An Enhanced Profiling Framework for the Analysis and Development of Parallel Primitives for GPUs

Nicola Bombieri
Dept. of Computer Science
University of Verona
Italy
Email: nicola.bombieri@univr.it

Federico Busato
Dept. of Computer Science
University of Verona
Italy
Email: federico.busato@univr.it

Franco Fummi
Dept. of Computer Science
University of Verona
Italy
Email: franco.fummi@univr.it

***Abstract*—Parallelizing software applications through the use of existing optimized primitives is a common trend that mediates the complexity of manual parallelization and the use of less efficient directive-based programming models. Parallel primitive libraries allow software engineers to map any sequential code to a target many-core architecture by identifying the most computational intensive code sections and mapping them into one ore more existing primitives. On the other hand, the spreading of such a primitive-based programming model and the different GPU architectures have led to a large and increasing number of third-party libraries, which often provide different implementations of the same primitive, each one optimized for a specific architecture. From the developer point of view, this moves the actual problem of parallelizing the software application to selecting, among the several implementations, the most efficient primitives for the target platform. This paper presents a profiling framework for GPU primitives, which allows measuring the implementation quality of a given primitive by considering the target architecture characteristics. The framework collects the information provided by a standard GPU profiler and combines them into optimization criteria. The criteria evaluations are weighed to distinguish the impact of each optimization on the overall quality of the primitive implementation. The paper shows how the tuning of the different weights has been conducted through the analysis of five of the most widespread existing primitive libraries and how the framework has been eventually applied to improve the implementation performance of a standard primitive.**

I. Introduction

Computing platforms have evolved dramatically over the last years. Because of the physical limitations imposed by thermal and power requirements, frequency scaling has proven to be no longer the solution to increase the performance of processors. As a consequence, many hardware manufacturers have turned to scale the number of cores in a processor in order to boost application performance. Apart from the significantly improved simultaneous multithreading capabilities, such heterogeneous multicore platforms also contain general-purpose graphic processing units (GPUs) to exploit fine-grained parallelism [1]. As a result of such hardware trends, the heterogeneity of these platforms and the need to program them efficiently has led to a spread of parallel programming models, such as CUDA and OpenCL.

On the other hand, while many software developers possess a working knowledge of basic programming concepts, they typically lack of expertise in developing efficient parallel programs in a short time. As a matter of fact, the programming process with a CUDA or OpenCL-based environment is much more complicated and time-consuming than that with a parallel programming environment for conventional multiprocessor systems. Programmability of such parallel platforms is consequently a strategic factor impacting on the approach feasibility as well as costs and quality of the final product.

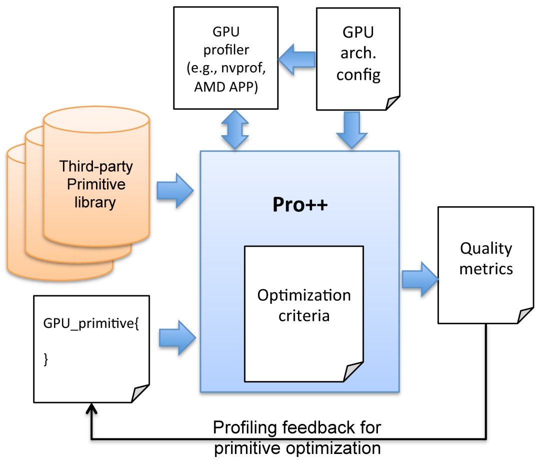

GPU profiler (e.g., nvprof, AMD APP)
GPU arch. config
Third-party Primitive library
Pro++
Optimization criteria
Quality metrics
GPU_primitive{
}
Profiling feedback for primitive optimization

Figure 1: Overview of the Pro++ framework

In this context, directive-based extensions to existing high-level languages (OpenACC[2], OpenHMPP[3]) have been proposed to help software engineers through sets of directives (annotations) for marking up the code regions intended for execution on a GPU. Based on this information, the compiler generates hybrid executable binary. Despite their user-friendliness and expressiveness, such directive-based solutions require notable effort from the developers in organizing correct and efficient computations and, above all, compilers are often over conservative, thus leading to poor performance gain by the parallelization process [4].

Domain-specific languages (DSLs) (e.g., Delite[5], Spiral[6]) have been also proposed to express the application parallelism for GPUs in specific problem domains. DSL-based approaches allow the language features and the specific problem domain features to be brought closer and, at the same time, the parallel applications to be developed not strictly customized for any particular hardware platform. Nevertheless, these solutions require the user to implement the algorithms by using a proprietary language, with consequent limitations to SW IP reuse and portability.

A more user-friendly and common trend is to implement the application algorithm through existing primitives for GPUs. This generally provides sound trade-off between parallelization costs and code performance. Such primitive-based programming model relies on identifying parts of code computationally intensive and re-implementing their functionality through one or more basic primitives provided by an existing library. Due to its efficiency, the primitive-based programming model has been recently also combined to both directive-based solutions and DSLs [7] to exploit the portability of annotations/DSLs as well the performance provided by the GPU primitives.

