 4.6).

Figure 3.6: Algorithm to perform a block sum scan



Listing 3.5: Pseudocode of Scan Parallel implementation in AClang

```
1  (a) Standard way to write scan algortihm
2    y[0] = 0;
3    for(i = 1 ; i < n ; i++)
4      y[i] = y[i-1] + x[i-1];
5
6
7  (b) Alternative way saving memory to write scan algorithm
8    int aux1 = x[0], aux2;
9    x[0] = 0;
10   for(i = 0 ; i < n ; i++){
11     aux2 = x[i];
12     x[i] = x[i-1] + aux;
13     aux = aux2;
14   }
15
16
```

Line 18 does a needed step to execute the scan algorithm. It creates an auxiliary buffer (as shown in Figure 3.6 ❷). Lines 19 to 25 generate code for the runtime library to call the OpenCL driver to compile the kernels and then dispatch for execution.

Lines 26 executes the necessary kernels to compute the scan algorithm. First, it computes the scan for every single block thread independently (as shown in Figure 3.6 ❶). Then, it executes the second kernel that is in charge to compute the sum of the additions from the auxiliary vector (as shown in Figure 3.6 ❸). It then executes the kernel in charge of distributing the corresponding additions to the positions on the resulting vector (as can see in Figure 3.6 ❹).

Finally, the runtime library transfers the solution data to the vector specified by the programmer; notice that, as we mentioned before, the programmer has two options to receive the resulting vector.

3.2.1 The Template

This section aims to explain how it is built the code of the parallel scan algorithm. Remember that the algorithm is based on the best algorithm known today [36].

To summarize this section, it can be said that the algorithm has only three parameters, those parameters could be different according to the applications. The first one is the type of variable; the second one is the operator defined for the variable used. Finally, the third one specifies how the programmer wants to use of new types of variables and operators. Thereby, as was mentioned before, all the information is collected in the compilation phase to be used for the generation of the kernel that will execute the parallel scan algorithm.

Listing 3.4a defines the header for each kernel. This example shows the header of the first kernel, which defines variable *dataType*. As mentioned before, that information was extracted from the clause scan. As expected, every kernel that uses that variable type has *dataType* replaced by the real variable type.

Listing 3.4b defines how the program will perform the calculations between two variables. In the case that the programmer uses an operation on primitive variables (int, double, float) such as +, $*$, &, | line 10 will be used.

On the other hand, when the programmer defined a new type of variable, he must also define its binary associative operator to be able to use the scan algorithm. In this case, the operation can not be computed simply, the operation has to be defined in the section "omp declare target" for the programmer. That information is recovered from the scan clause as a function and the way to use this new operation is specified in line 11.

Finally, the last component of the template Listing 3.4c refers to where the programmer wants to save the resulting vector. Line 16 does the tasks required when the programmer needs a new vector to save the result. Line 17 activates when the programmer wants to save the resulting vector in the input data vector.

Back to Listing 3.3c, Line 32 fills a standard header. Lines 33 to 36 analyze if the programmer defined a new type of variable with its respective operator. If is true, that information is placed immediately after the header mentioned before.

Lines 37 to 46 define the neutral value according to the operation defined by the programmer. A neutral value is defined by the operation ($a = a \oplus neutralValue$) where a is any variable and $\oplus$ an operator that operates any variable with the neutral. For example, for the sum operation, the neutral value is 0, for the multiplication is 1 and so on. When scan runs with a basic C/C++ primitive operator, the neutral element is set by default internally in the compiler. However when the programmer defines a new type of variable, the OpenMP standard enforces the programmer to define neutral value in the section "omp declare target". Line 48 defines the keyword *_dataType_*; it represents the type of variable used by the programmer. Finally, line 49 closes the template that will perform the scan parallel algorithm. The code describes here and its discussion is a small glimpse of the final OpenCL code generation process. Our goal was to show only the mainly components, in the hope that they could work as a guide to the understanding of the translation process from the omp clause to OpenCL code. Much information has been left aside for the sake of simplicity. To see a final version of the template for one example see Chapter 4.

Chapter 4

Using Scan

4.1 Stream Compaction

Stream compaction is an important primitive in a variety of general-purpose applications, including collision detection and sparse matrix compression. Also, Stream compaction is the primary method for transforming a heterogeneous vector, with elements of many types, into homogeneous vectors, in which each item has the same type. This is particularly useful with vectors that have some elements that are interesting and many elements that are not interesting. Stream compaction produces a smaller vector with only interesting elements. With this smaller vectors, computation is more efficient, because computation is performed only on the elements of interest.

Informally, stream compaction is a filtering operation: from an input vector, it selects a subset of this vector and packs that subset into a dense output vector. Figure 4.1 shows an example. More formally, stream compaction takes an input vector A and a predicate p, and outputs only those elements in A for which $p(A)$ is true, preserving the ordering of the input elements. Stream compaction requires two steps, a scan and a scatter.

1. The first step generates a temporary vector where the elements that pass the predicate are set to 1, and the other elements are set to 0. We then scan this temporary vector. For each element that passes the predicate, the result of the scan now contains the destination address for that element in the output vector.

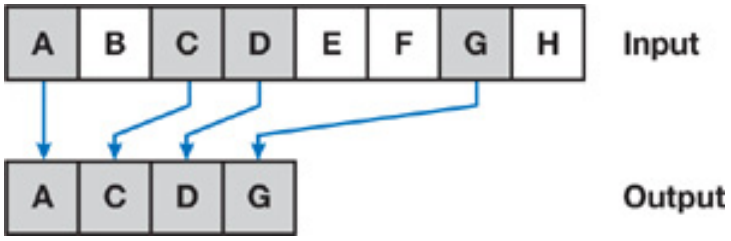
2. The second step scatters the input elements to the output vector using the addresses generated by the scan.

To illustrated technique, see the following example. Consider vector A below and a predicate that is 1 when an element is greater than 10.

