Stream Processing on Multi-Cores with GPUs: Parallel Programming Models' Challenges

Dinei A. Rockenbach[†*], Charles M. Stein[*], Dalvan Griebler[*†], Gabriele Mencagli[‡],
Massimo Torquati[‡], Marco Danelutto[‡], and Luiz G. Fernandes[†]
[*]Laboratory of Advanced Research on Cloud Computing (LARCC), Três de Maio Faculty (SETREM), Três de Maio, Brazil.
[†]School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil.
[‡]Computer Science Department, University of Pisa (UNIPI), Pisa, Italy.
***Corresponding authors**: {dinei.rockenbach, dalvan.griebler}@edu.pucrs.br

***Abstract*—The stream processing paradigm is used in several scientific and enterprise applications in order to continuously compute results out of data items coming from data sources such as sensors. The full exploitation of the potential parallelism offered by current heterogeneous multi-cores equipped with one or more GPUs is still a challenge in the context of stream processing applications. In this work, our main goal is to present the parallel programming challenges that the programmer has to face when exploiting CPUs and GPUs' parallelism at the same time using traditional programming models. We highlight the parallelization methodology in two use-cases (the Mandelbrot Streaming benchmark and the PARSEC's Dedup application) to demonstrate the issues and benefits of using heterogeneous parallel hardware. The experiments conducted demonstrate how a high-level parallel programming model targeting stream processing like the one offered by SPar can be used to reduce the programming effort still offering a good level of performance if compared with state-of-the-art programming models.**

I. Introduction

Stream processing [1], [2], [3], [4] is a popular computing paradigm with a renewed diffusion in several applications belonging to the Big Data domain. The idea underpinning this paradigm is that input data are not immediately available in the form of permanent data structures (as in traditional batch processing), but they consist of an infinite sequence of elementary data items received from several sources with a potentially variable input rate. Most of the modern stream processing applications [5], [6] extract actionable intelligence from such a transient data deluge, in order to run analytic tasks as well as other more complex data mining/machine learning algorithms while new inputs are incrementally received from the sources. Although this is a modern definition and application of stream processing systems, this paradigm has its roots in traditional problems in parallel computing and data-flow programming, where sequences of tasks must be efficiently scheduled to properly exploit underlying computing resources.

Most of these stream processing applications can potentially run on High-Performance Computing (HPC) servers, which are massively parallel architectures due to the combination of multi-core CPUs and many-core GPUs. We can easy find them as rack servers such as the NVidia's DGX-2 that is composed of 16 GPUs in a single box. This powerful computing resources only make sense if the software can actually take advantage of the parallelism available. Consequently, programmers have to follow parallel programming methodologies and use suitable APIs (Application Programming Interface) or parallel programming models. Our choice in this paper was to follow the structured parallel programming approach [7] since it has been proven to help programmers to achieve productivity along with the support of high-level parallel programming models or APIs. We picked-up SPar [3], TBB [8], and FastFlow [4] for expressing parallelism on multi-core as well as CUDA [9] and OpenCL [10] for expressing GPU parallelism. These solutions are representative for the industry and academy domains. Also, future work can extend our parallel algorithms for other programming frameworks.

Our goal is to point out the main challenges for implementing stream parallelism and evaluate the performance of the outcome solutions. First of all, we intend to use a streaming version of the Mandelbrot Set pseudo application to illustrate the main parallel programming models' challenges on multi-core machines equipped with GPU accelerators. Afterward, we intend to follow the best practices in the PARSEC's Dedup benchmark, showing up the robustness and providing new performance insights because Dedup has not been implemented for GPUs previously. Moreover, there are only few works that tackle this problem. Most of them concentrate on data parallelism exploitation for scientific applications or data stream analytics (Section II). Differently, our applications have a stream of data being processed in the CPU while data operators are offloaded in the GPU to compute. Thus, the main scientific contributions of this paper are highlighted as follows:

- A discussion of the parallel programming models's challenges for the implementation of parallel stream processing applications.
- A novel parallel implementation of the PARSEC's Dedup benchmark targeting multi-cores with GPUs.
- A set of experiments and performance evaluation for several parallel programming models (SPar, CUDA, OpenCL, TBB, and FastFlow).

This paper is organized as follows. Section II presents the related work and highlights the main contributions compared to the state-of-the-art. In Section III, we also introduce the

parallel programming models and APIs used. After, Section IV describes the parallel algorithms as well as discusses the programing challenges to express the parallelism. Experiments are demonstrated in Section V to evaluate the performance. Finally, Section VI make the conclusions.

II. Related Work

Our selected related researches are those targeting the same applications and approaching the challenges for stream parallelism exploitation on multi-core equipped with GPUs.

The performance of the Mandelbrot set pseudo application on three platforms (CPU, GPU and Cell BE) was evaluated by [11]. Instead of porting the algorithm for the platforms, they used the Mandelbrot implementation provided by NVIDIA's CUDA SDK code samples and implementations for Cell BE from "linux.conf.au 2008 hackfest". The main outcome is that this pseudo application scales very well on GPUs. Guo et al [12] used their auto-parallelizing compiler for the Single Assignment C (SAC) language in the Mandelbrot algorithm and report speedups of $\sim 22\times$. SPOC (Stream Processing on OCaml) [13] present interesting results in a 2-GPU environment: $134\times$ speedup on OpenCL. This algorithm is used as a case study of SkelCL library in [14] to compare the the programming effort of CUDA, OpenCL, and SkelCL.