An immediate consequence of such a trend has been the spreading of an extensive list of accelerated, high-performance libraries of primitives for GPUs ([8] and [9] are some exampling collections of them). On the one hand, all these libraries cover a wide spectrum of use cases, such as basic linear algebra, machine learning, and graph applications. On the other hand, many libraries provide different implementations of the same primitives. From the developer point of view, this moves the actual problem of parallelizing a software application to selecting the most efficient primitives for the target platform among several implementations.

The motivation for our work is precisely the observation that it would be nice to measure the implementation quality of a given primitive, with the aim of helping the software developer (i) to choose the best implementation of a given primitive among different libraries, and (ii) to understand whether such a primitive implementation fully exploits the architecture characteristics and how the implementation efficiency could be improved. To do that, this paper presents *Pro++* (see Figure 1), an enhanced profiling framework for the analysis and the optimization of parallel primitives for GPUs. *Pro++* collects the information about a given primitive implementation (i.e., profiling metrics) through a standard GPU profiler. The framework combines the standard metrics into *optimization criteria*, such as, multiprocessor occupancy, load balancing, minimization of synchronization overheads, and memory hierarchy use. The criteria are evaluated, weighed, and finally merged into an overall measure of quality metrics. The quality metrics allows the user to classify and compare the different implementations of a primitive in terms of performance over the selected GPU architecture configuration.

The main contributions of this work are the following:

- A classification of optimization criteria that mainly impact on the primitive performance.
- An analysis of such optimization criteria over five different primitive libraries for GPUs to weigh the impact of each single criterion on the overall primitive performance.
- A framework that combines profiling metrics, optimization criteria, and weights to provide (i) an overall quality metrics of a given primitive and (ii) profiling feedbacks to improve the primitive implementation.

The paper is organized as follows. Section II summarizes the key concepts of CUDA, GPU architectures and GPU profiling. Section III presents the optimization criteria by which the primitives are evaluated. Section IV reports the analysis conducted to measure the impact of the optimization criteria on the overall quality metrics of primitives. Section V presents the case study of Pro++ application while Section VI is devote to the conclusions.

II. Background on CUDA, GPUs and Profiler Metrics

Computed Unified Device Architecture (CUDA) is a parallel computing platform and programming model proposed by NVIDIA. CUDA comes with a software environment that allows developers to use C/C++ as a high-level programming language targeting heterogeneous computing on CPUs and GPUs. Through API function calls, called *kernels*, and language extensions, CUDA allows enabling and controlling the offload of compute-intensive routines. A CUDA kernel is executed by a *grid* of *thread blocks*. A thread block is a batch of threads that can cooperate and synchronize each other via shared memory, atomic operations and barriers. Blocks can execute in any order while threads in different blocks cannot directly cooperate.

Groups of 32 threads with consecutive indexes within a block are called *warps*. A thread warp executes in SIMD-like way the same instruction on different data concurrently. In a warp, the synchronization is implicit since the threads execute in lockstep. Different warps within a block can synchronize through fast barrier primitives. In contrast, there is no native thread synchronization among different blocks as the CUDA execution model requires independent block computation for scalability reasons. The lack of support for inter-block synchronization requires explicit synchronization with the host, which involves significant overhead.

A warp thread is called *active* if it successfully executes a particular instruction issued by the warp scheduling. A CUDA core achieves the full efficiency if all threads in a warp are active. Threads in the same warp stay idle (not active) if they follow different execution paths. In case of *branch divergence*, the core serializes the execution of the warp threads.

The GPU consists of an array of *Streaming Multiprocessors* (SMs), which, in turn, consist of many cores called *Stream Processors* (SPs). Each core is a basic processing element that executes warp instructions. Each SM has from one to four warp schedulers that issue the instructions from a given warp to the corresponding SIMD core. The hardware scheduler switches between warps with the aim of hiding the memory latency.

Each GPU core has a dedicated integer (ALU) and a floating point (FPU) data path that can be used in parallel. Both ALU and FPU can execute complex arithmetic instructions (e.g., multiplication, trigonometric functions, etc.) in one clock cycle. On the other hand, the SM has limited instruction throughput per clock cycle.

GPUs also feature a sophisticated memory hierarchy, which involves thread registers, shared memory, DRAM memory and two-level cache (L1 within a SP, while L2 accessible to all threads). In the last NVIDIA GPU architectures, Kepler and Maxwell, a small read-only cache per-SM (called Texture cache) is also available to reduce global memory data access.
Private variables of threads and local arrays with static indexing are placed into registers. Large local arrays and dynamic indexing arrays are stored in L1 and L2 cache. Thread variables that are not stored in registers are also called *local memory*. To fully exploit the memory bandwidth, multiple memory accesses of warp threads can be combined into single transactions (i.e., *coalesced memory access*).