$A = < 17, 4, 6, 8, 11, 5, 13, 19, 0, 24 >$ and the desired output is $< 17, 11, 13, 19, 24 >$

As mentioned before, the first step performs a bit-vector operation that produces vector *bits* where the i element is 1 if it satisfies the predicate, and 0 otherwise. For the example, the result of this step is:

Figure 4.1: Stream Compaction Example



Listing 4.1: Fragment of Stream Compaction Benchemark

```
1
2   int main(){
3
4     input = (int *)malloc( sizeof(int) * N );
5     bits = (int *)malloc( sizeof(int) * N );
6     bitsum = (int *)malloc( sizeof(int) * N );
7     output = (int *)malloc( sizeof(int) * N );
8
9     fill( input );
10
11    int predicate = read_predicate();
12
13    f( input, bits, N, predicate ); //Fill bits in parallel
14
15    bitsum[0] = 0;
16    #pragma omp target device(GPU) map(from:bits[:N],to:bitssum[:N])
17    #pragma omp parallel for scan(+:bitsum)
18    for(int i = 1 ; i < N ; i++)
19      bitsum[i] = bitsum[i-1] +  bits[i-1];
20
21    exclusive_to_inclusive( bitsum, bit ); //In parallel transform bitsum to its
      inclusive version
22
23    g( output,input, bitsum, bit ); //In parallel fill the output vector
24
25  }
26
```

$input$ $< 17, 4, 6, 8, 11, 5, 13, 19, 0, 24 >$
$bits$ $< 1, 0, 0, 0, 1, 0, 1, 1, 0, 1 >$

The algorithm then scans vector *bits* into *bitsum* below:

$bitsum$ $< 1, 1, 1, 1, 2, 2, 3, 4, 4, 5 >$

In a second step, for every element with value 1 in the vector *bits*, the value from the vector input is saved into the address that contains the element i of the vector *bitsum*. After that, the final output vector is computed as below:

$output$ $< 17, 11, 13, 19, 24 >$

Listing 4.1 presents a fragment from *StreamCompaction*. The algorithm is based on the two steps approach mentioned before. Lines 4 to 7 create the necessary vectors in

Listing 4.2: Fragment of Stream Compaction kernel generated

```
1  #pragma OPENCL EXTENSION cl_khr_fp64 : enable
2
3  #define _initializer 0
4
5
6  __kernel void kernel_0 (__global int *input,
7  __global int *S,
8  const int n) {
9  ...
10 }
11
12 __kernel void kernel_1 (__global int *input,
13 const int n) {
14
15 ...
16 }
17
18 __kernel void kernel_2(__global int *outpu, __global int *input,
19 __global int *S) {
20 ...
21 }
22
```

addition to input and output. Line 13 fills the vector bit in parallel. The `target` clause (lines 16–17) defines the portion of the program that will be executed by the accelerator device (lines 18–19) defined in the line 16. Since this first *scan clause* only provides an exclusive version, it is necessary an additional step (Line 21) to pass from the exclusive to the inclusive version. Finally, Line 23 fills the vector output with the information generated in the vectors *bitsum* and *bits*.

Listing 4.2 presents the kernel generated by AClang. Since this is a basic application of scan, the kernel generated has a basic structure. Line 3 defines the neutral or also called identity element. In this case, the operation used was a sum. Thus the neutral value is 0. As it can be seen, each kernel has the type of variable for the vectors; this information was extracted from the OpenMP clause and replaced as was explained in section 3.2.

Lines 6 to 10 show the kernel that performs steps ❶ and ❷ of Figure 3.6. Lines 12 to 16 show the kernel that computes the step ❸ of Figure 3.6. Finally, lines 18 to 21 show the kernel that performs the step ❹ of the same figure.

4.2 Radix Sort

A sorting algorithm puts elements of a list in certain order. This section presents a Radix Sort algorithm parallelized using the scan operator. It is well know how a Radix Sort algorithm works. For this reason, the section focuses only on explaining the parallelized version.

The basic idea is to considerer each element to be sorted digit by digit, from the least to the most significant digit. For every digit, the elements will be rearranged. Consider, for example, a list of four elements having four binary digits each. Listing 4.3 shows a visual representation of how the algorithm works.

The following steps show how the radix sort could eventually be parallelized.

1. Generate a vector of the list (bit in the same position, starting from the least signif-

Listing 4.3: Radix Sort algorithm basic idea

```
1  1) Elements representation
2  Element #    1       2       3       4
3  Value:       7       14      4       1
4  Binary:      0111    1110    0100    0001
5
6  2) At first step, Radix sort algorithm rearranges the elements by the values of
7     the bit analized(bit 0):
8  Element #    2       3       1       4
9  Value:       14      4       7       1
10 Binary:      1110    0100    0111    0001
11 bit 0:       0       0       1       1
12
13 3) Finalized the first step, it is neccesary analize the next bit (bit 1):
14 Element #    3       4       2       1
15 Value:       4       1       14      7
16 Binary:      0100    0001    1110    0111
17 bit 1:       0       0       1       1
18
19 4) And so on (bit 2):
20 Element #    4       3       2       1
21 Value:       1       4       14      7
22 Binary:      0001    0100    1110    0111
23 bit 2:       0       1       1       1
24
25 5) And move them again:
26 Element #    4       3       1       2
27 Value:       1       4       7       14
28 Binary:      0001    0100    0111    1110
29 bit 3:       0       0       0       1
30
31
```

icant bit) where every bit that is 0 in the new vector is 1 (Predicate: (bit&1==0)) otherwise the element in the vector is 0.

2. Scan the vector, and record the sum of the predicate vector in the process. Notice, the scan algorithm works for arrays of arbitrary sizes instead of 2^n sizes; however as explained before the *scan clause* works for any arbitrary size.

3. Flip bits of the predicate, and scan them.

4. Move the values in the vector using the following rule:

   (a) For the i^{th} element in the vector:

   (b) If the i^{th} predicate (from the vector generated in step 1) is true, move the i^{th} value to the index in the i^{th} element of the predicate scan.

   (c) Else, move the i^{th} value to the index in the i^{th} element of the opposite array of the Predicate Scan plus the sum of the original Predicate.

5. Move to the next significant bit (NSB).

In the code Listing 4.4, line 7 indicates the traversal of every bit, which, depending on the type of the variable could be 15, 31 or 63. Line 9 defines an auxiliary variable to help to work on the current bit. Next lines (10 – 14) generate the vector mentioned above in step 1, and also generates the opposite vector (see lines 12 – 13). The following lines (17

Listing 4.4: Fragment of Radix Sort benchmark

```
1  int main(){
2    ...
3    predicateTrueScan = ( unsigned int* )malloc( numElem * sizeof(unsigned int) );
4    predicateFalseScan = ( unsigned int* )malloc( numElem * sizeof(unsigned int) );
5    ...
6    unsigned int max_bits = 31; //Unsigned int type
7    for (unsigned int bit = 0; bit < max_bits; bit++){
8
9      nsb = 1<<bit;
10     for(int i = 0 ; i < N ; i++){
11       int r = ((inputVals[i] & nsb) == 0);
12       predicateTrueScan[i] = r;
13       predicateFalseScan[i] = predicate[i] = !r;
14     }
15
16
17     #pragma omp target device(GPU) map(tofrom:predicateTrueScan[:N])
18     #pragma omp parallel for scan(+:predicateTrueScan)
19     for(int i = 1 ; i < N ; i++)
20       predicateTrueScan[i] += predicateTrueScan[i-1];
21     ...
22     #pragma omp target device(GPU) map(tofrom:predicateFalseScan[:N])
23     #pragma omp parallel for scan(+:predicateFalseScan)
24     for(int i = 1 ; i < N ; i++)
25       predicateFalseScan[i] += predicateFalseScan[i-1];
26
27     for(int i = 0 ; i < N ; i++){
28       if ( predicate[i] == 1 )
29         newLoc = predicateFalseScan[i] + numPredicateTrueElements;
30       else
31         newLoc = predicateTrueScan[i];
32       outputVals[newLoc] = inputVals[i];
33     }
34
35   }
36 }
37
```

– 25) compute the scan operator for the vector of line 12. Finally, lines (27 – 32) move the elements in accordance to the vectors generated in the previous step.