To the best of our knowledge, there is not parallelism implementation for GPUs on PARSEC's Dedup benchmark. There are other Dedup applications such as Suttisirikul's work that has used Dedup in a cloud-based backup system [15]. It uses `Sha256` hashing algorithm to identify repeated files and blocks when running the backup. The backup system was able to run at client's machine GPU and transfer only the necessary data to a cloud server. By running the `Sha256` fingerprints on GPU, the speedup achieved was 53 when compared to the CPU version. Bhatotia et al. also implemented the parallelism in a different Dedup-based application for storage systems [16]. Similarly to our work, the application was structured as a pipeline in CPU. However, the strategy was different, where content batches are offloaded to the GPU for processing rabin fingerprints and `Sha1`. The performance achieved was a speedup of 5. Moreover, no support for multi-GPU was implemented and their focus was on optimizing batch partitioning and memory transferring.

Finally, in the perspective of stream processing for streaming analytics, we mention some relevant papers in this area. Saber [5] is a GPU-based framework written in Java using micro-batching to discretize the input stream. The system offloads the processing of each batch on the GPU. The framework supports relational algebra operators (e.g., select, project, filter, join) and window-based aggregates by leveraging a mixed execution model, with both CPU cores and GPU involved in the processing. G-Storm [6] extends Apache Storm [17], a popular streaming engine, with support to GPUs. G-Storm introduces the concept of GPU-Bolt, a general-purpose streaming operator able to offload the processing on a GPU device through a micro-batch processing.

III. Parallel Programming Models

We present in a nutshell the parallel programming models used to implement stream parallelism and further discuss their challenges on multi-core machines equipped wth GPUs.

A. FastFlow

FastFlow is an emergent parallel programming framework created in 2009 by researchers at the University of Pisa and University of Turin in Italy. It provides stream parallel abstractions from an algorithmic skeleton perspective. The implementation is built on top of efficient fine grain lock-free communication queues. During the last three years new features were integrated for high-level parallel programming for data parallel patterns (parallel "for", Macro DataFlow, stencil and pool evolution). Also, other architectures have been supported such as clusters and hardware accelerators (GPU, FPGA and DSPs) [4].

The FastFlow programming interface provides a C++ template library whose classes can be viewed as a set of building blocks. Although built for general-purpose parallel programming, it provides suitable building blocks to exploit stream-oriented parallelism in streaming applications that other frameworks do not. For instance, it gives more freedom to the programmer to compose different parallel patterns and build complex communication topologies in shared memory systems. Also, the runtime support can operate in the blocking and non-blocking mode and enables the programmer to attach their customized task scheduler.

B. Threading Building Block (TBB)

TBB (Threading Building Blocks) is an Intel tool for parallel programming. TBB is a library for implementing high-performance applications in standard C++ without requiring a special compiler for shared memory systems. It emphasizes scalable and data parallel programming. The benefit is to completely abstract the concept of threads by using tasks. TBB builds on C++ templates to offer common parallel patterns (map, scan, parallel_for, among others), equipped with a work stealing scheduler, which dynamically dequeues a stack of tasks implemented in a FIFO-like order [8].

TBB and FastFlow are quite similar in many aspects, but the runtime and programming interface approaches are different regarding the design patterns and algorithmic skeleton. In fact, the pipeline pattern is supported in both of them which allows TBB to support stream parallelism exploitation. Although its scheduler has been proven to achieve good performance in several applications, TBB's runtime does not allow one to attach a customized scheduler. Another drawback is that it only targets multi-core systems. Moreover, the performance and compatibility with GPU parallel programming models like OpenCL and CUDA is unknown.

C. SPar

SPar[1] is a Domain-Specific Language (DSL) focused on expressing stream parallelism [3], [18]. The main drivers

[1] SPar's home page: https://gmap.pucrs.br/spar

behind SPar are: (a) optimize programmer productivity by not requiring sequential code rewriting to exploit parallelism; and (b) offer efficient programming abstractions to avoid the need for the programmer to work on low-level or architecture dependent code. In SPar, the parallelism is expressed by means of C++ annotations, which are part of C++ Standard since 2011 [19]. There are 5 attributes in the SPar language, 2 of which are identifiers (`ToStream` and `Stage`) and 3 are auxiliary (`Input`, `Output` and `Replicate`). These attributes are parsed by the SPar compiler and source-to-source transformations are performed to produce parallel code with FastFlow library calls.

The `ToStream` attribute identifies the region on which stream parallelism should be employed, and must contain at least one `Stage` annotation. This one identifies a computing phase, analogous to a assembly line. The annotations `Input` and `Output` are used to specify the variables that represent stream items and other data needed on the stream region and the stages. Eventually, `Replicate` specifies that the stage has no internal state and multiple copies can be run in parallel, and therefore it increases the stage parallelism degree (*i.e.*, worker replica number). Listing 1 presents the Mandelbrot Streaming pseudo application annotated with SPar. We explain later in Section IV.

D. CUDA

The CUDA architecture boosted GPGPU, bringing in many professionals interested on harvesting the GPU computing power [9]. Using the CUDA C language, combined with the `nvcc` compiler, developers can define kernels to be called from CPU and executed on the GPU by means of the `__global__` declaration. Calling those kernels involves providing the three-dimensional parallelism degree in a special syntax `<<<...>>>` between kernel's name and its parameters. This call launches blocks of threads on the GPU, executing the kernel function. Each thread has its own identifier, which can be obtained inside the kernel through the `threadIdx`, `blockIdx` e `blockDim` special variables [20].

