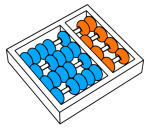
Universidade Estadual de Campinas
Instituto de Computação



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.

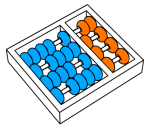
Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

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

Agradecimentos

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 principal para muitos problemas relevantes, tais como: algoritmos de ordenação, análises léxicas, comparação de cadeias de caracteres, filtragem de imagens, entre 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, 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 proposta da clausula OpenMP 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 [24] algorithms to parallelize the iterations of loops which do not have loop-carried dependencies. Although there are approaches such as DOACROSS [14], DSWP [30] and BDX [15] 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 [35, 12] 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 product, maximum, minimum, and logical `AND`, `OR`, and `XOR` to parallelize some very relevant algorithms [28, 13, 8]. Given the relevance of scan in computing, library based parallel implementations of scan have been proposed in the past [17, 9] and designed as library calls in languages like OpenCL and CUDA [32, 11]. Unfortunately, most of these implementations are problem specific leaving the programmer with the task of mastering the complexity of OpenCL and CUDA in order 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* [26]. 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 [21] 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.

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 to its hardware by releasing the OpenGL library they wanted that OpenGL was used as a standardized.

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 to create 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 figure 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 interesting 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 that is the first programmable GPU that means 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 were designed to allow to handle massive computations that are 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. Insomuch as that

processing should be in real time. it must be completed as fast as can be done. The GPU was the answer to how to do this. A GPU consists of many cores. These all cores are capable of executing a thread. Generally, each thread performs an operation on a sample data. This allows that the large data can be operated in parallel. GPU was designed for graphics processing in mind, and this was the main application for GPUs for a long time. Lately, researcher and programmers discovered a opportunity the ability of the GPUs for high levels of parallelization. This is becomes an opportunity to increase the performance of many general purpose programs. That is how started what is known as general purpose graphics processing unit programming or GPGPU programming. With the steady growth interest in GPGPUs programming, GPU vendors started creating GPUs for different applications in mind, thus, the toolkit was created to allow programming in the GPUs. This toolkit, known as CUDA, allows the programmer 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 to harness the potential of the GPU. As was mentioned before, GPUs have the ability to execute a program in parallel. The downside to using GPU is the fact that a CPU is necessary mandatory. The GPU by itself can't 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 what the parameters will be used in this operation. Also, an important function of the CPU is responsible for the memory management of the GPU. Thereby the operations are performed in the GPU, the data must be copied from the CPU memory. In the same way, when the GPU finished its work, it is necessary pass the data from the GPU to the CPU. This is a very expensive operation that often limits the programmers to be able to use the GPU. In addition to the above explained the use of the GPU has several other disadvantages. Similarly to the digital signal processor (DSP). The GPU has a slower clock speed than the CPU. It also does not have the same size of the cache. It does not implement branch prediction or any of that type of optimization. For this reason a GPU cannot keep up with the CPU in serial execution. For these reasons, it is very important to define what portions of the program can be done serially and executed on the CPU, meanwhile what portions could be parallelized and executed on the GPU. If the proportion to be executed in the GPU compensates in time with all the disadvantages mentioned above and it is faster than its serial version, worth the use of the GPU. There have been done many studies on the GPU and how is possible to use the GPU in several research areas. The new ease of use has also contributed to an increase in the number of people wanting to conduct research into the viability of a GPU implementation for their application. Studies also look at the various ways that the GPU can be utilized to improve performance. A very common situation nowadays is the waiting of many people for some research that makes possible the use of GPUs in their applications. As an example, Trancoso *et al.* [33] analysed a very simple application as 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. Another example, Owens *et al.* [29], they studied how the GPU can handle applications that were previously implemented on a CPU. That study looks at the GPU design and discusses about the possible performance improvements offered for the GPU. Also they analyzed how the GPU was used for specific applications such as in-game, physics and computational biophysics.

As CUDA was mentioned, it is also important to mention 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 with the hardware accelerator (mainly GPU), traditionally over PCI Express, or one kernel to communicate with another without host interaction. In the OpenCL model, the user schedules tasks 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 an framework that allows the use of several devices from different vendors. So many developers agree that CUDA has a better performance than OpenCL in Nvidia devices however, no all the costumers have Nvidia card therefore in several times is preferred OpenCL instead of CUDA. Clearly if Nvidia card is a option, CUDA will always be chosen.

2.1.3 GPU hardware

Before going into programming of GPUs (see Section 2.1.4), it is important to have some background about 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 a number of 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 GPUs; cheaper GPUs have as few as one MP, and as you go up in price 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 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 synchronisation primitive. However, if communication takes place across warps synchronisation is necessary. A barrier synchronisation 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. This allowed that in these days 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 designed 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 is consist of one or several GPU(s), which is used as a co-processor. Since GPU can involve a lot of parallelism, CUDA devices could help to accelerate those applications that have a lot of data to parallelize. Thus, parallelism is the key factor to decide if the application is appropriate for a CPU-GPU system.

CUDA Function Declaration

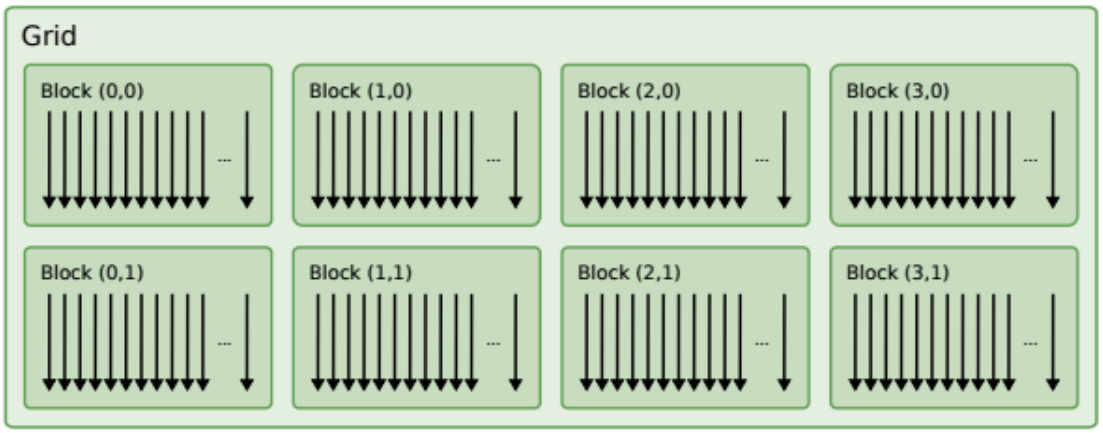
As stated above, a complete CUDA Program is a mixed code with both GPU and CPU code. Function declaration keywords is designed to support this kind of mix coding. As shown in Fig.4.1, functions in CUDA is declared as global, host and device. A kernel function is the function that will generate a large number threads and it is declared as global. During the compilation, the NVCC compiler will generate thousands threads for the kernel function and map them to the GPU. device is used to declare a CUDA device functions that can only be executed on GPU. In addition, a CUDA device function can only be called in a kernel function. The last keyword host is design for declaration for a host function which is run on CPU. host and device can be used to together to instruct the compiler to generate two versions for both GPU and CPU.