Listing 4.5 presents the OpenCL kernel generated by the AClang compiler. The kernel has three main components which were detailed before.

4.3 Polynomial Evaluation

Given a Polynomial P with coefficients $a_n, a_{n-1}...a_0$, the *polynomial evaluation* of $P_{(x)}$ is an operation that computes P when x takes some specific value. The use of polynomials appears in settings ranging from basic chemistry and physics to economics and social science. They are also used in calculus and numerical analysis to approximate functions.

$$P(x) = a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_1 x + a_0 \tag{4.1}$$

This section shows how to use a non primitive variable (`int`, `long`, `float`, `double`, `bool`, `char`) and the AClang scan clause implementation to solving polynomial evaluation. Listing 4.6 presents a fragment of the code to perform the value of the polynomial. Equation 4.1 is the basic representation of a polynomial.

Listing 4.5: Fragment of Radix Sort kernel generated

```
__kernel void kernel_0 (__global unsigned int *input,
__global unsigned int *S,
const int n) {
  ...
}

__kernel void kernel_1 (__global unsigned int *input,
const int n) {

  ...
}

__kernel void kernel_2(__global unsigned int *input,
__global unsigned int *S) {
  ...
}

```

The trick to solve polynomial evaluation using scan is to replace each element (Coefficient) of the Polynomial for a pair. In this case, element a_i becomes the pair (a_i, x) thus resulting in an array of pairs. To perform the scan operation on the new array of pairs, the $\oplus$ operator should be defined as follows:

$$(p, y) \oplus (q, z) = (pz + q, yz) \tag{4.2}$$

It is a little bit difficult to understand this at first, but each such pair is computed in order to summarize the essential knowledge needed for a segment of the array. This segment itself represents a polynomial. The first number in the pair is the value of the segment's polynomial evaluated for x, while the second is x^n, where n is the length of the represented segment of the polynomial.

To use the scan operator, it is necessary first to confirm that the operator is indeed associative. Equation 4.3 demonstrates that the operator is associative.

$$\begin{aligned} ((a, x) \oplus (b, y)) \oplus (c, z) &= (ay + b, xy) \oplus (c, z) \\ ((a, x) \oplus (b, y)) \oplus (c, z) &= ((ay + b)z + c, xyz) = (ayz + bz + c, xyz) \\ (a, x) \oplus ((b, y) \oplus (c, z)) &= (a, x) \oplus (bz + c, yz) = (ayz + bz + c, xyz) \end{aligned} \tag{4.3}$$

Now let us look at an example to see how it works. Suppose that it is necessary to evaluate the polynomial $x^3 + x^2 + 1$ when x is 2. In this case, the coefficients of the polynomial can be represented using the array $< 1, 1, 0, 1 >$. The first step of the algorithm is to convert it into an array of pairs.

$< (1, 2), (1, 2), (0, 2), (1, 2) >$

Now, is possible to apply the $\oplus$ operator defined above to get the result.

$$\begin{aligned} (1,2) \oplus (1,2) \oplus (0,2) \oplus (1,2) &= (1.2 + 1, 2.2) \oplus (0,2) \oplus (1,2) \\ (1,2) \oplus (1,2) \oplus (0,2) \oplus (1,2) &= (3,4) \oplus (0,2) \oplus (1,2) \\ (1,2) \oplus (1,2) \oplus (0,2) \oplus (1,2) &= (3.2 + 0, 4.2) \oplus (1,2) = (6,8) \oplus (1,2) \end{aligned}$$

Listing 4.6: Fragment of the Polynomial Evaluation benchmark

```
1  #pragma omp declare target
2  typedef struct tag_my_struct {
3    int x;
4    int y;
5  } Pair;
6
7  Pair op(Pair A, Pair C) {
8    Pair ans;
9    ans.x = A.x * C.y + C.x;
10   ans.y = A.y * C.y;
11   return ans;
12 }
13 #pragma omp end declare target
14
15 #pragma omp declare scan(op                                    \
16 : Pair                                                        \
17 : omp_out = op(omp_out, omp_in))                              \
18 initializer(omp_priv = (Pair){0, 1})
19
20 int main() {
21 Pair *h;
22 Pair *t;
23
24 t = (Pair *)malloc(N * sizeof(Pair));
25 h = (Pair *)malloc(N * sizeof(Pair));
26 ...
27 #pragma omp target device(GPU) map(from : t[:N]) map(to : h[:N])
28 #pragma omp parallel for scan(op : t)
29 for (int i = 1; i < N; i++)
30   t[i] = op(t[i - 1], h[i - 1]);
31
32
```

$$(1,2) \oplus (1,2) \oplus (0,2) \oplus (1,2) = (6.2+1, 8.2) = (13,16)$$

The result of the operation is $(13,16)$, in which the first element of the pair is the result of evaluating the polynomial for $x = 2$ as: $2^3+2^2+1 = 13$. In the computation above, we proceeded in a left-to-right order as would be done on a single processor. In fact, the parallel scan algorithm combines the first two elements and last two elements in parallel:

$$(1,2) \oplus (1,2) = (1.2+1, 2.2) = (3,4)$$
$$(0,2) \oplus (1,2) = (0.2+1, 2.2) = (1,4)$$

And then it would combine these two results to get the final result $(3.4+1, 4.4) = (13,16)$.

In the code of Listing 4.6, the `target` clause (lines 27–28) define the part of the code that will be executed by the device (lines 31–32). The `map` clauses control the direction of the data flow between the host and the target device. All definitions of data structures or functions that can be used by the `scan` clause, i.e, the `Polynomial` data structure and the `Operator` multiply function (operator$*$), must be enclosed within the `declare target` directives. This is done in lines 1–13 of Listing 4.6. The reader should notice that in this example the use of the operator overloading construct is necessary to solve the problem.