E. OpenCL

OpenCL aims to provide code portability among architectures and hardware designs [10]. Instead on focusing on GPU programming, OpenCL proposes a generic API to use all OpenCL-capable devices on computation. Therefore, the proposed [21] workflow for a OpenCL program consist of these steps: 1) discover the components on the heterogeneous system, devices and their characteristics; 2) create kernels that will run on the devices; 3) manage the devices and host' memories to ensure that the data needed by the computation are available; 4) execute the kernels and collect the results. OpenCL devices has work-items organized into work-groups, both of them also in a three-dimensional space. The thread's global identifier is obtained calling `get_global_id` function.

IV. Parallel Stream Processing Applications

In this section, we first used the Mandelbrot Streaming application to highlight the main challenges. Then, we applied the same techniques on a robust application.

A. Mandelbrot Streaming

The Mandelbrot set consists of all points in a complex plane that do not tend to infinity when iterating the function $z \leftarrow z^2 + p$, where p is a number that gives the position of the point on the complex plane. As it is difficult to prove that a point will not tend to infinity, the function is calculated up to a given maximum number of iterations (`niter`). If p leaves a circle of radius 2, it will certainly diverge [7]. When computing coordinates of numbers that are in the Mandelbrot set, it is necessary to compute all the `niter` iterations, while outside coordinates diverge and quickly leave the circle of radius 2 [11]. In turn, this is an additional challenge to get performance improvements on GPUs because minimize divergence among threads of the same warp is an important concern.

The Mandelbrot set is commonly plotted in the form of a fractal image where row and column positions represent the numbers. In this paper, we approach the Mandelbrot set streaming application, which processes each line of the fractal as a distinct stream item, thus introducing the possibilty to get partial results while computing. The pixel color is used to identify if the corresponding point belongs to the Mandelbrot set.

Listing 1 presents the Mandelbrot streaming algorithm annotated with SPar. The `ToStream` attribute in line 3 delimits the streaming region. Lines 4 and 5 are responsible for the stream management and for providing workload to the next stages. These lines represent the first stage of the pipeline. The first `Stage` annotation (line 6) is followed by the `Replicate` auxiliary attribute, which represents the parallelism degree of this stage. The second and last `Stage` attribute appears in line 22 and collects the results of the previous stage, showing the calculated line of the fractal image of the Mandelbrot in the screen. The `Input` and `Output` attributes are used throughout the code to provide information on data items that flows from one stage to another.

```
1  void mandelbrot(int dim, int niter, double init_a, double
       init_b, double range) {
2    double step = range/((double)dim);
3    [[spar::ToStream, spar::Input(dim, init_a, init_b, step,
       niter)]]
4    for(int i=0; i<dim; i++) {
5      double im = init_b + (step * i);
6      [[spar::Stage, spar::Input(i, im, dim, init_a, step,
        niter, img), spar::Replicate(workers)]]
7      for (int j=0; j<dim; j++) {
8        double cr;
9        double a = cr = init_a + step * j;
10       double b = im;
11       int k = 0;
12       for (k=0; k<niter; k++) {
13         double a2 = a * a;
14         double b2 = b * b;
15         if ((a2+b2) > 4.0) break;
16         b = 2 * a * b + im;
17         a = a2 - b2 + cr;
18       }
19       img[j] = (unsigned char) 255-((k*255/niter));
20     }
21
22     [[spar::Stage, spar::Input(img, dim, i)]] {
23       ShowLine(img,dim,i);
24     }
25   }
26 }
```

Listing 1. Mandelbrot Streaming with SPar.

The Mandelbrot Streaming pseudo application was also implemented in TBB and FastFlow using the pipeline parallel pattern. Each stream item in the pipeline represents a single line of the fractal image. The first stage allocates memory and sends a stream item for each line of the fractal image. The middle stage runs in parallel and performs the calculation of each pixel in the row, sending computed data to the last stage. To increase the degree of parallelism we used `tbb::filter::parallel` in the class constructor for TBB, and a vector of instances of the stage class in FastFlow with the Farm pattern. The last stage is responsible for showing the image and deleting unused memory.

Porting the Mandelbrot set application to GPUs seems to present no special challenges to a programmer with knowledge on CUDA and OpenCL. The logical way to provide GPU parallelism support is to offload the computation of each pixel in the row, mapping each thread index to a specific column of the row. In Listing 1, this is done by removing the `for` loop in line 7 and assigning each iteration of `j` to a given GPU thread. For each kernel invocation, we also copy the data back from the device memory to the host memory.

However, when testing this implementation for the machine described in Section V with a fractal image of 2000x2000 and 200,000 iterations for each number, CUDA and OpenCL versions present the same speedup of 3.1× (129s) for a single GPU with respect to the sequential time (400s). This is much worse than the 17× speedup from the CPU parallel version with 20 threads.

Another option is to organize the execution grid of threads and blocks to use more dimensions. However, when using 2D of threads and blocks, it presents even worse results. For instance, 1.6× (250s) with respect to the sequential time as shows Fig. 1. The bars in Fig. 1 represent the execution time in log scale and are related to the left Y-axis. The right Y-axis represents the speedup in times with respect to the sequential version, which is plotted as a red dotted line. The standard deviation is represented by black error-bars and is mostly negligible.