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

CUDA Thread Organization

When a kernel is executed, the execution is distributed over a grid of thread blocks as illustrated below in 2.1. Since all the threads are executed in the kernel function, there should be some mechanisms to help to distinguish themselves and know what area of data they should work on. In CUDA, all threads are organized in two-level hierarchy-block and grid, as shown in 2.2. A number of threads compose a block and use threadIdx to index them in a block. A grid is organized in the same way and use blockIdx to index each block in a grid. Both threadIdx and blockIdx are pre-defined variables of CUDA. In addition, there are another two pre-defined variables blockDim and gridDim, which are used to

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



indicate the dimension block and grid by number of threads in a block and number of blocks in a grid, an example is showed in 2.3.

CUDA Device Memory

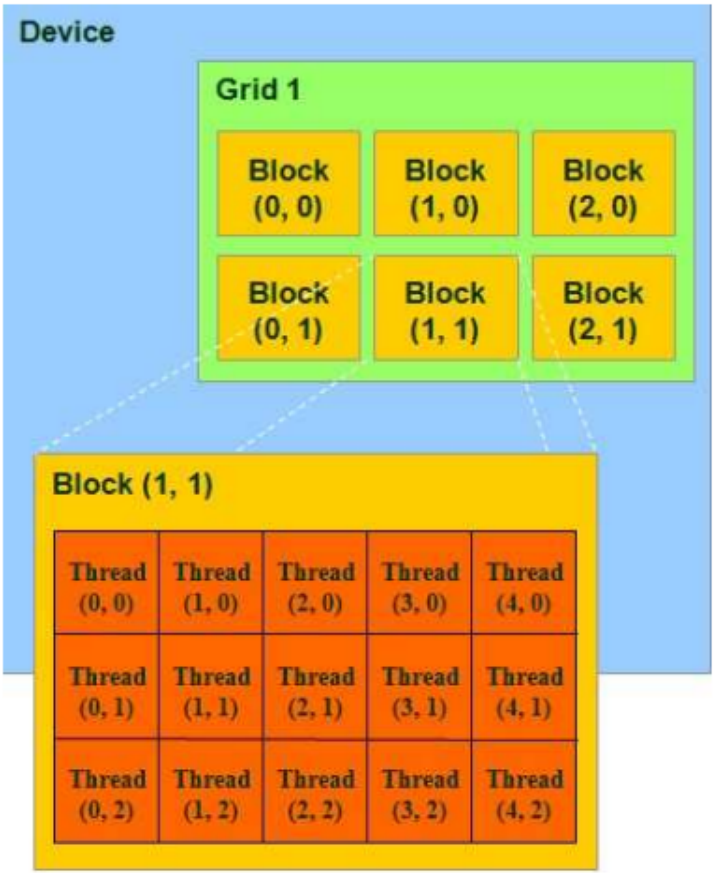
Memory hierarchy is one of the main factors in a system. 2.4 shows the overview of the CUDA memory hierarchy. Exists four kinds of memory in CUDA: global memory, constant memory, shared memory and register. In Fig. 2.4, can be seen that the global memory and constant memory are used to communicate with the host device.

Global memory can be read and write in a kernel while constant memory can only be read. However access to the constant memory is much faster than the global memory because it can be cached. Shared memory is designed for the data communication for threads in a block. This is fast but very limited. Each thread has several privates registers and is the fastest memory in GPU device. It is used for the most frequently used in the program. Since memory access contribute a lot in the computation time of the program, developers should take advantage of different kinds of memory. The key rule is that registers and share 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

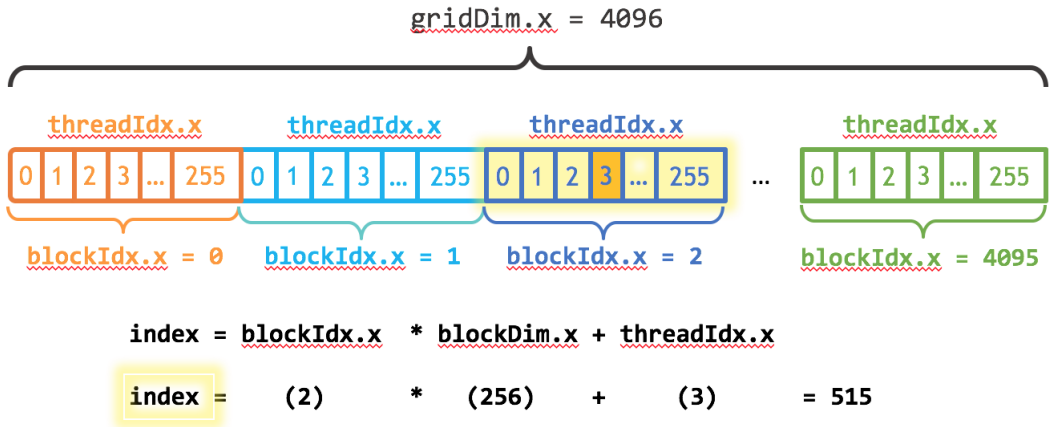
After the release of CUDA, an alternative open standard general-purpose programming API was released under the name OpenCL (Open Computing Language) [23]. Initially Developed by Apple and subsequently the Khronos Group, OpenCL allows developers to harness the GPU and multi-core CPUs for general-purpose parallel computation. However unlike CUDA, OpenCL has multi-vendor and multiplatform support thus allowing parallel code to be executed on AMD and NVIDIA GPUs as well as x86 CPUs. This gives OpenCL the advantage of portability between platforms. However as Kirk and Hwu [23] note, OpenCL programs can be inherently more complex if they choose to accommodate

Figure 2.2: CUDA Thread Organization



multiple platforms and vendors. Developers must use different features from each platform to maximise performance and so multiple execution paths dependent on the platform and vendor must be included. This can result in each platform 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 for that context [23]. For additional information regarding OpenCL, we direct the reader to the Khronos Group OpenCL specification [2].

Figure 2.3: CUDA parallel thread hierarchy



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.