EXTRACTED INFORMATION	INFORMATION SOURCE	DESCRIPTION
#SM	Hardware Info	Total number of stream multiprocessors.
block_size	Kernel Configuration	Number of threads per block associated to a kernel call.
grid_size	Kernel Configuration	Number of thread blocks associated to a kernel call.
#registers	Compiler Info	Number of used registers per thread associated to a kernel call.
Static_SMem	Compiler Info	Bytes of static shared memory per block.
Dynamic_SMem	Kernel Configuration	Bytes of dynamic shared memory per block.
active_warps	Profiler Event	Number of active warps per cycle per SM.
total_warps	Hardware Info	Maximum number of active warps per cycle per SM.
warps_launched	Profiler Event	Number of warps run on a multiprocessor.
threads_launched	Profiler Event	Number of threads run on a multiprocessor.
stall_sync	Profiler Event	Percentage of stalls occurring because the warp is blocked at a __syncthreads() call.
Int_instr, SP_instr, DP_instr	Profiler Event	Number of arithmetic instructions (integer, single-precision floating point, double-precision floatig point) executed by all threads.
arithmetic_throughput	Hardware Info	Maximum arithmetic instruction throughput calculated from maximum number of instructions per clock per SM, total number of SMs, and execution time.
local_accesses, shared_accesses texture_accesses, global_accesses	Profiler Event	Number of executed load/store instructions per warp on a SM where state space is specified as Local, Shared, Texure and Global.
cudacopy_size	Profiler Event	Number of bytes associated to a host-device memory transfer function.
DRAM_transactions	Profiler Event	Total number of DRAM memory accesses.
Available_mem_throughput	Hardware Info	Memory bandwidth calculated from DRAM clock frequency, bus width, number of memory interfaces.
kernel_start_time, cudacopy_start_time, kernel_time, cudacopy_time	Profiler Info	Start time and duration of a kernel call or CUDA memory transfer function.

TABLE I: Profiler events, compiler information, hardware (device) information, and kernel configuration considered in the proposed optimization criteria.

Finally, the host-GPU device communication bus allows overlapping CPU-GPU data transfers with the kernel computations to minimize the host-device data transfers.

A. Profiler Metrics

Developing high performance applications requires adopting tools for understanding the application behaviour and for analysing the corresponding performance. At the state of the art, there exist several profiling tools for GPU applications that provide advanced profiling information through the analysis of *events*, kernel configuration, hardware and compiler information. Table III summarizes a selected list of such profiling information, which are strongly related with the application performance.

In this work we refer to the NVIDIA *nvprof* profiler terminology and information. However, the proposed methodology is independent from the adopted profiler. *Nvprof* has two operating modes that generate two distinct outputs. The first mode is the *trace mode*, which provides a timeline of all activities taking place on the GPU in chronological order. From this mode, we extract the kernel configuration and any timing associated to a kernel (e.g., start time, latency, etc.). The second mode, called *summary mode*, reports a user-specified set of events for each kernel, both aggregating values across the GPU units and showing the individual counter for each SM.

III. OPTIMIZATION CRITERIA

We define different *optimization criteria*, which express the quality of a given primitive to exploit a GPU characteristic. Examples are the occupancy of all the computing (SP) resources, the load balancing, and the memory coalescing. The selection of the most representative and influential optimization criteria has been guided by the best practices guide [10], by the main CUDA books [11] [12] and by our programming experience [13]. The criteria are defined in terms of events and static information, which can be measured through the profiling phase. Each criterion value is expressed in the range [0, 1], where 1 indicates the maximum and 0 the worst optimization of such a criterion.

A. Occupancy (OC)

In order to take advantage of the computational power of the GPU, it is important to maximize the SP utilization of each SM. This criterion gives information on the maximum theoretical occupancy of the GPU multiprocessors in terms of active threads over the maximum number of threads that may concurrently run on the device.

The criterion value that is calculated statically, depends on the kernel configuration as well as on the kernel implementation. In particular, it depends on the block size (i.e., number of threads per block), grid size (i.e., number of blocks per kernel) as well as amount of used shared memory for the kernel variables, and number of used registers. In general, the kernel configuration of the primitives is set at compile time by exploiting information on the device compute capability and no tuning is allowed to the user (to comply to the principle of user-friendliness). The criterion takes into account how well the limited resources like registers and shared memory have been exploited in the kernel implementation and, thus, how and how many variables have been declared (e.g., automatic and shared). A low value means underutilization of the GPU multiprocessors. More in details, the overall occupancy is calculated as the minimum value between the occupancy related to block_size (taking into account also the maximum number of block per SM), to shared memory utilization (StaticalSMem + DynamicSMem), to the register utilization (#registers), and to the grid_size with respect to the minimum number of blocks required to keep busy all SMs:

$$\text{Reg\_OCC} = \left\lfloor \frac{\frac{block\_size}{32} \cdot \lceil 32 \cdot \#registers \rceil^{[256]}}{SM\_\text{Register}} \right\rfloor$$

$$\text{Block\_OCC} = \max \left(max\_SM\_blocks, \left\lfloor \frac{SM\_threads}{\lceil block\_size \rceil^{[32]}} \right\rfloor \right)$$

$$\text{Thread\_OCC} = \frac{grid\_size \cdot \lceil block\_size \rceil^{[32]}}{\#resident\_threads}$$

$$OCC = \min\left(\text{Reg\_OCC}, \text{Block\_OCC}, Thread\_OCC, 1\right)$$

B. Load Balancing (LB)

During the GPU execution it is crucial to avoid the situation in which a subset of SPs is doing most of the work while others are in the idle state. The load balancing criterion expresses how well the workload is uniformly distributed over the SPs, as follows:

$$LB = \frac{\sum_{\#SM} \frac{active\_warps}{total\_warps}}{\#SM}$$

C. Warp Efficiency (WE)

This criterion gives information on the thread divergence of the warps and, thus, to the quality of the kernel implementation.