When profiling the application, we find out that the large number of launched kernels with small workloads impacts on the performance, as the GPU is not fully utilized. To provide enough workload for the GPU, we designed batches of lines for each stream iteration. Listing 2 presents the Mandelbrot Streaming CUDA kernel function modified to process with batches in each kernel call. After obtaining the thread's global identifier in line 2, it calculates which is the fractal image line number within the batch so that the thread knows the corresponding memory address space (`i_batch` in line 3). This line number is positioned within the entire fractal image (`i` in line 4). Thus, each thread will calculate a single column of this line (`j` in line 5).

This kernel can also be easily ported to OpenCL. The only modifications required is to configure `threadIdGlobal` as the result of a call to `get_global_id(0)`, exchanging the `__global__` keyword to `__kernel`, and adding the `__global` keyword to the `img` parameter variable.

```
1  __global__ void mandel_kernel(int batch, int batch_size,
       int dim, double init_a, double init_b, double step,
       int niter, unsigned char *img) {
2    int threadIdGlobal = blockIdx.x * blockDim.x +
       threadIdx.x;
3    int i_batch = floor((double)threadIdGlobal/dim);
4    int i = batch * batch_size + i_batch;
5    int j = threadIdGlobal - i_batch*dim;
6    if (i < dim && j < dim) {
7      double im = init_b+(step*i);
8      double cr;
9      double a = cr = init_a+step*j;
10     double b = im;
11     int k = 0;
12     for (k = 0; k < niter; k++) {
13       double a2 = a*a;
14       double b2 = b*b;
15       if ((a2+b2) > 4.0) break;
16       b = 2*a*b+im;
17       a = a2-b2+cr;
18     }
19     img[i_batch*dim+j] = (unsigned char)255-((k*255 /
         niter));
20   }
21 }
```

Listing 2. Mandelbrot Streaming with batch processing.

Since the Titan XP GPU (compute capability 6.1) supports 2,048 active threads per streaming multiprocessor (SM) and have 30 SMs, it has up to 61,440 resident threads across the entire board. Consider also that each line of the fractal image represents 2,000 numbers to calculate. To fully utilize the GPU capabilities, we need to process 30.7 lines on each kernel call, provided that this amount of threads does not fill up the amount of registers (64,000) and shared memory (96 KB) available on each multiprocessor. The compiler tells us that the kernel function in Listing 2 uses only 18 registers, thus it is not a limiting factor for achieving maximum GPU utilization. Observe that we are not using shared memory. Moreover, by sending batches of 32 lines to the kernel function, we can achieve 44× speedup (9.1s) using OpenCL and 45× speedup (8.9s) using CUDA compared to the sequential version as presented in Fig. 1.

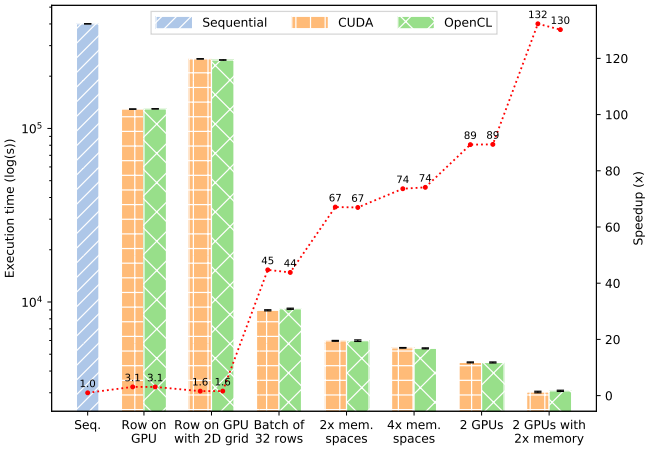

Fig. 1. Optimizing Mandelbrot Streaming application.

Further increasing the application performance is possible by overlapping data transfer and computations in the GPU device. In a single CPU thread, this is done by using asyn-

chronous memory copies and page-locked memory. To manage dependencies between memory copies and kernel function calls in CUDA, we used multiple `cudaStream`. In OpenCL, we used multiple `cl_command_queue` and `cl_event`. However, this customization doubles the memory requirements of the application since we need to allocate at least 2 memory spaces: one for copying data and another to perform computations. It yields $67\times$ speedup (5.98s) compared to the sequential version in CUDA and OpenCL.

We can allocate more than 2 memory spaces and use them in a round-robin fashion to push the performance a little further. Using $4\times$ more memory than the sequential version it is possible to obtain $74\times$ speedup (5.4s) using both CUDA and OpenCL. Allocating more memory spaces does not provide performance improvements.

These memory spaces can be easily assigned to different GPUs in a round-robin scheduling to enable multi-GPU support. Using two Titan XP GPU with a single memory space to each one (thus increasing by $2\times$ the host memory requirements with respect to the sequential version) provides $89\times$ speedup (4.48s). Assigning 2 memory spaces to each GPU (using $4\times$ more host memory and $2\times$ more memory on each device) provides speedups of $132\times$ (3.02s) with CUDA and $130\times$ (3.07s) with OpenCL. Fig. 1 shows the impact of each optimization, presenting the execution times and speedup.

It is worth noting that we focused our investigations in the data structures and task distribution inherent to streaming applications. Therefore, we do not discussed algorithm-specific optimization for better use shared memory and cache levels by implementing different data access patterns. Although the Mandelbrot set is a simple code, efficiently using the GPU always requires a significant parallel programming effort. The main challenges faced are related to the management of multiple asynchronous events on the GPUs, and to provide enough workload to the devices using a single CPU thread.