Listing 4.7 shows the header and signatures of the kernel functions generated by the

Listing 4.7: Fragment of Polynomial Evaluation kernel generated

```
1  struct Pair{
2    int x;
3    int y;
4  };
5
6  Point op(Pair A, Pair C) {
7    Point ans;
8    ans.x = A.x * C.y + C.x;
9    ans.y = A.y * C.y;
10   return ans;
11 }
12
13
14 #define omp_priv (Pair){ 0, 1 }
15
16 __kernel void kernel_0 (__global Pair *input,
17 __global Pair *S,
18 const int n) {
19   ...
20 }
21
22 __kernel void kernel_1 (__global Pair *input,
23 const int n) {
24   ...
25 }
26
27 __kernel void kernel_2(__global Pair *output,
28 __global Pair *input,
29 __global Pair *S) {
30   ...
31 }
32
```

compiler for the example showed in Listing 4.6 (Polynomial evaluation). As shown in Listing 4.7, the first lines (1 – 11) is the information about the structure and operator used. The lines (16 – 20) kernel_0 represents the first step of the algorithm which applies the scan operator to the whole problem into blocks, Lines (22 – 25) kernel_1 represent the second step of the algorithm which applies the scan operator over the vector filled in the previous step to get the cumulative sums for all the blocks. Lines (27 – 31) kernel_2 is the final step which fixes cumulative sums for every element to get the final vector.

4.4 Parallelizing Matrix Exponentiation

Given a square matrix A the *Matrix Exponentiation* A^k is an operation that performs the iterative multiplication of A k times. A^k is a central operation in many scientific problems like finding multiple recurrent sequences, solving dynamic programming with fixed linear transitions, finding strings under constraints, among others [30].

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} fib_{n+1} & fib_n \\ fib_n & fib_{n-1} \end{bmatrix} \tag{4.4}$$

Among all problems solved though matrix exponentiation, finding the first n numbers of the Fibonacci sequence is the most well-known [24]. This section shows, from the programmer perspective, how AClang works when using the proposed scan clause to solve this problem. Listing 4.8 presents a fragment from the calculation of the Fibonacci

series using matrix exponentiation[1]. The algorithm is based on Equation 4.4, which can be proven by mathematical induction.

The `target` clause (lines 28–29 in Listing 4.8) defines the portion of the program that will be executed by the accelerator device (lines 30–32). The `map` clauses control the direction of the data flow between the host and the target. All definitions of data structures or functions that can be used by the `scan` clause, i.e, the `Matrix` data structure and the `Matrix` multiply function (operator$*$), must be enclosed in the `declare target` directives. This is done by lines 1–18 in the example. The `declare target` construct will result in the extraction of the appropriate code to be stored inside the kernel.

Notice that the implementation of the scan clause proposed in this work is powerful enough to handle the operator overloading construct already available in OpenMP (lines 20–22). This construct was previously defined in OpenMP for the `reduction` clause and was extend in the AClang compiler to enable the usage in the `scan` clause as well. Listing 4.8 shows how a programmer can use the `scan` clause with the user-defined matrix multiplication operator (`*`). This operator and its neutral value (the identity matrix, in this case) are defined by the `declare scan` directive (lines 20–22). The AClang transformation engine (see Figure 2.5 ❹) gathers this piece of information and through pattern matching techniques builds the kernel that will be dispatched to the target device so as to perform the scan operation.

Listing 4.9 shows the header and signatures of the kernel functions generated by the compiler for the example showed at Listing 4.8 (Fibonacci series). Notice that this example uses the new OpenCL 2.2 for which the kernel language is a static subset of the C++14 standard which includes classes, templates, lambda expressions, function overload, etc. The OpenCL kernel language of any version older than 2.2 is an extended subset of C99, which does not feature operator overloading.

As shown in Listing 4.9, the data type and the user-defined functions in Listing 4.8 are passed to the kernel file as is, and the `omp_priv` variable that represents the identity matrix in the example (neutral element) is transformed to a `#define`. The size (N) of the input matrix x is divided, according to the target device capacity in nt threads and nb blocks. The `kernel_0` function (lines 18–22) is responsible for executing the up-sweep and down-sweep phases for each block of the input array x, and to store into the auxiliary matrix sb (scan block) the cumulative user-defined operation (matrix multiply) of each block. The `kernel_1` function (lines 24–27) is responsible for executing the up-sweep and down-sweep phases of the auxiliary matrix sb that was generated in the `kernel_0` function. Finally, `kernel_2` (lines 29–33) is responsible for applying the user-defined operation (matrix multiply) of element i of the scanned block sb (`kernel_1`) to all values of the scanned block $i+1$ of the input array x, thus producing as result the output matrix y.

The current offloading mechanism in AClang implements the OpenMP 4.X `target data`, `target` and `declare target` constructs. This is done through the *AClang runtime library* which has two main functionalities: (i) it hides the complexity of OpenCL code from the compiler; and (ii) it provides a mapping from OpenMP directives to the OpenCL

[1]Note that in real applications, this is counted in terms of the number of *bigint* arithmetic operations, not primitive fixed-width operations.

Listing 4.8: Fragment of the Fibonacci series benchmark

```
#pragma omp declare target
struct Matrix {
long x00, x01, x10, x11;
// default constructor:
Matrix() { x00 = 1; x01 = 1; x10 = 1; x11 = 0; }
// constructor:
Matrix(long x00_, long x01_, long x10_, long x11_) {
  x00 = x00_; x01 = x01_; x10 = x10_; x11 = x11_;
}
};

Matrix operator*(Matrix A, Matrix C) {
return Matrix(A.x00 * C.x00 + A.x01 * C.x10,
  A.x00 * C.x01 + A.x01 * C.x11,
  A.x10 * C.x00 + A.x11 * C.x10,
  A.x10 * C.x01 + A.x11 * C.x11);
};
#pragma omp end declare target

#pragma omp declare scan( * : Matrix:                          \
omp_out = omp_out * omp_in)  \
initializer(omp_priv = Matrix(1,0,0,1))

int main() {
Matrix *x = new Matrix[N];
Matrix *y = new Matrix[N];
...
#pragma omp target device(GPU) map(tofrom: y[:N]) map(to: x[:N])
#pragma omp parallel for scan( * : y)
for (int i = 1; i < N; i++)
  y[i] = y[i - 1] * x[i - 1]);
  ...
}

```

API, thus avoiding the need for device manufacturers to build specific OpenMP drivers for their accelerator devices.

The AClang compiler generates calls to the AClang runtime library whenever a `target data` or `target` directive is encountered. As shown in the Fibonacci Series example (Listing 4.8), the `declare target` construct will result in the extraction of the appropriate code to be stored inside the kernel. Also, the AClang runtime library is responsible to initialize the data structures that handle the devices and the context and command queues for each device. In addition, it creates the necessary data structures to store the handlers for the kernels and the buffers and to offload data to the accelerator device memory. In AClang, it is responsibility of the compiler to generate the code needed to manage all the phases required by the scan algorithm. Therefore, no changes were made to the runtime library.