$$\text{WE} = \frac{threads\_launched}{warps\_launched}$$

In complex parallel code, thread divergence is almost unavoidable. Nevertheless, a low value of this criterion outlines a wrong control flow logic, which does not take into account SIMD nature of the GPU warps. A low warp efficiency value indicates that the code execution is serialized, which directly translates in performance degradation.

D. Synchronization Overhead (SO)

The synchronization overhead takes into account the total time spent by the primitive for synchronizing threads and thread blocks. The criterion consists of two weighted values as follows:

$$SO = \frac{\text{stall\_sync}}{\text{kernel\_time}} \cdot W_1 + \frac{\text{Kernel\_synch\_time}}{\text{Total\_computation\_time}} \cdot W_2$$

The first takes into account the amount of time spent by the threads in the waiting state as a consequence of a thread barrier, for each thread barrier in the kernel. It depends on the load balancing among threads as well as the number of synchronization points (i.e. thread barriers) in the kernel. It is provided by the profiler as percentage of GPU time spent in synchronization stalls. The second value considers the amount of time spent by the cuda runtime to coordinate two or more kernel calls that compose a primitive. A fragmented computation that involves many kernels and many small data transfers is penalized with a low value of the second part of the formula. The *kernel_time* value is computed as the sum of the times spent on the host between the first kernel call (*start_time*) and the last kernel call (*start_time + duration*) of a tested primitive.

E. Instruction Optimization (IO)

This criterion takes into account the amount of instructions that make use of arithmetic units, both integer and floating point over the whole number of instructions run in the primitive execution:

$$IO = 1 - \frac{\text{Int\_instr} + \text{SP\_instr} + \text{DP\_instr}}{\text{arithmetic\_throughput}}$$

This criterion is strictly related to the code optimization. A high value of instruction optimization criterion means that the ALU and FPU units have not been wasted. Considering also the same functionality of all tested code for the same primitive, this information indicates how much the code is optimized. Some simple examples are the use of *shift* instead *multiplication* instructions, template arguments to create constant expressions, etc.

F. Memory Hierarchy (MH)

This criterion measures how much the memory hierarchy has been exploited in the primitive implementation. We take care of all memory spaces, by providing an indicative measure of how much the fast on-chip memories are used against the slow off-chip DRAM memory:

$$MH = \frac{\text{texture} + \text{local} + \text{shared accesses}}{\text{texture+local+shared+global accesses}}$$

Fast on-chip memories allow carefully controlling spatial data-locality of a parallel application. Customizing specific applications to fully exploit the memory hierarchy is very important to take advantage of the limited GPU resources. On the other hand, it is also the most complex phase of parallel programming. A low value of this criterion suggests restructuring the code and reorganizing the data through techniques such as tiling and problem partition to improve the performance. It does not take into account cache memory, since cache accesses is not fully under the user's control. Efficacy on cache use is included in memory coalescing.

G. Memory Coalescing (MC)

This criterion measures the efficiency of the primitive to exploit the bus bandwidth. Achieving high memory bandwidth, and thus high application throughput, requires a high level of concurrency and memory access pattern that allows coalescing:

$$MC = \frac{\frac{\sum \text{DRAM\_transactions}}{\text{kernel\_time}}}{\text{Available\_mem\_throughput}}$$

The coalescing control in GPUs is hardware-implemented and relies on the use of L1 cache memory. The memory coalescing is the key to reduce the overhead involved by DRAM memory latency.

H. Data Transfer (DT)

It takes gives a quality measure of the primitive to address the data transfer overhead. As an example, pipelining (overlapping) between data transfer and data computation allows the primitive to rich higher value of this criterion:

$$DT = 0.5 + \frac{Overlapping\_mem\_transf}{problem\_size + \sum cudacopy\_size} - \frac{\sum cudacopy\_size}{problem\_size + \sum cudacopy\_size}$$

Overlapping data transfers with kernel computation may reduce the execution time, but it requires a fine-tuning of the data size to be transferred. Too large data sizes may involve no advantage, while too small sizes may involve heavy synchronization overhead.

It also takes into account the amount of bytes transferred in the host-device communication during a kernel computation over the actual I/O bytes required for the computation. Any extra data transfer between host and device is considered as overhead.