The integration of multi-core libraries with GPU parallel code also present unique challenges. The `cl_kernel` objects of OpenCL library are not thread-safe [10] and must be allocated for each thread. In the multi-core framework codes using OpenCL, we put a `cl_kernel` and a `cl_command_queue` object on each stream item, which are allocated by the first stage of the pipeline. Additionally, the middle stage invokes an asynchronous memory copy to transfer data from device to host after computation and send a `cl_event` to the last stage. Then, the last stage calls a `clWaitForEvents` to wait for the memory transfer to complete.

In CUDA, the `cudaSetDevice` function also has thread-side effects, thus, it must be called after initializing each thread. With respect to the implementations combining multi-core (SPar, TBB, and FastFlow) with CUDA, we put a `cudaStream` object for each stream item to properly define dependencies between data transfer and kernel function calls. The middle stage also invokes an asynchronous memory copy before sending the stream item to the last stage. The last stage uses `cudaStreamSynchronize` to ensure that the data transfer is completed. There were no special challenges for the CPU-only implementations as the Mandelbrot set is a relatively small code. The main challenges when combining multi-core (SPar, TBB, or FastFlow) with GPU (CUDA or OpenCL) parallel code were identifying non-thread-safe objects and correctly managing them. For the SPar with GPU support, we implemented the CUDA and OpenCL code along with the SPar annotations. All source codes are available online[2].

B. Dedup

The parallel implementation for Dedup with SPar was based on [22], which was adapted from the sequential version of PARSEC's Benchmark Suite [23]. That SPar implementation produces a pipeline with 3 stages. The first stage is responsible for the fragmentation of data based on rabin fingerprint algorithm to create batches and later perform the compression. The second stage performs SHA-1 hashing of blocks, which is checking duplicated blocks based on SHA-1 hash and is performing block compressing when necessary. The third stage reorders data and write in the output. For the SPar implementation, the second stage was replicated.

For our implementation on GPU[3], we changed the fragmentation method. Instead of using Dedup's rabin fingerprint that generates different batch sizes, we made it to generate fixed batch sizes (1MB) and generate different blocks sizes with rabin fingerprint. This modification was necessary to best benefit from GPU capabilities when a large batch of data has to process. In order to still benefit from the rabin fingerprint, we ran the algorithm on CPU and saved all the indexes where the algorithm would fragment the data. These indexes were used on all the stages to guarantee the equivalence with the original implementation.

In Fig. 2, we illustrate how batch and block are processed in this application. The batch consists of data chunks where `startPos` (processed by rabin fingerprint) are the indexes of the block for each batch. These indexes are necessary for the Dedup algorithm find duplicated data.

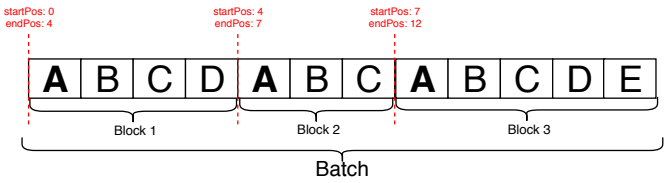

Fig. 2. Batch for LZSS.

Moreover, instead of using the originally PARSEC's compression algorithms (Bzip2 and Gzip), we used the LZSS compression algorithm as we already implemented it on GPUs in [24]. Observe that the integration has other challenges that will be discussed later. It also requires a new parallel algorithm implementation. The SPar annotation schema was also changed due to the GPU support implementation. Fig. 3 demonstrate the produced pipeline with five stages, which

[2]https://github.com/larcc-group/mandel-gpu-stream-parallelism
[3]https://github.com/larcc-group/dedup-gpu-stream-parallelism

previously was with only 3 stages. Therefore, each one of the stages perform as follows:

1) on CPU, this stage read the input file and generate batches of 1MB. Over these batches, it run the rabin fingerprint algorithm and generate blocks based on the indexes of rabin fingerprint. The batches and the blocks are sent to the next stage;
2) this stage transfers the blocks to GPU and generates SHA-1 hashes. Our strategy was that each GPU thread calculates the SHA-1 of one block. The result is saved in an array and sent to the next stage. This stage is replicated as many as necessary to offload computation to the GPUs available;
3) it checks if blocks in the batch are duplicated and send the results to the next stage;
4) it compress every not duplicated blocks on GPU. This stage reuses data already on GPU to prevent unnecessary data transfers;
5) it reorders the batches and writes it in the output.

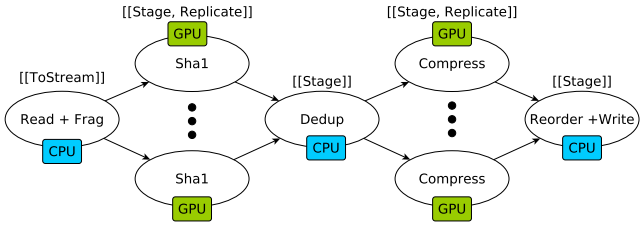

Fig. 3. Dedup parallel activity graph using SPar with CUDA/OpenCL.