Listing 4.9: Fragment of Fibonacci Series kernel generated

```
struct Matrix {
long x00, x01, x10, x11;
Matrix() { x00 = 1; x01 = 1; x10 = 1; x11 = 0; }
Matrix(long x00_, long x01_, long x10_, long x11_) {
  x00 = x00_; x01 = x01_; x10 = x10_; x11 = x11_;
}
};

Matrix operator*(Matrix A, Matrix C) {
  return Matrix(A.x00 * C.x00 + A.x01 * C.x10,
  A.x00 * C.x01 + A.x01 * C.x11,
  A.x10 * C.x00 + A.x11 * C.x10,
  A.x10 * C.x01 + A.x11 * C.x11);
};

#define omp_priv Matrix(1, 0, 0, 1)

__kernel void kernel_0 (__global Matrix *x,
__global Matrix *sb,
const int nt) {
...
}

__kernel void kernel_1 (__global Matrix *sb,
const int nb) {
...
}

__kernel void kernel_2(__global Matrix *y,
__global Matrix *x,
__global Matrix *sb) {
...
}

```

Chapter 5

Related Works

The all-prefix-sums operation has been used around for centuries as the recurrence $x_i = a_i + x_{i-1}$. In 1963, Ofman [32] suggested the use of the scan operation to execute a parallel circuit for the addition of binary numbers. Later in 1971, Stone [37] suggested an implementation of parallel scan on a perfect shuffle network to implement polynomial evaluation. Ladner and Fischer [26] first showed a general method for deriving efficient parallel solutions to the scan problem (the prefix problem was the term used by them) on Boolean circuits that simulate finite-state transducers. Brent and Kung [11] in their discussion about chip complexity of binary arithmetic showed an efficient VLSI layout for a scan circuit. At software level, parallel scan algorithms can be classified into two categories: those that assume that the number p of processors is unlimited and those that assume that p is fixed and $p < n$.

During the 80's Hillis and Steele [19] developed approaches to parallelize many serial algorithms. Although at that time these algorithms seemed to have only sequential solutions, they were able to parallelize them by using the `The Connection Machine` (CM) [20] which had many thousands of processors (unlimited processors). One of these algorithms was the sum of the elements of an array, also known as reduction. With a slight modification of the reduction algorithm, Hillis and Stelle proposed a novel solution to compute *All Partial Sums* of an array, which today is known as *prefix sum* or simply scan. However, thescan algorithm in [19] has a limitation: it only works when the number of values in the array is a power of two. Comparing with the serial version that performs $O(n)$ operations, this proposed algorithm performs $O(nlg\ n)$ operations.

To reduce the number of additional operations, in 1989, in the work *The scan operation and their applications* [9] Blelloch discussed extensively the problem and argued convincingly that the scan operation should be considered a primitive parallel operation and should be, whenever possible, implemented in hardware. In the Connection Machine (CM), in which project Blelloch participated, the scan primitive was implemented as microcode. This scan was implemented using a binary balanced tree as was explained in 3.1 and was showed that the number of operations performed was reduced to $O(n)$. Scan was then used to parallelize some very relevant algorithms like: Maximum-Flow, Maximal Independent Set, Minimum Spanning Tree, K-D Tree and Line of Sight, thus improving their asymptotic reaching a complexity of $\mathcal{O}(nlog\ n)$ to $\mathcal{O}(log\ n)$ for some of these algorithms.

With the emergence of general purpose GPUs(limited number of processors), Horn [21] adapted the algorithm proposed in [19] using his GPU prefix sum implementation. The algorithm was used to solve the problem of extracting the undesired elements of a set. Scan was used to determine the undesired elements, and this was followed by a search and gather operation to compact the set. This problem is known as *Stream Compaction*, and has a running time of $\mathcal{O}(log\ n)$.

In [18] Mark Harris et al. adapted the algorithm of Blelloch [9] for GPU. That implementation is better than the solution proposed by Horn [21]. The main difference between those two approaches is the number of operations executed to solve the problem. In the case of [21], the total number of operations is in order of $O(n.log\ n)$, and in the case of [18] the total number of operations is $O(n)$, the same number as in the serial version.

Also in [18], Harris et al. presented a solution to handle large arrays in GPUs (rembember that in this case the number of processors is limited). Their scan algorithm overcame the power of two constraint through *divide-and-conquer* and *padding* so that arrays of arbitrary size could be handled. This novel solution was explained in 3.1.

In [35] Sengupta and Harris presented several optimizations for the implementation proposed in [18]. Those optimizations were designed to deliver maximum performance for regular execution paths via a *Single-Instruction, Multiple-Thread* (SIMT) architectures and regular data access patterns through memory coalescing. That work was the base for the widely used CUDPP library [1], which presents an easy and efficient, but limited use of the scan operator.

In [22] Bell and Hoberock designed a library called Thrust. That library resembles the C++ Standard Template Library (STL). Thrust parallel template library allows to implement high-performance applications with minimal programming effort. The library offers an implementation of the scan operator that easies the task of the programmer. Thrust was used to implement the CUDA version of the benchmarks described in Chapter 6.

Shengen Yan et. al. [41] implemented the scan operator in OpenCL based on [18]. He improved the performance by reducing the number of memory accesses from $3n$ to $2n$ and eliminating global barrier synchronization completely.

In 2015, Wiefferink [39] implemented other version of the scan operation in OpenCL. This work improved the branch divergence of the algorithm in [9] [18]. As expected, this implementation works on NVIDIA and AMD GPU platforms, unlike most previous versions that just worked on NVIDIA GPUs.

Chapter 6

Experimental Evaluation

This section presents an experimental evaluation using a prototype implementation of the OpenMP scan clause in the AClang compiler. The experiments in this section use three heterogeneous CPU-GPU architectures:

1. A desktop with 2.1 GHz 32 cores Intel Xeon CPU E5-2620, NVIDIA Tesla K40c GPU with 12GB and 2880 CUDA cores running Linux Fedora release 23;

2. A laptop with 2.4 GHz dual-core Intel Core i5 processor integrated with an Intel Iris GPU containing 40 execution units, and running MacOS Sierra 10.12.4; and

3. A mobile Exynos 8890 Octa-core CPU (4x2.3 GHz Mongoose & 4x1.6 GHz Cortex-A53) integrated with an ARM Mali-T880 MP12 GPU (12x650 Mhz), and running Android OS, v6.0 (Marshmallow)

The experiments were carried out by a set of micro-benchmarks shown on Table 6.1 that were specially selected to evaluate the proposed scan clause and to provide significant insight on the strengths and weaknesses of its implementation in OpenMP. This set of micro-benchmarks was designed to enable the exploration of the parallel scan algorithms of representative applications in scientific computing. For each micro-benchmark used in the evaluation three versions were developed:

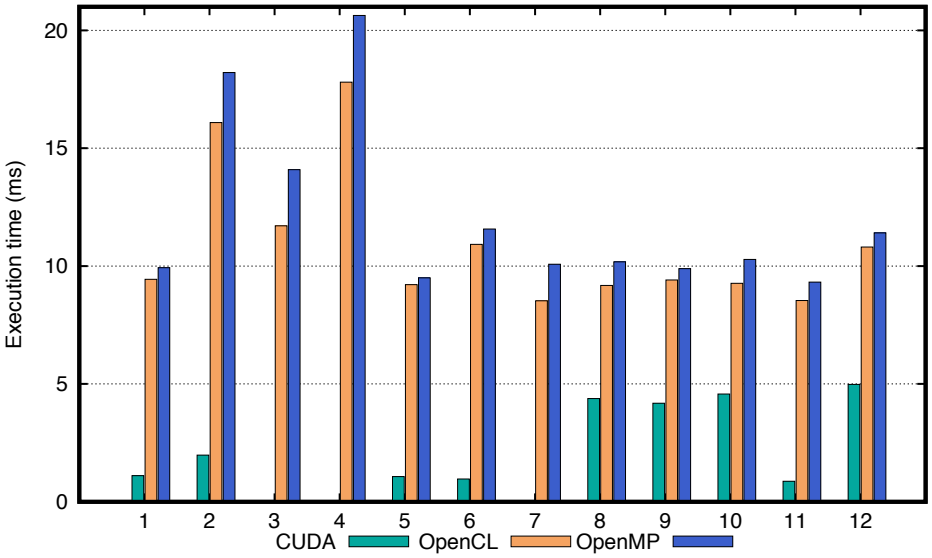
1. A CUDA based version, using the Trust C++ template library [4]. Thrust provides a rich collection of data parallel primitives such as scan, sort, and reduce, allowing the implementation of high performance parallel applications with minimal programming effort through a high-level interface that is fully interoperable with CUDA C. However, the parallel scan implementation only allows vectors of primitive data types, i,e. it does not allow the use of structures (compound data types);

2. An OpenCL version using the same algorithms used in the implementation of the OpenMP scan clause in AClang; and,

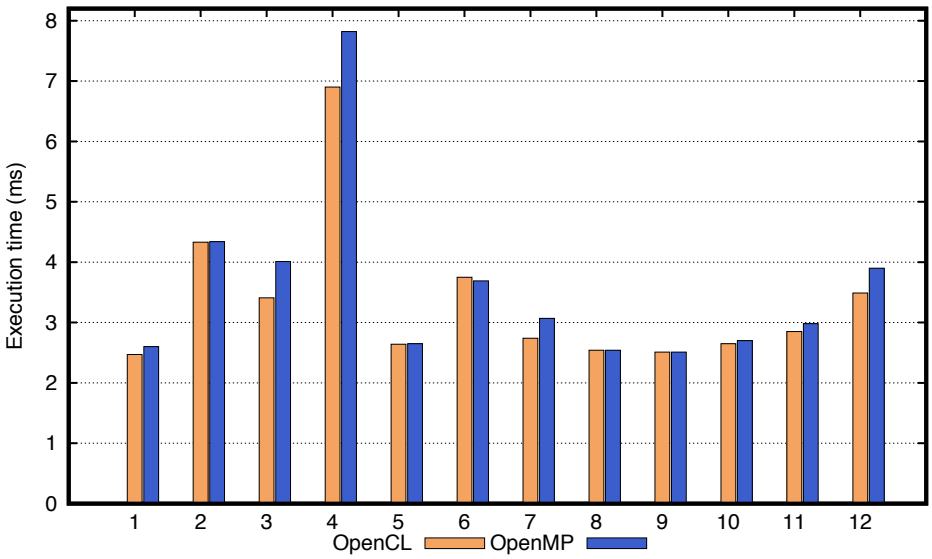
3. a C/C++ version using the proposed OpenMP parallel scan clausA which enables a higher level of abstraction when compared to the OpenCL and CUDA versions.

