Pro++: A Profiling Framework for Primitive-based GPU Programming

Nicola Bombieri, *Member, IEEE,* Federico Busato, *Member, IEEE,* and Franco Fummi, *Member, IEEE*

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 *Pro++*, a profiling framework for GPU primitives that 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 two standard and widespread primitives.

Index Terms—GPUs, Performance model, Parallel applications.

1 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. In this context, many parallel applications frmo different context have been developed for GPUs, ranging from artificial intelligence [2], to electronics design automation [3], [4], [5].

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.

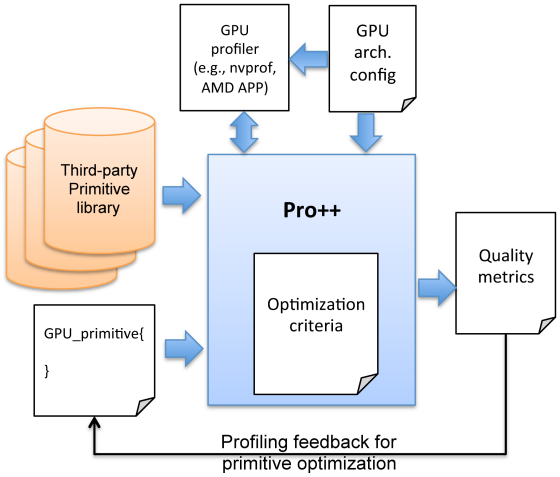


Fig. 1: Overview of the Pro++ framework

In this context, directive-based extensions to existing high-level languages (OpenACC[6], OpenHMPP[7]) 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 [8].

Domain-specific languages (DSLs) (e.g., Delite[9], Spiral[10]) have been also proposed to express the appli-

• *N. Bombieri, F. Busato, F. Fummi are with the Department of Computer Science, University of Verona, Italy.*
E-mail: name.surname@univr.it

cation 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 [11] 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 ([12] and [13] 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.

This article is based and extends the works presented in [14]. Compared to [14], this work presents:

- An extended and more detailed analysis of the state of the art, with a new section devoted to the related work;
- A revision and extension of the model (which is the core of the whole framework) proposed to measure the quality of the GPU implementations. In such a model, different optimization criteria have been optimized and several others have been added to better cover all the crucial properties a GPU application should satisfy to exploit the full potential of the GPU device;
- An extended experimental analysis, in which (i) we applied the new extended performance model to the existing case study, and (ii) we added a new case study (matrix transpose) to underline the robustness and efficacy of the proposed method.

The article is organized as follows. Section 2 presents the analysis of the state of the art. Section 3 summarizes the key concepts of CUDA, GPU architectures and GPU profiling. Section 4 presents the optimization criteria by which the primitives are evaluated. Section 5 reports the analysis conducted to measure the impact of the optimization criteria on the overall quality metrics of primitives. Section 6 presents the case studies of Pro++ application while Section 7 is devote to the conclusions.

2 Related Work

Different performance models for GPU architectures have been proposed in literature. They can be classified into *specific models*, which apply on a particular application or pattern only, and *general-purpose models*, which are applicable to any program/kernel for a comprehensive profiling [15].

In the class of specific models, [16] proposes an approach for performance analysis of *code regions* in CUDA kernels, while, in [17], the authors focus on profiling divergences in GPU applications. A different analytical approach is proposed in [18], which aims at predicting the kernel execution times of sparse matrix-vector multiplications.

General-purpose models allow profiling applications from more optimization criteria point of view and, thus, they can give different hints on how to optimize the code. As an example, [19] proposes a model for NVIDIA GPUs based on two different metrics: Memory warp parallelism (MWP) and computation warp parallelism (CWP). Although the model predicts the execution costs fairly well, the understanding of performance bottlenecks from the model is not so straightforward. This model has been extended in two different ways [20], [21]. [20] introduces two kernel behaviors, MAX and SUM, and shows how they allow generating predictions close to the real measurements. Nevertheless, they do not provide clues as how to choose the right one for

a given kernel. In contrast, [21] extends the model with additional metrics, such as cache effect and SFU instructions.

All these analytical performance models, although accurate in several cases, rely on simulators (e.g., Ocelot [22], GPGPU-Sim [23], Barra [24], Multi2Sim [25]) to collect necessary information for profiling, which implies a high overhead in the profiling phase. An attempt has been made in [26] for collecting more efficiently information on the GPU characteristics and using simple static analysis methods to reduce the overhead of runtime profiling.