I. Overall Quality Metrics (QM)

All the proposed values of the optimization criteria are finally combined into an overall quality metric to provide, through a single value, an evaluation of the profiled code. We express this value as the weighted average of the values of the optimization criteria as follows:

$$\mathbf{QM} = \frac{\begin{array}{c}\mathrm{OC} \cdot \mathrm{W_{OC}} + \mathrm{LB} \cdot \mathrm{W_{LB}} + \mathrm{WE} \cdot \mathrm{W_{WE}} + \mathrm{IO} \cdot \mathrm{W_{IO}} + \\ \mathrm{MH} \cdot \mathrm{W_{MH}} + \mathrm{MC} \cdot \mathrm{W_{MC}} + \mathrm{DT} \cdot \mathrm{W_{DT}}\end{array}}{\mathrm{W_{OC}} + \mathrm{W_{LB}} + \mathrm{W_{WE}} + \mathrm{W_{IO}} + \mathrm{W_{MH}} + \mathrm{W_{MC}} + \mathrm{W_{DT}}}$$

W_{xy} express the weight of each single criterion in the overall quality measure. In this work, we tuned the different weights through the analysis of different libraries of primitive, as detailed in the following section.

IV. Weighing of Optimization Criteria on the Overall Quality Metrics

The impact of the optimization criteria classified in the previous section on the overall quality metrics has been measured through the analysis of five primitive libraries for NVIDIA GPU architectures. The first library, *Thrust*[14], is provided by NVIDIA in the CUDA Toolkit and it is based on the C++ Standard Template Library high-level interface. This library provides a wide range of parallel primitives to simplify the parallelization of fundamental parallel algorithms such as *scan*, *sort*, and *reduction*. The second library, *CUB*[15], provides a set of high performance parallel primitives for generic programming for both host and device programming layer. The third library, *CUDPP*[16], focuses on common data-parallel algorithms such as *reduction* and *prefix-scan*, and includes also a set of specific-domain primitives such as compression and suffix array functions. The fourth library, *ModernGPU* (MGPU) [17], implements basic primitives such as *reduction* and *prefix-scan* but the main goal of ModernGPU is providing very efficient implementations of *parallel binary search* algorithm applications such as *segmented reduction/prefix-scan*, *load balancing* algorithm, *merge*, *set operations* and matrix-vector multiplication. Finally, *ArrayFire* [18] includes hundreds of high performance parallel computing functions. In particular, it is focused on complex algorithms across various domains such image processing, computer vision, signal processing and linear algebra. In *ArrayFire*, the common parallel primitives are proposed as vector algorithms.

These libraries have been selected as they provide different implementations of widely used and common primitives for the parallelization of fundamental algorithms. This allowed us to compare such implementations by running them over several datasets and by measuring their actual speedups w.r.t. a reference sequential implementation. The comparison results has been finally used to heuristically tune the weight of each optimization criteria in the overall quality metrics.

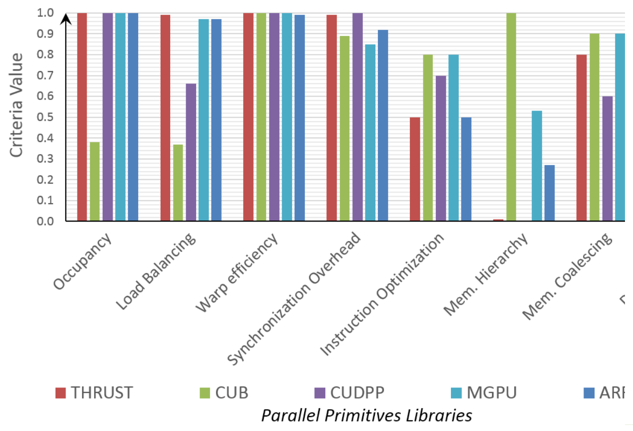
Table II summarizes the parallel primitives that have been evaluated for such a tuning, by specifying which libraries provide an implementation of a specific primitive. The primitives are grouped by similar functionality in seven main classes. The most basic primitives implementing data elaboration are grouped in the *Independent Linear Transformation* class, which applies concurrent operations on every single element of the input data. This class includes primitives implementing predicate functions for linear transformation on subsets of the input data as well as on multiple sets of data concurrently. The second class, *Advanced Copying*, includes two classic collective operations, i.e., *gathering* and *scattering*, as well as their version with *predicate*. The *Reduction* class refers to all the primitives that apply an operation to the input data and that return a single value as result (e.g., counting, maximum, reduction). The segmented version of the reduction applies the operation on a subset of input data. The fourth class includes all the variants (i.e., inclusive, exclusive, etc.) of the *prefix-scan* procedure, which represents the building blocks of many parallel algorithms. The *search* class contains primitives for searching elements in sorted or unsorted sets of data. The *load-balancing* primitives are a specialization of the *vectorized sorted search*. They are largely used to extrapolate, from a given input data, the indices to map threads to the corresponding input elements. The primitives in the *Reordering* class include different procedures to manipulate the input data or to select a subset of such a data by using predicates. Finally, the *Set* class covers the most common operations on sets represented as continuous sorted data values.