Figure 6.1: Analysis of parallel scan using a set of micro-benchmarks

(a) The execution on Intel Xeon CPU E5-2620 with NVIDIA Tesla K40c



(b) The execution on Intel Core i5 with Intel Iris GPU



(c) The execution on Exynos 8890 Octa-core with ARM/Mali-T880

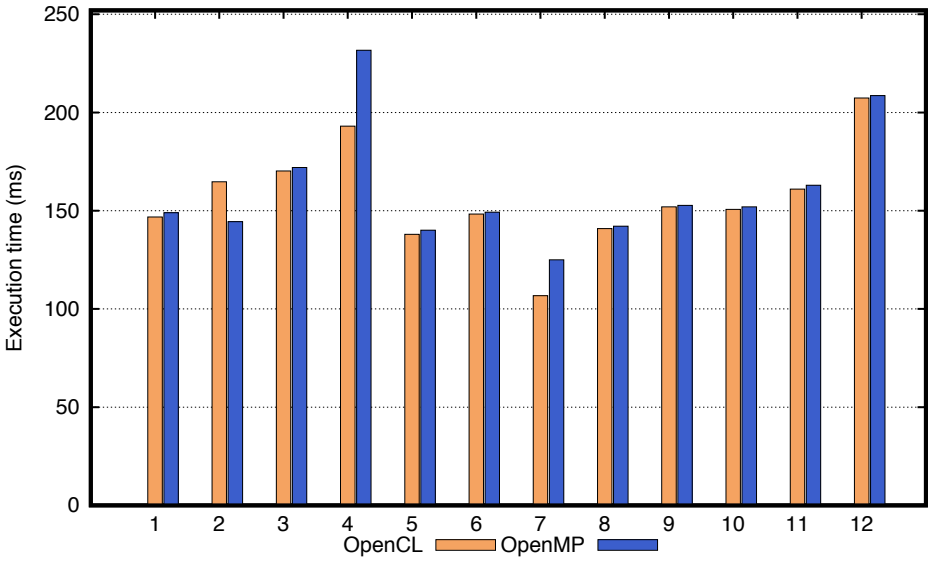


Table 6.1: micro-benchmarks

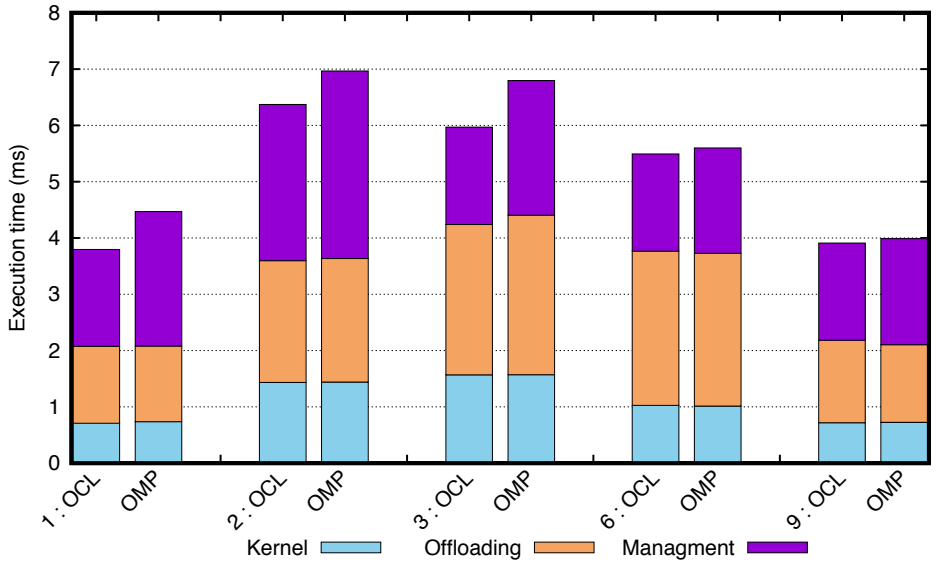
Index	Name	Definition	Used on
1	Stream Compaction	Operation of removing unwanted elements in a collection.	Parallel breadth tree traversing, ray tracing, etc.
2	Longest Span with same Sum in two Binary arrays	Given two binary arrays $arr1[]$ and $arr2[]$ of same size n, find the length of the longest common $span(i, j)$ where $j >= i$ such that $arr1[i]+arr1[i+1]+\ldots+arr1[j] = arr2[i]+arr2[i+1]+\ldots+arr2[j]$.	Programming competitive.
3	Polynomial Evaluation	Given an array a of coefficients and a number x, compute the value of: $a_n x^n+a_{n-1}x^{n-1}+\ldots+a_1 x^1+a0$	Calculus, Abstract Algebra
4	Linear Recurrences	A recurrence relation is an equation that recursively defines a multidimensional array of values. Given one or more initial terms, each additional term of the sequence or matrix is defined as a function of the preceding terms.	Analisys of algorithms, digital Signal processing, Fibonacci Numbers.
5	Random Number Generator	Given n numbers, each with some frequency of occurrence, return a random number with probability proportional to its frequency of occurrence.	Statistics, cryptography, gaming, gambling, videogames
6	Upward & Downward Accumulation	Upward/Downward accumulation refers to accumulating on each node information about all decedents/every ancestor.	Solve N-body problem, solve optimization problems on trees, such as Minimum covering set and Maximal independent set.
7	Adding Big Integers	Sum of big integer numbers	Public-key cryptography, mathematical constant computation such as π
8	Count the number of ways to divide an array in three contiguous parts having equal sum	Given an array of n numbers, find out the number of ways to divide the array into three contiguous parts such that the sum of three parts is equal.	Programming competitive.
9	Maximum sum of two non-overlapping subarrays of given size	Given an array, find two subarrays with a specific length K such that sum of these subarrays is maximum among all possible choices of subarrays.	Programming competitive, video games.
10	Maximum Subarray sum modulo m	Given an array of n elements and an integer m find the maximum value of the sum of its subarray modulo m.	Programming competitive.
11	Maximum occurred integer in n ranges	Given n ranges of the form L and R, the task is to find the maximum occurred integer in all the ranges. If more than one such integer exits print the smallest one.	Programming competitive.
12	Find the prime numbers which can written as sum of most consecutive primes	Given an array of limits, for every limit find the prime number which can be written as the sum of the most consecutive primes smaller than or equal to limit.	Cryptography. Programming Competitive.

The results presented in all experiments of this section are average over ten executions. Variance is negligible; hence, we will not provide error intervals.

To evaluate the performance of the implementation of the proposed OpenMP scan clause, three experiments were performed. In first hardware platform (NVIDIA Tesla) three versions of parallel scan were tested for each benchmark program: (i) CUDA; (ii) OpenCL; and (iii) OpenMP. The other two hardware platforms (Intel Iris and ARM Mali) do not support (CUDA) and thus only the OpenCL and OpenMP implementations were used.

The graphs in Figures 6.1a, 6.1b & 6.1c display the results. The horizontal axis of

Figure 6.2: Analysis of the performance difference between the OpenCL and OpenMP implementations



the graphs denote the number of the benchmark as in Table 6.1 and the vertical axis the execution time. To provide a minimum fair load for the GPUs and to minimize the influence of the data offloading latency appropriate data sizes were used for each input data. In other words, input sizes of $1M$ elements were used for the NVIDIA platform and inputs of $512K$ elements were used for the other two (smaller) hardware platforms (Intel and ARM) .

The graph in Figure 6.1a do not show the results for the CUDA version of experiments 3 (Polynomial Evaluation), 4 (Linear Recurrences) and 7 (Adding Big Integers) due to the lack of support to structured inputs in the CUDA Thrust library.

As shown in Figure 6.1a for all programs the CUDA version performed much better than the OpenCL and OpenMP versions. This is expected, given that the Trust library is optimized and specialized to NVIDIA devices. On the other hand, the focus of this work is to enable a generic scan implementation that could run on a broad range of heterogeneous devices and not only NVIDIA devices. For this reason, our implementation synthesizes generic OpenCL. Of course this does not preclude us from synthesizing CUDA in the future.

In order to better compare the performance of the proposed OpenMP scan clause to the performance of OpenCL code, we measured their percentage difference in all three hardware platforms. The experiments revealed a maximum 20.3%, an average 6.2%, and a standard deviation 7.4% difference in performance. This strongly suggests that the proposed clause can result in a similar performance as when directly programming in OpenCL with the advantage of a smaller programming complexity.

Although small, the performance difference between the OpenCL code and the new OpenMP scan clause is puzzling given that they use the exact same algorithm. After a thorough analysis, we observed that the performance difference was likely due by the AClang runtime library. To evaluate that, a new set of experiments with profile enabled was performed. Figure 6.2 shows the total execution time for some micro-benchmarks.

On the x-axis of the figure are benchmark programs identified by their numbers as listed in Table 6.1 followed by a label OCL (OpenCL) or OMP (OpenMP) to indicate the corresponding implementation. On the y-axis are program execution times. Each bar in the figure is broken down according to the following tasks performed during program execution: (i) kernel computation (Kernel bar); (ii) kernel data offloading (Offloading bar) and (iii) runtime tasks like context creation, queue management, kernel objects creation and GPU dispatch (Managment bar). The analysis reveals that 80% to 90% of the slowdown over the OpenCL implementation are due to the AClang runtime library, not the algorithm itself. In fact, the runtime library does not have specific routines to handle the scan operation data management. This was implemented using existing offload and dispatch operations in the library. We believe that it is possible to reduce this performance difference significantly by slightly adapting the runtime library to provide routines specific to the new scan clause.