Besides the often prohibitive overhead introduced in the profiling phase, especially for complex applications, a big problem of the simulator-based models is portability. They can be applied to profile applications on GPU models that are supported by the simulator, which, often, is not updated to the last releases of GPU models.

Differently from the analytical models, [27] and [28] are based on machine-learning techniques, which allow identifying hardware features and using feature selection, clustering and regression techniques to estimate the execution times. In particular, the work in [27] derives relationships between application characteristics and performance on GPU and CPU devices. The model relies on the Ocelot PTX simulator and on regression analysis to characterize benchmarks and to derive relationships to machine and application parameters. The authors in [28] propose a performance prediction model for OpenCL kernels for automatic selection at run-time of the best-suited accelerator for a specific computation. The work applies a linear regression model to identify kernel function parameters with a strong correlation with the application performance. Nevertheless, both these models are inaccurate, thus providing approximate estimations with high variability.

We propose a general-purpose performance model that, similarly to the machine-learning models, relies on a regression suite of parallel primitives to characterize the device and on several application criteria to measure the implementation quality, gives interpretable hints and accurate performance prediction.

3 Background on CUDA, GPUs and Profiler Metrics

3.1 Computed Unified Device Architecture (CUDA)

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.

3.2 Graphic Processing Unit (GPU)

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*).

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

3.3 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 1 summarizes a selected list of such profiling information, which are strongly related to the application performance.

EXTRACTED INFORMATION	INFORMATION SOURCE	DESCRIPTION
#SM	Hardware Info	Total number of stream multiprocessors.
#SM_threads	Hardware Info	Total number of threads per stream multiprocessor.
reg_granularity	Hardware Info	Register allocation granularity.
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.
SM_registers	Hardware Info	Total number of registers per Streaming Multiprocessor.
#Blocks_per_SM	Profiler Event	Number of resident blocks per Streaming Multiprocessor.
max_SM_blocks	Hardware Info	Maximum number of resident blocks per Streaming Multiprocessor.
#resident_threads	Hardware Info	Maximum number of threads that can run concurrently on the device.
Static_SMem	Compiler Info	Bytes of static shared memory per block.
Dynamic_SMem	Kernel Configuration	Bytes of dynamic shared memory per block.
SM_SMem	Hardware Info	Total available shared memory per Streaming Multiprocessor.
active_warps	Profiler Event	Number of active warps per cycle per SM.
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.
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.
#L1_transactions	Profiler Event	Total number of L1 memory accesses.
#L2_transactions	Profiler Event	Total number of L2 memory accesses.
#Mem_instr$_T$	Profiler Event	Total number of global memory instructions of size T. Where T can be $1/2/4/8/16$ bytes.
#SharedLoadTrans, #SharedStoreTrans	Profiler Event	Total number of shared memory load/store transactions.
#SharedLoadAcc, #SharedStoreAcc	Profiler Event	Total number of shared memory load/store accesses.
alu_utilization	Profiler Event	Utilization level of the GPU arithmetic units (ALU/FPU).
ld_st_utilization	Profiler Event	Utilization level of the GPU load/store instruction units.
KernelStart, cudacopy_start_time, KernelExeTime, cudacopy_time	Profiler Info	Start time and duration of a kernel call or CUDA memory transfer function.

TABLE 1: Profiler events, compiler information, hardware (device) information, and kernel configuration considered in the proposed optimization criteria.

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.

4 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 [29], by the main CUDA books [30] [31] and by our programming experience [32]. The optimization criteria are defined to cover all the crucial properties a GPU application should satisfy to exploit the full potential of the GPU device. We consider the properties adopted in [21], [16], [17], [19], [20] concerning divergence, memory coalescing, and load balancing. In addition, we define optimization criteria to cover synchronization issues. Differently from the literature, the proposed criteria are more accurate (i.e., fine-grained) to evaluate such properties. All the criteria are defined in terms of events and static information, which are all provided by any standard GPU profiler. Each criterion value is expressed in the range [0, 1], where 0 represents the worst and 1 represents the best evaluation of such an optimization.