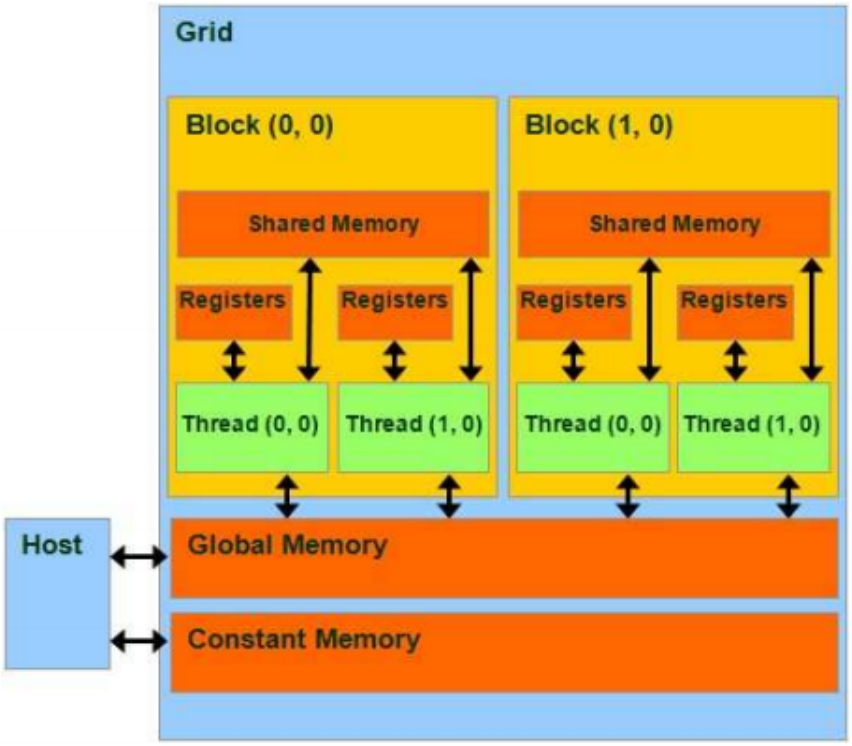
Introduced through OpenMP 4.0 the new *OpenMP Accelerator Model* [25] proposes a number of new clauses aimed at speeding up the task of programming heterogeneous architectures. This model extends the concept of offloading and enables the programmer to use dedicated directives to define offloading target regions that control data movement between host and devices. Although most OpenMP directives used for multicore hosts can also be used inside 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 an *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

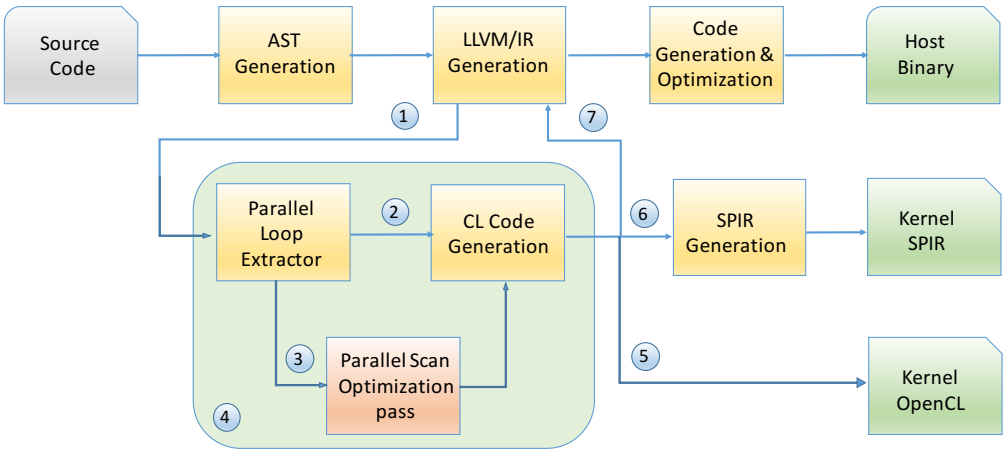
[1]Historically, this was referred to as *codegen*

Figure 2.4: CUDA Memory Hierarchy



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.

Figure 2.5: AClang compiler pipeline.



Chapter 3

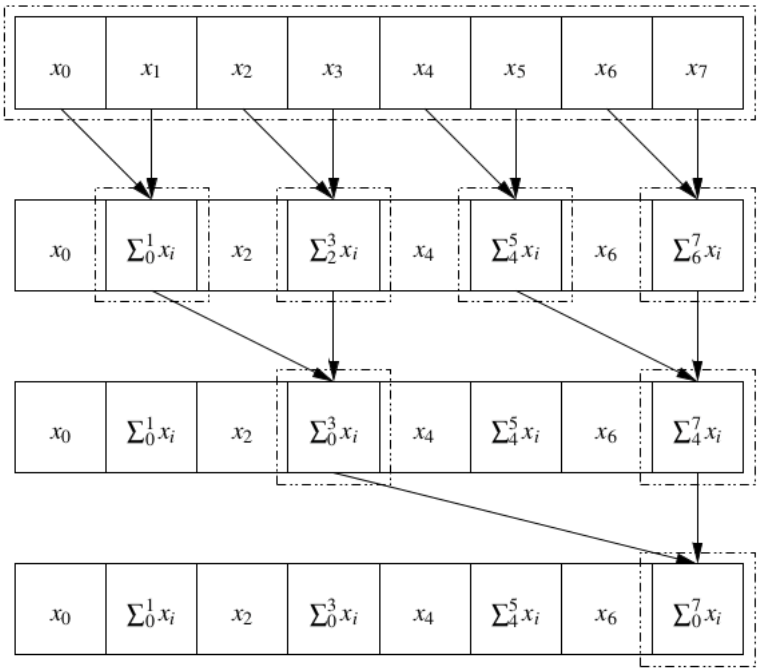
The Parallel Scan

3.1 The Scan algortihm

In the 80's Hillis and Stelle [17] presented several algorithms parallelized. To execute these algorithms, it was used on the Connection Machine System [18]. Thus, a very important observation of these algorithms is that Hillis considered a unlimited number of processors, so they did not worry about that. The algorithms was tested with 65536 elements (maximum number of processors of The Connection Machine [18]). One algorithm presented in the work mentioned aboves performs a sum of an array of numbers (see Figure 3.1). In this days this problem is known as *reduction*.
The main idea to solve the problems is to organize the addends at the leaves of a binary tree and performing the sums at each level of the tree in parallel. Exist several ways to organize an array into a binary tree. Figure 3.1 illustrates the method chosen by them. As an example, it is presented an array of 8 elements named x_0 through x_7. In this algorithm, for purposes of simplicity, the number of elements to be summed is assumed to be an integral power of two. There are as many processors as elements, and the line 4 (in Listing 3.1a) causes all processors to execute the lines 5 and 6 (Listing 3.1a) in synchrony, but the 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.
As described in Listing 3.1a at each level (iteration) d of the tree, if the processor k that meets the property of the line 5 stored the sum of the neighbors at distance 2^{d-1} on his own position. For example, at level $d = 1$ of Figure 3.1 the processor $k = 1$ stored the sum of the neighbors at 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 will be $2^{2-1} = 2$ and so on until reach the last level where the root of the three holds the sum of all nodes in the array. The algorithm is computed in $O(log\ n)$ time, it is performed $log\ n$ levels and each level is performed in parallel which means a constant time. The idea of the algorithm will be used later for one version of scan parallel algorithm.