For all the parallel primitives, we firstly measured the value of each optimization criterion as proposed in Section III. Figure 2 reports, as an example, the values of the optimization criteria of the *reduction* and *inclusive prefix-scan* primitives. The evaluation of all the primitives has been run on two different systems: a NVIDIA Kepler GeForce GTX 780 device with CUDA Toolkit 6.0, AMD Phenom II X6 1055T 3GHz host processor, and Debian 3.2.60 OS and a NVIDIA Fermi GeForce GTX 570 device with CUDA Toolkit 6.5, AMD FX-4100 1.4 GHz host processor, and Debian 3.2.0 OS.

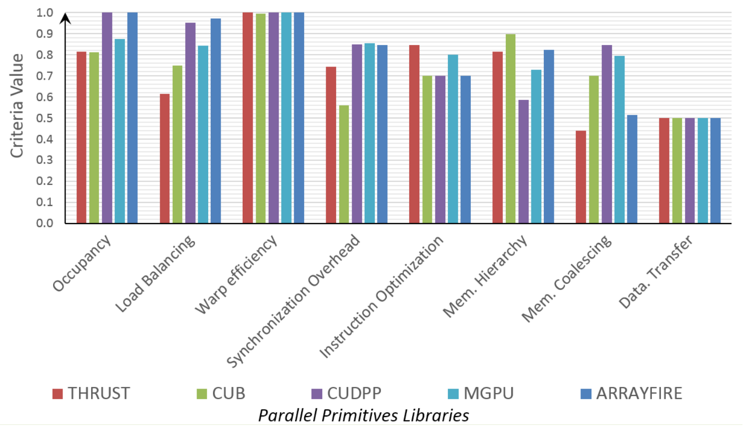
The dataset applied for the evaluation consists of a large sets of random generated input data. The figure shows that the *Warp efficiency* criterion reaches almost the maximum value for all the implementations of the five libraries of both the primitives. This is due to the fact that both the *reduction* and the *inclusive prefix-scan* implement a highly regular computation on the input data, which does not cause thread divergence. The different implementations of the reduction also show a high value of the *Synchronization overhead* criterion. This is due to two main reasons. First, the reduction

Parallel Primitives		**Library** Thrust	CUB	CUPDD	MGPU	ArrayFire
Independent Linear Transformation	Fill/Generate/Sequence/Tabulate	X				X
	Modify/Transform/Replace/Adjacent Difference	X				X
	Modify_If	X				
	Comparison	X				
	Simple Copy	X				
Advanced Coping	Gathering	X				
	Gathering_If	X				
	Scattering	X				
	Scattering_If	X				
Reduction	Couting	X				X
	Extrema	X	X			X
	Reduction	X	X	X	X	X
	Reduce_by_keys/Segmented_Reduction	X	X		X	X
	Histogram		X			X
Prefix-Scan	Inclusive	X	X	X	X	X
	Exclusive	X	X	X	X	X
	Prefixscan_By_Key/Segmented_Prefixscan	X		X		
Search	Unsorted Search/Find	X				
	Vectorized Binary Search	X			X	
	Load-Balancing Search				X	
Reordering	Partitioning/Partitioning_If	X	X			
	Compaction/Copy_If/Select	X	X	X		
	Merge	X			X	
	Merge Sort				X	X
	Radix Sort	X	X	X		
Set (ordered)	Union	X			X	X
	Intersection	X			X	X
	Set Difference	X			X	
	Unique	X	X			X

TABLE II: Parallel primitives evaluated for the weight tuning



(a) Reduction



(b) Inclusive Prefix-Scan

Figure 2: Optimization criteria evaluation of the *reduction* and *inclusive prefix-scan* primitives

Parallel Primitives	Quality metrics value ([0, 1])					GPU/CPU Sim. speedup				
	Thrust	**CUB**	**CUDPP**	**MGPU**	**ArrayFire**	**Thrust**	**CUB**	**CUDPP**	**MGPU**	**ArrayFire**
Reduction	0.50	0.89	0.41	0.75	0.64	202	505	163	252	233
Reduction_by _Keys	0.55	0.67		0.65	err	13	101		40	err
Inclusive PrefixScan	0.70	0.77	0.76	0.79	0.74	64	190	119	301	62
Prefix-Scan_by_Keys	0.58		0.69			36		128		
Vectorized_Binary _Search	0.30			0.55		687			3576	
Partition	0.65	0.71				15	87			
Compaction	0.66	0.69	0.67			19	67	34		
RadixSort	0.57	0.55				22	45			
Unique	0.69	0.68			err	15	88			err

TABLE III: Quality metrics values obtained with $W_{OC} = 30$; $W_{LB} = 30$; $W_{WE} = 15$; $W_{SO} = 10$; $W_{IO} = 70$; $W_{MH} = 100$; $W_{MC} = 100$; $W_{DT} = 50$ and the corresponding actual GPU vs. CPU simulation speedup.

primitives have been implemented, in all the libraries, by a single kernel function and, second, they have been implemented with few barriers and highly balanced threads. In contrast, the *prefix-scan* primitive has been implemented, in all the evaluated libraries, through a two-phase algorithm and a kernel per phase. This involves synchronization overhead between the two kernels invocations.