Since we achieved very good results when parallelizing LZSS in our previous work [24], we followed the same strategy when integrating into Dedup. However, after running experiments, the performance achieved was very poor in Dedup (see Fig. 5). Then, we find out that the problem was actually in the LZSS parallel implementation. The GPU kernel function has been invoked for too many times without using efficiently the GPU resources. Therefore, we focused on optimizing the number of times that the GPU kernel function was called to avoid this overhead. In [24], this kernel is called `FindMatch`. It is responsible for finding the longest match for each element in the block being compressed. Our challenge in this implementation was to use the same batch and blocks from Dedup to run the compression in every block at once. To achieve a good performance, we rebuild this kernel running all the `FindMatch` operations in a single kernel function, considering the `startPos` presented in Fig. 2.

Our new parallel implementation of the LZSS method accepted as argument the input batch and the vector with the `startPos` presented in Fig. 2. In Listing 3, we already provide the optimized kernel for `FindMatch`, processing all the blocks inside a batch at once. As we can not use bidimensional vectors on GPU, we first need to find the block (start and end position) that has the character being searched based on the `startPos` (lines 4 to 10). The `FindMatch` operation uses `startPos` and `lastPos` to guarantee that the search operation is limited to the block of the element being compressed. In CPU, we used the result of the kernel function to run the compression on each block and generate the compressed data. This improved the performance because the GPU kernel functions invoked have enough workload to compute now.

```
1  __global__ void FindMatchKernel(unsigned char *input, int
       sizeInput, int* startPoss, int startPosSize,int*
       matchesLength, int* matchesOffset){
2    int idX = blockIdx.x * blockDim.x + threadIdx.x;
3    if(idX >= sizeInput) return;
4    int startPos = 0;
5    int foundAt = 0;
6    for(int k = 0; k < startPosSize; k++){
7      startPos = startPoss[k] < idX + 1 ? startPoss[k] :
         startPos;
8      foundAt = startPoss[k] < idX + 1 ? k : foundAt;
9    }
10   int lastPos =foundAt == breakSize - 1 ? sizeInput:
       startPoss[foundAt+1];
11   int lengthPos = lastPos-startPos;
12   unsigned char* uncodedLookahead = input + idX;
13   int thisBatchI = idX - startPos ;
14   int longestLength = 0;
15   int longestOffset = 0;
16   for(int current = max(thisBatchI - WINDOW_SIZE,0);
       current < thisBatchI; current++ ) {
17     if(current+ startPos < sizeInput && input[current+
         startPos] == uncodedLookahead[0]) {
18       int j = 1;
19       while(lastPos > current+ startPos + j && current +
         j< thisBatchI && current+ startPos + j < sizeInput
20          && idX + j < sizeInput
21          //limits the uncoded lookahead
22          && idX + j < lastPos
23          //find until start of uncodedLookahead
24          && input[current+ startPos + j]  ==
       uncodedLookahead[j]
25       ){
26         if (j >= MAX_CODED) break;
27         j++;
28       }
29       if(j > longest_length) {
30         longestLength = j;
31         longestOffset = Wrap(current, WINDOW_SIZE);
32       }
33     }
34   }
35   matchesOffset[idX] = longestOffset;
36   matchesLength[idX] = longestLength;
37 }
```

Listing 3. LZSS find match in batch.

As we discussed in Section IV-A, implementing multi-GPU support using a single CPU thread involves a lot of code refactoring, thus, we chose for not implementing it with CUDA and OpenCL single threaded version. Our previous optimization for Dedup already increased the performance, but we also implementing memory overlap (identified as 2x mem. spaces in the plots). We followed Mandelbrot set's strategy using 2 streams to process and copy data in Dedup for each GPU device. Thus, creating multiple `cudaStreams` for CUDA and `command_queues` for OpenCL.

We faced three main challenges in this application for implementing stream parallelism targeting multi-core equipped with GPUs. The first challenge was to re-factor the code to best benefit from GPU. This was the most time-consuming task because the application was originally implemented for CPU. The constraints in GPU are different, which required the aggregation of a larger load in order to be worth when invoking a kernel function. Another big challenge was to optimize the LZSS algorithm to achieve good performance even for small

block sizes. The last main challenge was to fix bugs regarding the parallel implementation on GPUs. Most of the errors are not intuitive when dealing with stream parallelism mechanism such as queue, batch, and memory space management. In the other hand, the use of SPar for multi-core is quite easy, simple, and productive. Such an abstraction is highly demanded for GPUs to increase coding productivity.

V. Experiments

The experiments were carried out on a machine that has a Intel(R) Core(TM) I9-7900X @ 3.3GHz (10 cores and 20 threads), 32GB of RAM memory and two Titan XP GPUs with compute capability 6.1 and each one has 12GB of memory. The system was running on Ubuntu OS (kernel 4.15.0-43-generic). All programs were compiled using -O3 compiler flags. The software used were G++ 7.3, NVCC 10.0.130, OpenCL 1.2, SPar, TBB, and FastFlow. Arithmetic means and standard deviations are computed over 10 samples.

A. Mandelbrot Streaming

Fig. 4 presents the execution time means and speedup computed over the sequential version. The CPU-only experiments were performed using 19 workers for the middle stage of the pipeline while the multi-core with GPU versions used 10 workers. In TBB, we set the `max_number_of_live_tokens` for CPU-only versions on 38 tokens (2×19 workers) and for TBB with GPU versions we used 50 tokens (5×10 workers). The GPU-only versions (CUDA and OpenCL) ran with 4× more memory per GPU, as detailed in Section IV-A. These are the best configurations and were chosen by empirical testing the applications under different configurations.