During the 80's Hillis and Steele [17] developed approaches to parallelize many serial algorithms. Although at that time these algorithms seemed to have only sequential solutions, by using `The Connection Machine` [18] they managed to achieve execution times in $\mathcal{O}(log\ n)$. One of these algorithms was the sum of the elements of an array, also known

Figure 3.1: Hillis and Steele reduction algorithm



as reduction. With a slight modification of reduction Hillis and Stelle proposed a solution to compute *All Partial Sums* of an array, a problem that is currently known as prefix sum or simply scan.

The generalization that scan is an important primitive for parallel computing was presented by Guy E. Blelloch in [9]. In that work scan was defined as a *unit time* primitive under the `PRAM (Parallel Random Access Machine)` model and Blelloch presented a new technique to perform the scan operator. That technique was implemented using a binary balanced tree and was explained in Chapter 3. 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 running time to $\mathcal{O}(log\ n)$.

In [19] Horn proposed an efficient implementation of scan in GPUs. That algorithm was based in Hillis and Steele's work and 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 $\mathcal{O}(n.log\ n)$.

In [16] Mark Harris et al. implemented in GPUs the scan operator based on the the work of Blelloch [9]. That implementation proved to be better than the solution proposed by Horn [19]. The main difference between those two approaches is the number of operations executed to solve the problem. In the case of [19], the total number of operations is $n.log\ n$, and in the case of [16] the total number of operations is n, the same number as in the serial version.

Also in [16], Harris presented a solution to treat large input vectors in GPUs. It is well known that the size of one GPU block is limited and thus the computation of scan

Listing 3.1: The parallel scan and reduction implementation from 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
```

for larger input sizes is done in two scan steps: (i) a first step inside each block; and (ii) a second step among the blocks. The approach proposed in Section 3.2 uses [16] to deal with large input vectors and the work in [9] to perform the individual scan steps.

In [31] Sengupta and Harris presented several optimizations for the implementation proposed in [16]. Those optimizations were designed to deliver maximum performance for regular execution paths via a *Single-Instruction, Multiple-Thread* (SIMT) architecture 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 [20] 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.

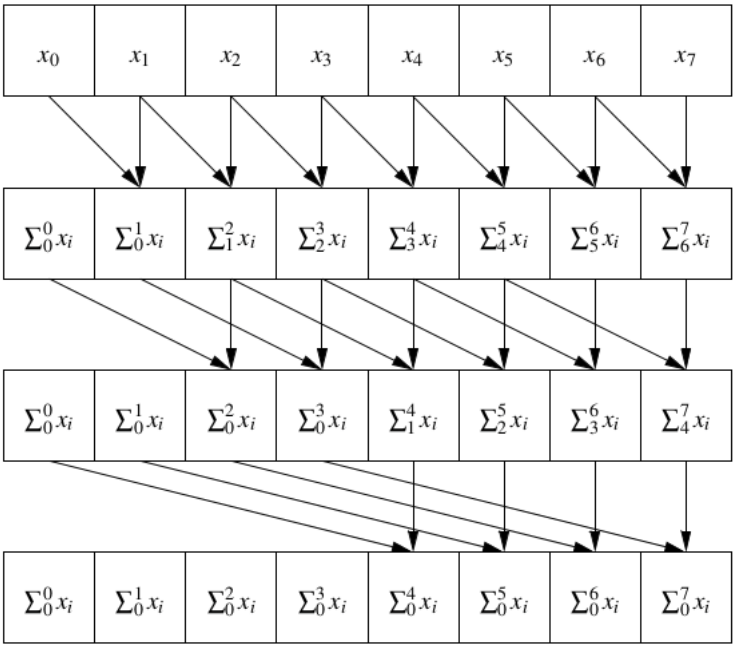
Shengen Yan et. al. [36] implemented the scan operator in OpenCL based on [16], He improved the performance by reducing the number of memory accesses from $3n$ to $2n$ and eliminating global barrier synchronization completely.

In 2015, Wiefferink [34] implemented other version of the scan operation in OpenCL. This work improved the branch divergence of the algorithm in [9] [16]. As expected, this implementation works in NVIDIA and AMD GPU platforms, unlike most previous versions that just worked in NVIDIA GPUs.

This section describes the parallel scan algorithm used in the design of the proposed scan clause. The algorithm is based on the work of Mark Harris et al. [32] and is currently known as the best approach to the parallel computation of scan.

For the sake of simplicity and without any loss of generality please consider the loop in Listing 3.2a (lines 3–10) which contains a sequential scan operation that will be parallelized with the scan clause. Notice that unlike the code of Listing 1.1a (line 5) the current example stores the result of the scan operation into the array x itself (line 8 of Listing 3.2a).

Figure 3.2: Hillis and Steele scan algorithm



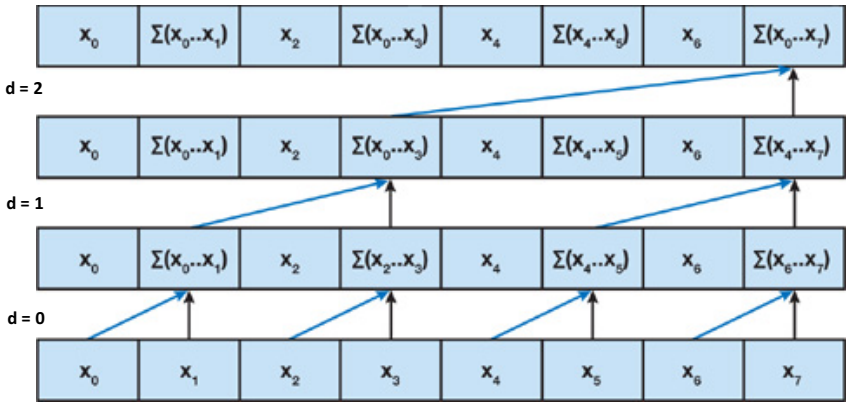
The key idea in [32] is to build a balanced binary tree on the input data x and sweep it to and from the root, scanning at each phase half of the elements of the array. The tree is not an actual data structure, but a concept used to determine what each thread does at each one of the two phases of the 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 will be added, and orange arrows represent copy statements.

As shown in Listings 3.2b – 3.2c, the algorithm consists of two phases: *up-sweep* and *down-sweep*. In the up-sweep phase, described in Listing 3.2b 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 3.2b 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.

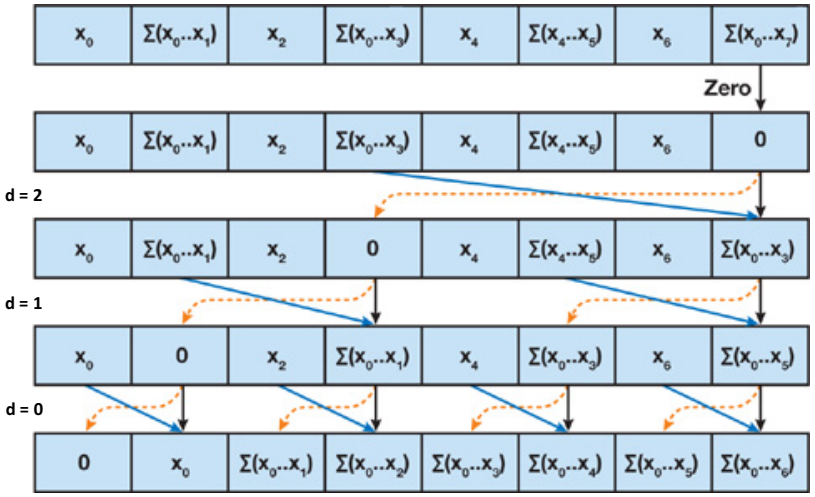
In the down-sweep phase, the tree is traversed top-down and the partial sums computed in the previous phase are propagated downward to accumulate with the entries which did not have their partial sums computed previously in the up-sweep phase. The phase starts by inserting zero at the root of the tree. 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 own 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

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

(a) Up-sweep phase [16]



(b) Down-sweep phase [16]



will occur: (i) the value at index 7 is summed to the value at index 5 (left child of 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 is computed in $\mathcal{O}(1)$ 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.

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 [32] 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

Listing 3.2: The parallel scan implementation