The impact of each criteria on the overall quality metrics value has been weighed by considering the criteria values and the actual CPU vs. GPU speedup of each single primitive obtained during simulation. The tuning has been performed with the aim of obtaining the quality metrics value of each primitive implementation linearly proportional to the actual CPU vs. GPU speedup of such an implementation. The weight values of the optimization criteria are calculated through a multi-variable regression analysis between all information returned by the different criteria and the execution time. Since the weights depend on the actual architecture, our future work aims at automating such a weight computation. The idea is to define a software framework based on a collection of primitives to be run on the target architecture and that automatically extrapolates the weight values.

Table III reports some of the most meaningful obtained results. The table reports the weights of the optimization criterion extrapolated during simulation, the corresponding quality metrics values and the actual CPU vs. GPU simulation speedup of each parallel primitive. The results show how, given the weights reported in the table caption, the values of the overall quality metrics reflect the actual simulation speedup. The performance accuracy of our model is with 10%-15%, as shown in the experimental results All the other results, which have not been reported in the table for the sake of brevity, show the same correlation.

From the results reported in Figure 2, it is possible to compare different implementations of a given primitive in terms of performance and to understand which characteristics of such implementations lead to the corresponding speedup. As an example, the *CUB* library provides the best implementation of the *reduction* primitive even though such an implementation presents low *occupancy* and low *load balancing*. On the other hand, the code has been implemented by fully exploiting *memory coalescing* and *memory hierarchy*, whose criteria values have more impact in the overall quality metrics. The *reduction* primitive implemented in *CUDPP* shows a *load balancing* value much lower than the *occupancy* value. This underlines that the warp workloads during the primitive execution are not uniform. Another example is the very different values of *memory hierarchy* and *memory coalescing* criteria obtained with the *prefix-scan* primitive of *Thrust*. A high value of memory hierarchy, that indicates a correct local reorganization of data, should also imply a good memory coalescing.

This analysis allows us to understand whether, given a primitive implementation, there is room to improve such an implementation and how. We applied the proposed profiling framework to analyse and improve the implementation of a *load balancing search*, as explained in the following section.

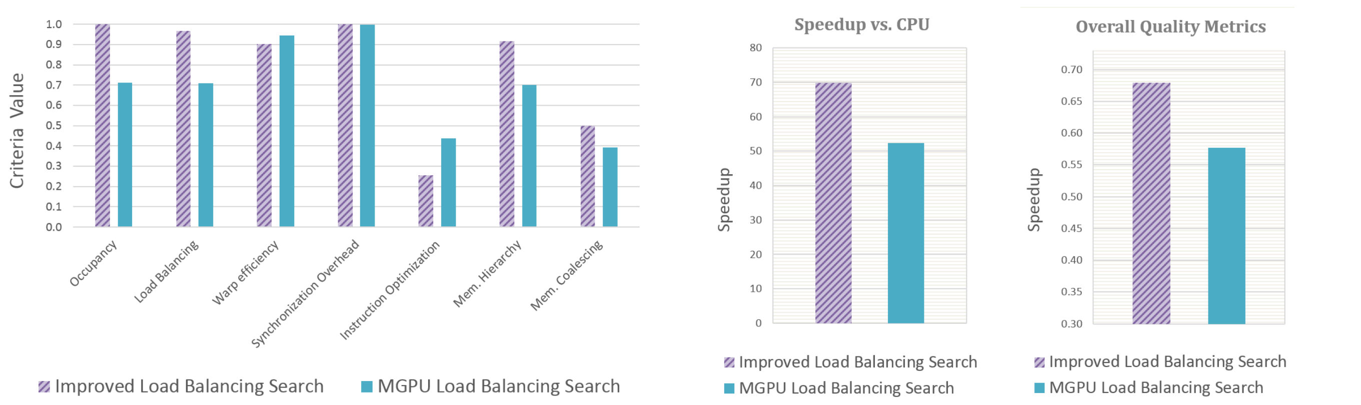
V. CASE STUDY: THE *Load Balancing Search* PRIMITIVE

The *load balancing search* is a special case of *vectorized sorted search* (i.e., binary search). It is commonly applied as auxiliary function to uniformly partition irregular problems. Given a set of input values that represent the problem workload, the primitive generates a set of indices for mapping threads to the corresponding input elements.

Among the libraries evaluated in this work, only *MGPU* provides an implementation of the *load balancing search* primitive. We applied *Pro++* to such a primitive to calculate the optimization critera values, the CPU/GPU simulation speedup, and the overall quality metrics value by considering the weights proposed in Section IV (Table III). Figure 3 reports the results (*MGPU* columns). Then, starting from the *MGPU* implementation, we optimized the code by exploiting the profiling information with the aim of improving the CPU vs. GPU simulation speedup.