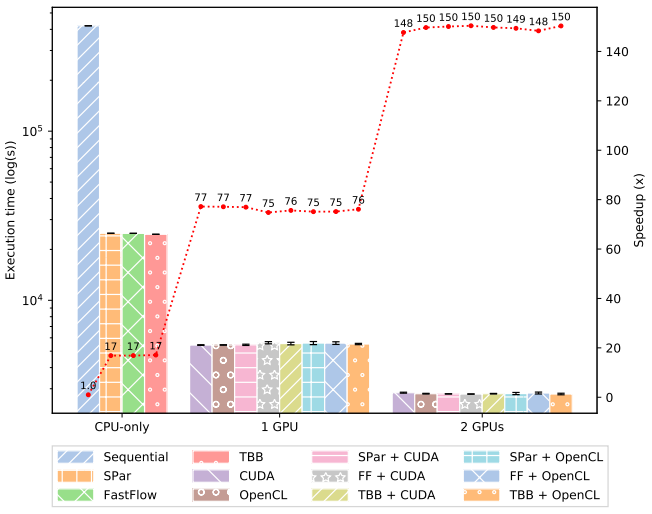

Fig. 4. Mandelbrot results.

We obtained very similar performance results using SPar, TBB, FastFlow, CUDA, and OpenCL in this application with a scalable performance. With a closer look when using a single GPU, we can observe that SPar with CUDA achieved the same performance as CUDA and OpenCL solely as well as better than other programming models and their combination with OpenCL and CUDA. When using two GPUs, the single thread on GPU degrades the performance since combining SPar, TBB, or FastFlow with CUDA increases the performance. Moreover, each programming model requires a very different programming effort to implement the Mandelbrot Streaming application efficiently. For instance, TBB and FastFlow requires to implement every stage of the pipeline in a different class, refactoring the sequential code. Additionally, TBB required fine-tuning the number of live tokens to obtain a speedup similar to FastFlow and SPar.

B. Dedup

We used 3 datasets to evaluate the performance in Dedup's parallel versions: 1) Input Large: Dataset used for PARSEC Benchmark Suite to test the Dedup application, consisting of 185MB; 2) Linux[4]: extracted from linux kernel source code with 816MB; 3) Silesia[5]: is a corpus of data that represents real-world files (XML, DLLs, and many others) with 202.13MB. All the tests were repeated 5 times for each sample to compute the throughput average and standard deviation. In SPar, we fixed the number of replicas in 19 for CPU only as well as single and multi-GPU.

In Fig. 5, the parallel versions with and without the batch processing optimization and the use of 2× memory spaces (see Section IV-B) are plotted. The versions with batch optimization increased significantly the throughput. The best results were achieved combining SPar with CUDA in all input datasets. It is important to highlight that we had to reduce the batch size for OpenCL because the number of items being processed resulted in a out of memory error. Therefore, in OpenCL and CUDA versions we used batches of 1MB instead of a batch with 10MB. We also observed that the optimization of 2× memory space version increased performance for OpenCL. However, it was not the case for CUDA. We could not cover this limitation because CUDA stream when performing asynchronous memory copies needs to deal with page-locked memory allocations. Dedup uses `realloc` in a memory buffer, which is not supported by CUDA as well as requires several memory movements to work with page-locked memory.

VI. Conclusions

This paper discussed the main challenges related to the parallel implementation of stream processing applications on multi-core systems equipped with GPUs. The main contributions are from the one hand a novel parallel implementation of the PARSEC's Dedup application and from the other side a thorough evaluation of the performance obtained by different parallel programming models. The results demonstrated that the full exploitation of the available parallelism of current heterogeneous multi-core systems is quite challenging. For the Dedup application, we observed that the programming model offered by the SPar provides suitable and productive

[4]Available in https://www.kernel.org/

[5]Available in http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia

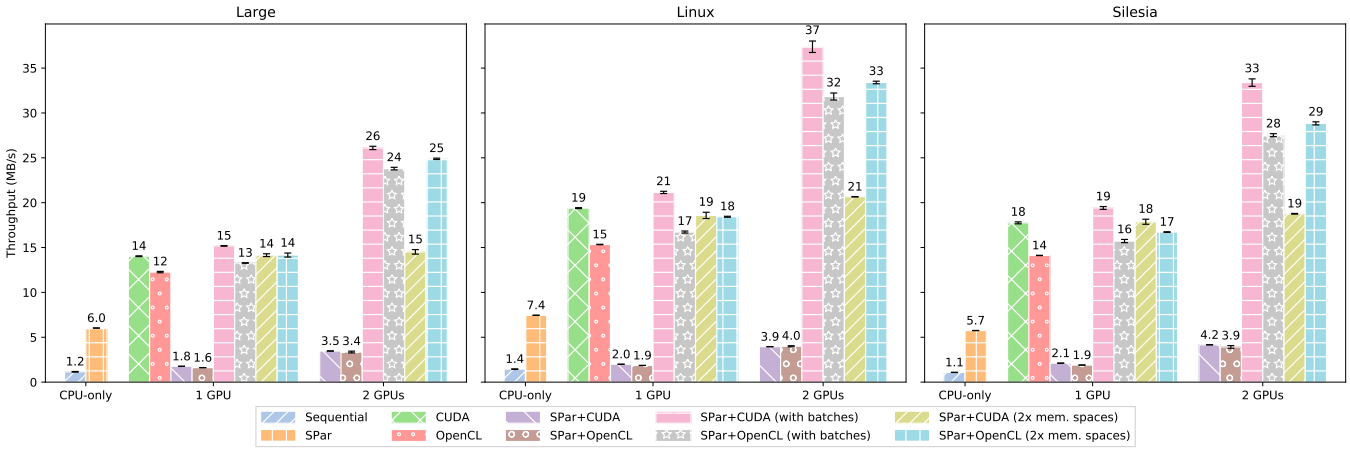

Fig. 5. Dedup results.

abstractions allowing to obtain performance comparable with state-of-the-art programming models as well as performance compatibility with CUDA and OpenCL when targeting GPUs.