```
(a) Modified sequential implementation

aux = x[0];
x[0] = 0;
#pragma omp parallel for scan(+: x)
for(int i = 1; i < n; i++) {
  temp = x[i];
  x[i] = x[i-1] + aux;
  aux = temp;
}

(b) Up-sweep phase of scan parallel implementation

x[0] = 0;
for(d = n >> 1; d > 0; d >>= 1){
  // We parallelize this section
  for(k = 0 ; k < n ; k += (1<<(d+1)) ){
    x[k + (1<<(d+1)) - 1] = x[k + (1<<d) -1] +
    x[k + (1<<(d+1)) - 1];
  }
}

(c) Down-sweep phase of scan parallel implementation

x[n-1] = 0;
for(d = log2(n); d >= 0 ; d--){
  // We parallelize this section
  for(k = 0 ; k < n ; k += (1<<(d+1))){
    t = x[k + (1<<d) - 1];
    x[k + (1<<d) - 1] = x[k + (1<<(d+1)) - 1];
    x[k + (1<<(d+1)) - 1] = t + x[k + (1<<(d+1)) - 1];
  }
}

```

locality.

In order 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 [16]. This method applies twice the scan algorithm described in Listings 3.2b – 3.2c to spread the scan of each block to all blocks of the array. Such method is shown in the block diagram of Figure 3.4 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 scanning 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

clause, as detailed in Listing 3.3a, lines 3 through 20.

Lines 3 to 7 gets the list of variables and the kind of operation associated with the scan clause.

In Listing 3.3b, it's recovered the buffer where the result of the scan algorithm will be written, as can be seen in Listings 3.5a – 3.5b. In Listings 3.5a – 3.5is a standard way to write scan algorithm (input and output vectors). In Listings 3.5b – 3.5is showed a second way to write scan algorithm, in this case it's necessary just an input data and auxiliary variables.

Lines 24 to 34 retrieve the scan parameters and and makes an initial configuration of OpenCL. Line 24 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's impossible to solve the problem. Hereby, to fix it, it's founded the closest higher number that is a product of two powers of two. Let's 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 properties mentioned before is 16, where we found more of one solution such as (B:1 - T:16), (B:2 - T:8), (B:4 - T:4) and so on. Attention, in some cases it's possible to find some solution with T bigger than the threads provides for the architecture used. Thus, T is limited according of the architecture programmer features, similarly for B.
Therefore, the maximum size to compute the scan algorithm is limited for the resources provided in the architecture, later in 7 it will show a proposal to extend that limit. Line 25 retrieves the type of variable 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 26 find the operator that the programmer defined to use in the scan algorithm. Attention, this operator should be a binary associative operator to could use the scan algorithm, the most common are: (+, $*$, &, $\|$, max, min). In the case that a new variable was defined it is mandatory define an operator for this variable (See example in Listing 4.6).

Line 28 does a needed step to execute the scan algorithm, it created an auxiliary buffer (as can see in 3.4 ❷). 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: $CGM.operation1.MPtoGPURuntime().operation2$.
Lines 29 to 33 generate code for the runtime library to call the OpenCL driver to compile the kernels and dispatch then for execution.

Lines 31 to 33 execute the necessaries kernels to compute the scan algorithm. Line 31 computes the scan for every single block thread independently (as can see in 3.4 ❶). Line 32 executes the second kernel that is in charge to computed the sum of accumulates from the auxiliary vector (as seen in 3.4 ❸). Line 33 executes the kernel in charge to distributes the corresponding accumulates to the positions on the result vector (as can see in 3.4 ❹).

Finally, line 34 transfers the solution vector data to the programmer desired vector, as we mentioned before, the programmer has two options to receive the result 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 [32].

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 refers to the need of the programmer, exactly refers to the use of new types of variables and operators. Additionally, the programmer has two options to recover the result vector. Thereby, as was mentioned before, all the information is recovered in the compilation phase to be used to generate the kernel to execute the parallel scan algorithm.

Listing 3.4a defineds the header for each kernel. This example shows the header of the first kernel. It can be appreciated that it is necessary to define the variable *dataType*. As was mentioned before, that information was extracted from the clause scan. As expected, every kernel that involves the use of the variable type, it was replaced *dataType* for the real variable type.

Listing 3.4b defineds how the program will perform the calculations between two variables. It refers to use a new operation defined for a new variable type. Is possible to operate two variable in a simple way for a primitive variables, for example for two variables x and y of type *int* we will use the line 10 to compute the result.

On other hand, when the programmer defined a new type of variable, he must define also its binary associative operator to be able to use the scan algorithm. In this case, the operation can't be computed in a simple way, 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 was specified in the line 11.

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

Back to Listing 3.3d, Line 38 fillings a standard header. Lines 39 to 42 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 43 to 52 define the neutral value according to the operation, a neutral value is defined by the operation ($a = a \oplus neutralValue$) where a is any variable $\oplus$ is an operator, as we can see, when we operate any variable with the neutral. For example, for the sum operation, the neutral value is 0, for the multiplication is 1 and so on. This information is necessary because the scan algorithm needs it. When it is worked with a basic operation, for default this neutral is defined internally, however when the programmer defined a new type of variable, it is mandatory define the neutral value in the section "omp declare target"

Line 54 defines the word *dataType*, it represents the type of variable that is used for the programmer. Finally, line 55 closes the file that will perform the scan parallel algorithm. Remember, the explained is a very summarized code of the total process. The aims to explain that code is to show the mainly components to understand the process to how pass from the omp clause to OpenCL code. Many informations have been left aside for reasons of priority. To see a final version of the template for one example, see Chapter 4.

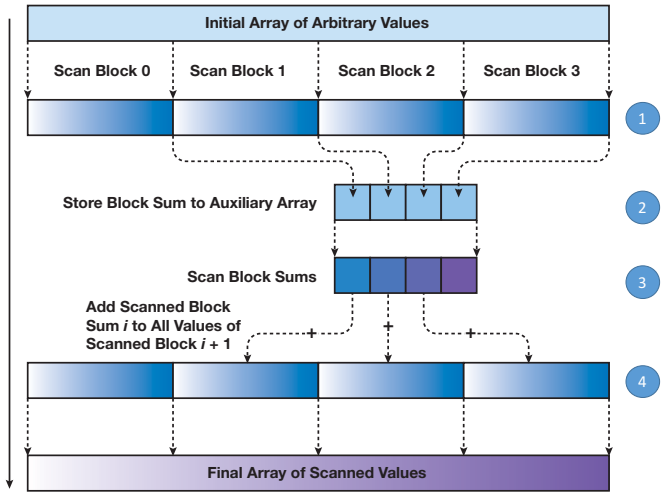
Listing 3.3: Pseudocode of Scan Parallel implementation in Aclang