When dealing with large inputs, the algorithm computes the scan operator in accordance to the available architectural resources. In other words, the algorithm will divide the input size in slices according to the total number of threads available in the GPU. For example if there are 1M threads and the size of the input is 10M, the algorithm will run a slice of 1M threads 10 times, and then will merge the partials slices to obtain the final answer.

Chapter 7

Conclusions and Future Works

The scan operation is a simple and powerful parallel primitive with a broad range of applications. This work presented an efficient implementation of a new scan clause in OpenMP which exhibts a similar performance as direct programming in OpenCL at a much smaller design effort. The main findings are:

- It is possible to improve the performance of the scan clause by providing specific routines to handle scan (and reduction) operations into the AClang runtime library.
- Based on the evaluated benchmarks, and after investigating the reasons for the differences in performance between the OpenMP and OpenCL versions, it is concluded that the use of the scan clause is perfectly acceptable due to the ease of programming given the high level of abstraction of OpenMP when compared to CUDA and OpenCL.

As a future work, we intend to extend this approach in order to synthesize a CUDA kernel from an scan annotated OpenMP kernel.

Bibliography

[1] Cudpp: Cuda data-parallel primitives library. `http://cudpp.github.io/`.

[2] Opencl: The open standard for parallel programming language of heterogeneous systems. `http://www.khronos.org/opencl`.

[3] Spir: An opencl standard portable intermediate language for parallel compute and graphics. `https://www.khronos.org/spir`.

[4] Thrust c++ template library for cuda. `http://docs.nvidia.com/cuda/thrust/#axzz4imZJuJ1f`. Accessed: 2017-03-01.

[5] Nvidia . Cuda c programming guide. 01 2018.

[6] Anwar M. Ghuloum Allan L. Fisher. Parallelizing complex scans and reductions. In *Programming language design and implementation*, volume 29, pages 135–146, New York, NY, USA, 1994. ACM.

[7] Frances E. Allen. Control flow analysis. In *Proceedings of a symposium on Compiler optimization*, volume 5, pages 1–19, New York, NY, USA, 1970. ACM.

[8] C. Berge and A. Ghouila-Houri. *Programming, Games, and Transportation Networks*. John Wiley, New York, 1965.

[9] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions On Computers.*, 38(11), 1989.

[10] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, nov 1990.

[11] R. P. Brent and H. T. Kung. The area-time complexity of binary multiplication. *J. ACM*, 28(3):521–534, July 1981.

[12] G. Capannini. Designing efficient parallel prefix sum algorithms for gpus. In *2011 IEEE 11th International Conference on Computer and Information Technology*, pages 189–196, Aug 2011.

[13] Ding-Kai Chen, J. Torrellas, and Pen-Chung Yew. An efficient algorithm for the run-time parallelization of doacross loops. In *Proceedings of Supercomputing '94*, pages 518–527, Nov 1994.

[14] Franklin C. Crow. Summed-area tables for texture mapping. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pages 207–212, New York, NY, USA, 1984. ACM.

[15] R Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference for Parallel Processing*, pages 836–844, 1986.

[16] Guido Araujo Divino César Lucas. The batched doacross loop parallelization algorithm. In *In 13th International Conference on High Performance Computing and Simulation (HPCS)*, Amsterdam, Netherlands, 2015. IEEE.

[17] M. Gomez, M. Pereira, X. Martorell, and G. Araujo. Automatic scan parallelization in openmp. In *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pages 85–90, Oct 2017.

[18] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with cuda. *GPU Gems*, 3(39):851–876, 2007.

[19] W. Daniel Hillis and Guy L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12), 1986.

[20] William Daniel Hillis. The connection machine. *Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology*, 1985.

[21] Daniel Horn. Stream reduction operations for gpgpu applications. In M Pharr, editor, *GPU Gems 2*, chapter 36, pages 573–589. Addison-Wesley, 2005.

[22] Wen-mei W. Hwu, editor. *GPU Computing Gems Jade Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[23] G. Iannello. Efficient algorithms for the reduce-scatter operation in loggp. In *IEEE Transactions on Parallel and Distributed Systems*, volume 8, pages 970–982. IEEE, 1997.

[24] Robert C Johnson. *Fibonacci numbers and matrices*. y Maths Dept, Durham University, Durham City, DH1 3LE, UK, 2009.

[25] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2 edition, 2013.

[26] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, October 1980.

[27] Leslie Lamport. The parallel execution of do loops. *Commun. ACM*, 17(2):83–93, February 1974.

[28] Chunhua Liao, Yonghong Yan, Bronis R. de Supinski, Daniel J. Quinlan, and Barbara Chapman. *Early Experiences with the OpenMP Accelerator Model*, pages 84–98. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[29] Rafael C. F. Souza Marcio M. Pereira and Guido Araujo. Compiling and optimizing openmp 4.x programs to opencl and spir. In *13th International Workshop on OpenMP (IWOMP 2017)*, Sept 21-22 2017.

[30] Cleve Moler and Charles Van Loan. Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM Review*, 45(1):3–49, 2003.

[31] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970.

[32] Yu Ofman. On the algorithmic complexity of discrete functions. 7:589, 12 1962.

[33] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[34] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 177–188, Washington, DC, USA, 2004. IEEE Computer Society.

[35] Shubhabrata Sengupta, Mark Harris, and Michael Garland. Efficient parallel scan algorithms for gpus. Technical Report NVR-2008-003, NVIDIA Corporation, December 2008.

[36] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[37] H. S. Stone. Parallel processing with the perfect shuffle. *IEEE Transactions on Computers*, C-20(2):153–161, Feb 1971.

[38] P. Trancoso and M. Charalambous. Exploring graphics processor performance for general purpose applications. In *8th Euromicro Conference on Digital System Design (DSD'05)*, pages 306–313, Aug 2005.

[39] Thijs Wiefferink. Optimization, specification and verification of the prefix sum program in an opencl environment. *Twente Student Conference on IT*, (23), 2015.

[40] C. Z. Xu and V. Chaudhary. Time stamp algorithms for runtime parallelization of doacross loops with dynamic dependences. *IEEE Transactions on Parallel and Distributed Systems*, 12(5):433–450, May 2001.

[41] Shengen Yan, Guoping Long, and Yunquan Zhang. Streamscan: Fast scan algorithms for gpus without global barrier synchronization. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 229–238, New York, NY, USA, 2013.