As future work, we intend to automatically generate parallel OpenCL and CUDA code through the SPar compilation toolchain. This should further increase the parallel programming productivity when targeting heterogeneous multi-core systems. Also, we plan to explore the opportunities for designing automatic GPU memory optimizations to enhance performance.

Acknowledgment

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, Univ. of Pisa PRA_2018_66 DECLware, and by the FAPERGS 01/2017-ARD project ParaElastic (No. 17/2551-0000871-5). We thank Nvidia's GPU Grant program for the GPUs donation.

References

[1] W. Thies and S. Amarasinghe, "An Empirical Characterization of Stream Programs and Its Implications for Language and Compiler Design," in *International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. Austria: ACM, Sep 2010, pp. 365–376.

[2] H. C. M. Andrade, B. Gedik, and D. S. Turaga, *Fundamentals of Stream Processing*. New York, USA: Cambridge University Press, 2014.

[3] D. Griebler, M. Danelutto, M. Torquati, and L. G. Fernandes, "SPar: A DSL for High-Level and Productive Stream Parallelism," *Parallel Processing Letters*, vol. 27, no. 01, p. 1740005, March 2017.

[4] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "FastFlow: High-Level and Efficient Streaming on Multi-core," in *Programming Multi-core and Many-core Computing Systems*, ser. PDC, vol. 1. Wiley, 2014, p. 14.

[5] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch, "Saber: Window-based hybrid stream processing for heterogeneous architectures," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16, 2016, pp. 555–569.

[6] Z. Chen, J. Xu, J. Tang, K. Kwiat, and C. Kamhoua, "G-Storm: GPU-enabled high-throughput online data processing in Storm," in *IEEE International Conference on Big Data*, Oct 2015, pp. 307–312.

[7] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier Science, 2012.

[8] J. Reinders, *Intel Threading Building Blocks*. USA: O'Reilly, 2007.

[9] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2010.

[10] *The OpenCL Specification*, The Khronos Group, Oct 2018, v2.2-8.

[11] A. R. Brodtkorb and T. R. Hagen, "A Comparison of Three Commodity-Level Parallel Architectures: Multi-core CPU, Cell BE and GPU," in *Mathematical Methods for Curves and Surfaces. MMCS 2008*, M. Dæhlen, M. Floater, T. Lyche, J.-L. Merrien, K. Mørken, and L. L. Schumaker, Eds., vol. 5862. Tønsberg, Norway: Springer, Berlin, Heidelberg, 2010, pp. 70–80, lecture Notes in Computer Science.

[12] J. Guo, J. Thiyagalingam, and S.-B. Scholz, "Breaking the GPU Programming Barrier with the Auto-parallelising SAC Compiler," in *Proceedings of the 6th Workshop on Declarative Aspects of Multicore Programming*, ser. DAMP '11. ACM, jan 2011, pp. 15–24.

[13] M. Bourgoin, E. Chailloux, and J.-L. Lamotte, "SPOC: GPGPU Programming Through Stream Processing With Ocaml," *Parallel Processing Letters*, vol. 22, no. 2, p. 1240007, 2012.

[14] M. Steuwer, P. Kegel, and S. Gorlatch, "SkelCL - A portable skeleton library for high-level GPU programming," in *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE, may 2011, pp. 1176–1182.

[15] K. Suttisirikul and P. Uthayopas, "Accelerating the Cloud Backup Using GPU Based Data Deduplication," in *International Conference on Parallel and Distributed Systems (ICPADS)*, Dec 2012, pp. 766–769.

[16] P. Bhatotia and R. Rodrigues, "Shredder: GPU-Accelerated Incremental Storage and Computation," in *USENIX Conference of File and Storage technologies (FAST)*, Feb 2012.

[17] J. Leibiusky, G. Eisbruch, and D. Simonassi, *Getting Started with Storm*. O'Reilly Media, Inc., 2012.

[18] D. Griebler, "Domain-Specific Language & Support Tools for High-Level Stream Parallelism," Ph.D. dissertation, Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil, June 2016.

[19] ISO/IEC, *ISO/IEC 14882:2017 - Programming languages – C++*, 5th ed., International Organization for Standardization, Geneva, Switzerland, dec 2017, https://www.iso.org/standard/68564.html.

[20] D. B. Kirk and W. mei W. Hwu, *Programming Massively Parallel Processors*. Morgan Kaufmann, 2013.

[21] A. Munshi, B. R. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg, *OpenCL Programming Guide*. Addison-Wesley, 2012.

[22] D. Griebler, R. B. Hoffmann, M. Danelutto, and L. G. Fernandes, "High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2," *International Journal of Parallel Programming*, pp. 1–19, February 2018. [Online]. Available: https://doi.org/10.1007/s10766-018-0558-x

[23] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[24] C. M. Stein, D. Griebler, M. Danelutto, and L. G. Fernandes, "Stream Parallelism on the LZSS Data Compression Application for Multi-Cores with GPUs," in *27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. Pavia, Italy: IEEE, February 2019.