```
(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);
}

(b) Detecting programmer needs
 if (num_mapped_data == 1) {
    ...
 }
else {
if (num_mapped_data > 2) {
 llvm_unreachable("Unsupported scan clause with more than two mapped data");
}
if (MapClauseTypeValues[0] == OMP_TGT_MAPTYPE_TO ||
     MapClauseTypeValues[0] == OMP_TGT_MAPTYPE_TOFROM) {
  ...
}

(c) Preparing the scan algorithm parameters and openCL environment

ThreadBytes = EmitRuntimeCall(CGM.getMPtoGPURuntime().cl_get_threads_blocks(), KArg);
Q = dyn_cast<ArrayType>(Q.getTypePtr())->getElementType();
OpenMPScanClauseOperator op = cast<OMPScanClause>(*I)->getOperator();

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_release_buffer(), Aux);

(d) 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

```
(a) Header for every kernel
Header_kernel = """
  __kernel void kernel_0 (__global _dataType_ *input,
  __global _dataType_ *S,
  const int n) {
"""

(b) Kind of Operation

  Oper_0_basic = """  block[bi] = block[bi] _operation_ block[ai];  """
  Oper_0_user  = """  block[bi] = _operation_(block[bi], block[ai]);"""


(c) Vector result

  Tail_input_basic  = """ input[gid]  = input[gid] _operation_ S[bid];  """
  Tail_output_basic = """ output[gid] = input[gid] _operation_ S[bid]; """


```

Figure 3.4: Algorithm to perform a block sum scan



Listing 3.5: Pseudocode of Scan Parallel implementation in Aclang

```
(a) Standard way to write scan algortihm
  y[0] = 0;
  for(i = 1 ; i < n ; i++)
    y[i] = y[i-1] + x[i-1];


(b) Alternative way saving memory to write scan algorithm
  int aux1 = x[0], aux2;
  x[0] = 0;
  for(i = 0 ; i < n ; i++){
    aux2 = x[i];
    x[i] = x[i-1] + aux;
    aux = aux2;
  }


```

Chapter 4

Using Scan

4.1 Radix Sort

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

The basic idea is to considerer each element to be sorted digit by digit, from least to most significant. For every digit the elements will be rearranged. Let's take a simple example of 4 elements with 4 binary digits in its representation. Listing 4.1 shows a visual representation of how the algorithm works.

Maicol: Os itens abaixo precisam ser reescritos. Não está claro a explicação do algoritmo But, How can Listing 4.1 do parallel? The next steps show how to do it.

1. Generate a vector of the list (bit in common, starting form the least significant bit) where every bit that is 0 in the new vector is 1 else the value is 0.

2. Scan the vector, and record the sum of the predicate in the process. Notice, Scan algorithm works for arrays of arbitrary size instead of 2^n size, however as how was explained before the propose resolves for any arbitrary size.

3. Flips bits of the predicate, and scan that.

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

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

   (b) If the i^{th} predicate (vector generate in the 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 next significant bit (NSB).

In the code Listing 4.2, the line 7 indicates the analyze for every bit, depending of the type of variable could be 15, 31 or 63. The line 9 is an auxiliary variable to heps to work the current bit. Next lines (10 – 14) generated the vector mentioned above in the

Listing 4.1: 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
```

step 1 also generate the opposite vector of the first one. The following lines (17 – 25) compute scan operator for the to vector mentioned before. The final lines (27 – 32) move the elements in accordance of the vectors generated in the previous step.

In Listing 4.3 is presented the kernel generated for the framework AClang, the kernel has 3 main components which were detailed before.

4.2 Sequence alignment

Sequence alignment is an important tool for researchers in molecular biology in their efforts to relate the molecular structure and function to the underlying sequence. Biological sequence data mainly consists of DNA and protein sequences, which can be treated as strings over a fixed alphabet of characters. Usually, is considered the comparison of two biological sequences. This comparison is done by aligning the two sequences, which refers to stacking one sequence above the other with the intention of matching characters from the two sequences that lie in the same position. To deal with missing characters and extraneous characters, gaps may be inserted into either sequence. A scoring mechanism is designed for each possible alignment and the goal is to find an alignment with the best possible score.

Let m and n be the lengths of the two sequences to be compared. Sequence alignment algorithms typically use dynamic programming in which a table, or multiple tables of size $(m+1)x(n+1)$ are filled.

Listing 4.2: Fragment of Radix Sort benchmark

```
int main(){
  ...
  predicateTrueScan = ( unsigned int* )malloc( numElem * sizeof(unsigned int) );
  predicateFalseScan = ( unsigned int* )malloc( numElem * sizeof(unsigned int) );
  ...
  unsigned int max_bits = 31; //Unsigned int type
  for (unsigned int bit = 0; bit < max_bits; bit++){

    nsb = 1<<bit;
    for(int i = 0 ; i < N ; i++){
      int r = ((inputVals[i] & nsb) == 0);
      predicateTrueScan[i] = r;
      predicateFalseScan[i] = predicate[i] = !r;
    }