Considering the different optimization criteria weights, we started from the analysis of the *Memory hierarchy* and *Memory coalescing* criteria values. To improve these values, we modified the code to better organize the data in shared memory, registers and texture memory. Such a modification led to a better organization of the data in local memory, which also simplified the management of the memory accesses and allowed us to improve the memory coalescing among threads. These first modifications of the code increased the memory hierarchy and memory coalescing criteria values from 0.7 to 0.9 and from 0.4 to 0.5, respectively. Further improving memory coalescing has been evaluated as a hard task, due to the many sparse global memory accesses that are closely related to the algorithm. Thus, it has not been further investigated.

On the other hand, improving the two memory criteria



(a) Optimization criteria values ([0 ,1]) (b) CPU vs. GPU sim. speedup (c) Quality metrics values ([0 ,1])

Figure 3: *Load balancing search* primitive evaluation

required the introduction of many extra control flow statements, which decreased, with respect to the original *MGPU* implementation, the value of the *Instruction optimization* criterion. Nevertheless, considering such a decrease and the the weight of the instruction optimization criterion, we didn't invest effort to limit such a side-effect.

Then, the analysis results underline the low value of the *Occupancy* criterion. To improve this criterion, we modified the code by improving the kernel configuration, the use of automatic variables (and thus the use of SM registers), and the allocation of shared memory. Beside an improvement on occupancy, these modifications had impact on the value of the *load balancing* criterion. This is due to the fact that the execution flows of all threads during the primitive execution take similar paths and, as a consequence, improving the occupancy criterion leads also to an improvement of the load balancing criterion. The modifications also slightly reduced the *Warp efficiency* value, which, on the other hand, still remains close to the maximum. As a consequence, any further investigation or modification of the code targeting warp efficiency would not be worth to improve the overall quality of the primitive implementation. The synchronization overhead criterion had the highest value, both in the original and the modified version of the code. Thus, no modifications on barriers or synchronization have been considered.

In conclusion, the use of Pro++ allowed us to improve the *loading balancing search* primitives by better concentrating the effort in those code optimizations with more room for improvement and, as a consequence, to save time. The case of study has shown how Pro++ framework has been applied to significant improve step-by-step, in the optimization cycle, the performance of the *load balancing search* exploting the suggested guideline on the optimization criteria.

VI. Conclusion

This paper presented *Pro++*, a profiling framework for GPU primitives that allows measuring the implementation quality of a given primitive. The paper showed how the framework collects the information provided by a standard GPU profiler and combines them into optimization criteria. The criteria evaluations are weighed to distinguish the impact of each optimization on the overall quality of the primitive implementation. The paper reported the analysis conducted on five among the most widespread existing primitive libraries to tune the different weights. Finally, the paper presented how the framework has been applied to improve the implementation performance of a standard primitive.

References

[1] "Hybrid System Architecture - HSA Foundation," http://www.hsafoundation.com.
[2] "OpenACC - Directives for Accelerators," http://www.openacc-standard.org/.
[3] D. R., B. S., and B. F., "Hmpp: A hybrid multicore parallel programming environment," 2007.
[4] M. Sugawara, S. Hirasawa, K. Komatsu, H. Takizawa, and H. Kobayashi, "A comparison of performance tunabilities between opencl and openacc," in *Proc. of the 2013 IEEE 7th International Symposium on Embedded Multicore/Manycore System-on-Chip (MCSOC'13)*, 2013, pp. 147–152.
[5] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A compiler architecture for performance-oriented embedded domain-specific languages," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4s, pp. 134:1–134:25, 2014.
[6] "Spiral - Software/Hardware Generation for DSP Algorithms," http://www.spiral.net/bench.html.
[7] W. Tan, W. Tang, R. Goh, S. Turner, and W. Wong, "A code generation framework for targeting optimized library calls for multiple platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, pp. 1–12, 2014.
[8] "NVIDIA CUDA ZONE - GPU-accelerated libraries," https://developer.nvidia.com/gpu-accelerated-libraries.
[9] "CLPP - OpenCL Parallel Primitives Library," http://gpgpu.org/2011/06/03/opencl-parallel-primitives-library.
[10] C. NVidia, "C best practices guide," *NVIDIA, Santa Clara, CA*, 2012.
[11] D. B. Kirk and W. H. Wen-mei, *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.
[12] J. Cheng, M. Grossman, and T. McKercher, *Professional Cuda C Programming*. John Wiley & Sons, 2014.
[13] F. Busato and N. Bombieri, "BFS-4K: an efficient implementation of BFS for kepler GPU architectures," *IEEE Transactions on Parallel Distributed Systems*, vol. preprint, no. 99, pp. 1–14, 2015.
[14] J. Hoberock and N. Bell, "Thrust: A parallel template library," 2014. [Online]. Available: http://thrust.github.io/
[15] D. Merrill, "Cub," 2015.
[16] M. Harris, J. Owens, S. Sengupta, Y. Zhang, and A. Davidson, "Cudpp: Cuda data parallel primitives library," 2014.
[17] S. Baxter, "Modern gpu," 2014.
[18] J. Malcolm, P. Yalamanchili, C. McClanahan, V. Venugopalakrishnan, K. Patel, and J. Melonakos, "Arrayfire: a gpu acceleration platform," 2014. [Online]. Available: http://arrayfire.com/