    #pragma omp target device(GPU) map(tofrom:predicateTrueScan[:N])
    #pragma omp parallel for scan(+:predicateTrueScan)
    for(int i = 1 ; i < N ; i++)
      predicateTrueScan[i] += predicateTrueScan[i-1];
    ...
    #pragma omp target device(GPU) map(tofrom:predicateFalseScan[:N])
    #pragma omp parallel for scan(+:predicateFalseScan)
    for(int i = 1 ; i < N ; i++)
      predicateFalseScan[i] += predicateFalseScan[i-1];

    for(int i = 0 ; i < N ; i++){
      if ( predicate[i] == 1 )
        newLoc = predicateFalseScan[i] + numPredicateTrueElements;
      else
        newLoc = predicateTrueScan[i];
      outputVals[newLoc] = inputVals[i];
    }

  }
}

```

4.3 Polynomial Evaluation

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

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

Polynomial evaluation is used in many problems as was mentioned above, this section shows how to use a non primitive variable (int, long, float, double, bool, char) using the scan clause to solve this problem, from the programmer perspective and how AClang works when is used the proposed. Listing 4.6 presents a part of the code to perform the value of the polynomial. Equation 4.1 is the basic representation of a polynomial.

Before of describe the code of for the problem, in this section is showed how can solve the problem using Scan operator.

The trick to solve the problem is to replace each element (Coefficient) of the Polynomial to became a pair. In this case, was changed element a_i to become the pair (a_i, x) that created an array of pairs. To perform scan operator on the new array of pairs, the $\oplus$

Listing 4.3: 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) {
  ...
}

```

Listing 4.4: Fragment of Sequence alignment

```

int main() {
int *h;
int *t;

t = (int *)malloc(N * sizeof(int));
w = (int *)malloc(N * sizeof(int));
h = (int *)malloc(N * sizeof(int));

...

#pragma omp target device(GPU) map(from : t[:N]) map(to : h[:N])
#pragma omp parallel for scan(max : t)
for (int i = 1; i < N; i++)
  t[i] = max(t[i - 1], h[i - 1]);


```

operator defined as follows:

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

Where does it come from? It's a little difficult to understand at first, but each such pair is meant 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 scan operator, it's necessary first confirm that the operator is indeed associative.

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

In Equation 4.3 is demonstrated that the operator is associative.
Now let's look at an example to see how it works. Suppose that it's necessary to evaluate

Listing 4.5: Fragment of Radix Sort kernel generated

```
1  __kernel void kernel_0 (__global unsigned int *input,
2  __global unsigned int *S,
3  const int n) {
4  ...
5  }
6
7  __kernel void kernel_1 (__global unsigned int *input,
8  const int n) {
9
10 ...
11 }
12
13 __kernel void kernel_2(__global unsigned int *input,
14 __global unsigned int *S) {
15 ...
16 }
17
```

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.

$(1,2) \oplus (1,2) \oplus (0,2) \oplus (1,2) = (1.2+1, 2.2) \oplus (0,2) \oplus (1,2)$
$(1,2) \oplus (1,2) \oplus (0,2) \oplus (1,2) = (3,4) \oplus (0,2) \oplus (1,2)$
$(1,2) \oplus (1,2) \oplus (0,2) \oplus (1,2) = (3.2+0, 4.2) \oplus (1,2) = (6,8) \oplus (1,2)$
$(1,2) \oplus (1,2) \oplus (0,2) \oplus (1,2) = (6.2+1, 8.2) = (13,16)$

So the result of the operation is $(13,16)$, which has the desire value to compute 13 as its first element: $2^3 + 2^2 + 1 = 13$ of the pair. In the computation above, we proceeded in left-to-right order as would be done on a single processor. In fact, parallel scan algorithm would combine 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 $(3.4+1, 4.4) = (13,16)$.

Now in the code, the `target` clause (lines 27–28 in Listing 4.6) defines 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 in the `declare target` directives. This is done by lines 1–13 in Listing 4.6. Attention, in this example is presented the use of the operator overloading construct that it's necessary to solve the problem.

Listing 4.7 shows the header and signatures of the kernel functions generated by the compiler for the example showed at 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 represented the first part of the algorithm

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
```

who applies scan operator of the whole problem into blocks, The lines (22 – 25) kernel_1 represented the second who applies scan operator over the vector filled in the previous step to get the cumulative sums for all the blocks. The lines (27 – 31) kernel_2 is the final step who 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 [27].

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

Among all problems solved though matrix exponentiation, finding the first n numbers of the Fibonacci sequence is the most well-known [22]. 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

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

Listing 4.7: 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
```

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

Listing 4.8: Fragment of the Fibonacci series benchmark

```
1  #pragma omp declare target
2  struct Matrix {
3  long x00, x01, x10, x11;
4  // default constructor:
5  Matrix() { x00 = 1; x01 = 1; x10 = 1; x11 = 0; }
6  // constructor:
7  Matrix(long x00_, long x01_, long x10_, long x11_) {
8    x00 = x00_; x01 = x01_; x10 = x10_; x11 = x11_;
9  }
10 };
11
12 Matrix operator*(Matrix A, Matrix C) {
13 return Matrix(A.x00 * C.x00 + A.x01 * C.x10,
14   A.x00 * C.x01 + A.x01 * C.x11,
15   A.x10 * C.x00 + A.x11 * C.x10,
16   A.x10 * C.x01 + A.x11 * C.x11);
17 };
18 #pragma omp end declare target
19
20 #pragma omp declare scan( * : Matrix:                    \
21 omp_out = omp_out * omp_in)  \
22 initializer(omp_priv = Matrix(1,0,0,1))
23
24 int main() {
25 Matrix *x = new Matrix[N];
26 Matrix *y = new Matrix[N];
27 ...
28 #pragma omp target device(GPU) map(tofrom: y[:N]) map(to: x[:N])
29 #pragma omp parallel for scan( * : y)
30 for (int i = 1; i < N; i++)
31   y[i] = y[i - 1] * x[i - 1]);
32   ...
33 }
34
```

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

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) {
...
}

```

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.

Chapter 5

Related Works

During the 80's Hillis and Steele [17] developed approaches to parallelize many serial algorithms. Although at that time these algorithms seemed to have only sequential solutions, by using the `The Connection Machine` [18] they managed to achieve execution times in $\mathcal{O}(log\ n)$. One of these algorithms was the sum of the elements of an array, also known as reduction. With a slight modification of reduction Hillis and Stelle proposed a solution to compute *All Partial Sums* of an array, a problem that is currently known as prefix sum or simply scan.

The generalization that scan is an important primitive for parallel computing was presented by Guy E. Blelloch in [9]. In that work scan was defined as a *unit time* primitive under the `PRAM (Parallel Random Access Machine)` model and Blelloch presented a new technique to perform the scan operator. That technique was implemented using a binary balanced tree and was explained in Chapter 3. 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 running time to $\mathcal{O}(log\ n)$.

In [19] Horn proposed an efficient implementation of scan in GPUs. That algorithm was based in Hillis and Steele's work and 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 $\mathcal{O}(n.log\ n)$.

In [16] Mark Harris et al. implemented in GPUs the scan operator based on the the work of Blelloch [9]. That implementation proved to be better than the solution proposed by Horn [19]. The main difference between those two approaches is the number of operations executed to solve the problem. In the case of [19], the total number of operations is $n.log\ n$, and in the case of [16] the total number of operations is n, the same number as in the serial version.

Also in [16], Harris presented a solution to treat large input vectors in GPUs. It is well known that the size of one GPU block is limited and thus the computation of scan for larger input sizes is done in two scan steps: (i) a first step inside each block; and (ii) a second step among the blocks. The approach proposed in Section 3.2 uses [16] to deal with large input vectors and the work in [9] to perform the individual scan steps.

In [31] Sengupta and Harris presented several optimizations for the implementation

proposed in [16]. Those optimizations were designed to deliver maximum performance for regular execution paths via a *Single-Instruction, Multiple-Thread* (SIMT) architecture 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 [20] 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. [36] implemented the scan operator in OpenCL based on [16], He improved the performance by reducing the number of memory accesses from $3n$ to $2n$ and eliminating global barrier synchronization completely.

In 2015, Wiefferink [34] implemented other version of the scan operation in OpenCL. This work improved the branch divergence of the algorithm in [9] [16]. As expected, this implementation works in NVIDIA and AMD GPU platforms, unlike most previous versions that just worked in 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: (i) 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; (ii) 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 (iii) 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)

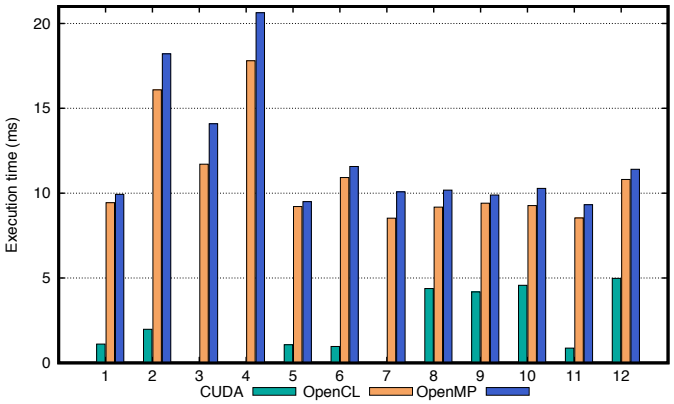
The experiments were carried out by a set of micro-benchmarks shown on Table **??** that were specially selected to evaluate the proposed scan clause and to provide significant insight on the strengths and weaknesses of its implementation in OpenMP. This set of micro-benchmarks was designed to enable 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: (i) 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); (ii) an OpenCL version using the same algorithms used in the implementation of the OpenMP scan clause in AClang; and, (iii) a C/C++ version using the proposed OpenMP parallel scan clause which enables a higher level of abstraction when compared to the OpenCL and CUDA versions.

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

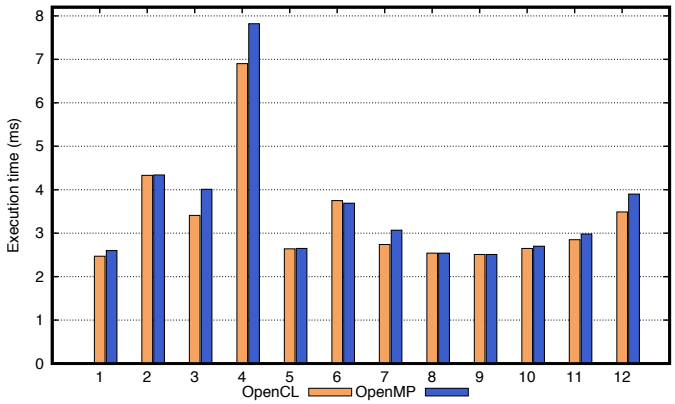
To evaluate the performance of the implementation of the proposed OpenMP scan clause, three experiments were performed. In first hardware platform (NVIDIA Tesla) three versions of parallel scan were tested for each benchmark program: (i) CUDA; (ii) OpenCL; and (iii) OpenMP. The other two hardware platforms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

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

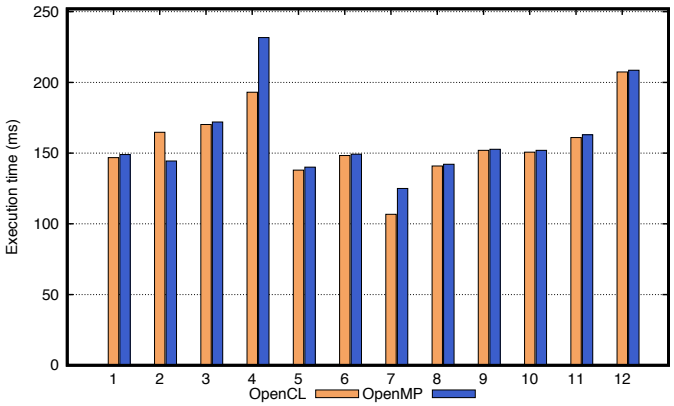
(a) The execution on Intel Xeon CPU E5-2620 with NVIDIA Tesla K40c



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

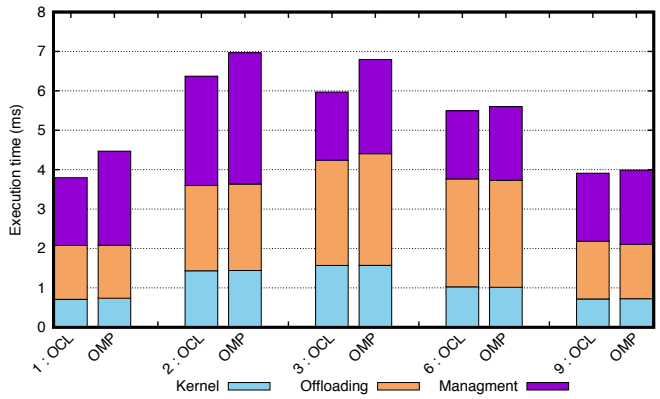


(c) The execution on Exynos 8890 Octa-core with ARM/Mali-T880



The graphs in Figures 6.1a, 6.1b & 6.1c display the results. The horizontal axis of the graphs denote the number of the benchmark as in Table **??** and the vertical axis the execution time. In order 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

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



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 **??** 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 existent 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.

About large inputs, the algorithm computes of scan operator in accordance of user's resources, it means that the algorithm will divide the input size in slices of total threads available in the GPU, for example if the threads available are 1M and the size of the input is 10M, the algorithm will run 10 times slices of 1M 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.

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] 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.

[12] 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.

[13] 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.

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

[15] 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.

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

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

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

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

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

[21] 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.

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

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.