4.1 Occupancy (OCC)

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, which 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 with 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 blocks 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 [1]:

$$\text{SMEM\_OCC} = \left\lfloor \frac{\text{SM\_SMem}}{\text{StaticSMem} + \text{DynamicSMem}} \right\rfloor$$

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

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

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

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

4.2 Host Synchronization (HSync)

Many complex parallel applications organize the compute-intensive work into several functions offloaded to GPUs through host-side kernel calls. Depending on the code complexity and on the workflow scheduling, this mechanism may involve significant overhead that can compromise the overall application performance. The host synchronization criterion aims at evaluating the amount of time spent to coordinate the kernel calls. It is defined as follows:

$$\text{HSync} = \frac{\sum_{\text{i}=1}^{\text{N}} \text{KernelExeTime}_\text{i}}{\text{KernelStart}_N + \text{KernelExeTime}_N - \text{KernelStart}_1}$$

1. The notation $\lceil A \rceil^{[B]}$ denotes the nearest multiple of B equal or greater than A.

where N is the number of kernels in which the application has been organized, $KernelExeTime_i$ is the real execution time of kernel i on the device, and $KernelStart_i$ is the clock time in which kernel i starts executing. A fragmented computation that involves many kernel invocations and many small data transfers is represented by a low value of this criterion.

This criterion helps programmers to understand if the overall application speedup is bounded by an excessive host synchronization activity. Merging different kernels, using inter-block synchronization [33] or reducing small memory transfers improve the quality value of this criterion.

4.3 Device Synchronization (DSync)

In GPU computing, the synchronizations of threads in blocks are one of the main causes of idle state and, thus, they strongly impact on the application performance. Beside introducing overhead in the kernel execution, they also limit the efficiency of the multiprocessors in the warp scheduling activity. This criterion gives a quality value of a kernel by measuring the total time spent by the kernel for synchronizing thread blocks:

$$\text{DSync} = 1 - \left(1 - \frac{\text{TotActiveWarps}/\text{warps\_size}}{\text{CLK\_cycles}} \right) \cdot \text{StallSync}$$

$$\text{where TotActiveWarps} = \sum_{\text{i}=1}^{\text{CLK\_cycles}} \text{ActiveWarps}_\text{i}.$$

$|Warps|$ represents the maximum number of thread warps of the device, while CLK_cycles represents the total number of GPU clock cycles elapsed to execute the kernel. The formula takes into account the number of active warps at each clock cycle, and it adds them to the total counter $TotActiveWarps$. The value in the round bracket represents the overall percentage of inactivity of the GPU warps (i.e., warps in waiting state). $StallSync$ represents the percentage of the GPU time spent in synchronization stalls over the total number of stalls. $StallSync$ depends on the load balancing among threads as well as the number of synchronization points (i.e. thread barriers) in the kernel.

4.4 Thread Divergence (TDiv)

Branch conditions that lead threads of the same warp to execute different paths (i.e., thread divergence) are one of the main causes of inefficiency of a GPU kernel. This criterion evaluates the thread divergence of a kernel as follows:

$$\text{TDiv} = \frac{\#\text{ExeInstructions}}{\#\text{PotExeInstructions}}$$

where $\#ExeInstructions$ represents the total number of instructions executed by the threads of a warp and $\#PotExeInstructions$ represents the total number of instructions potentially executable by the threads of a warp. The final value is calculated as the average over all warps run by the kernel. The criterion gives a clear evaluation of the branching factor of a kernel code.

4.5 Warp Load Balancing (LB_W)

This criterion expresses how well the workload is uniformly distributed over the cores of each single SM:

$$LB_W = \frac{\left(\frac{\text{TotActiveWarps}}{\text{TotActiveCycles}}\right)}{\left(\frac{\text{block\_size}}{\text{warp\_size}}\right) \cdot \#Blocks\_per\_SM}$$

where $TotActiveCycles$ represents the total number of clock cycles in which the single SMs are not in idle state. The formula takes into account the number of active warps at each clock cycle, and it adds them to the total counter $TotActiveWarps$. The denominator represents the theoretical maximum occupancy of the SMs in terms of number of warps. It is calculated by considering the block size and the number of blocks mapped to each single SM. A low value of this criterion underlines that some warps doing most of the work while others are in the idle state. This is a common behavior in irregular problems and suggests to programmers to choose a different load balancing strategy.

4.6 Streaming Multiprocessor Load Balancing (LB_{SM})

Besides the load balancing on each single SM, the model evaluates the load balancing at SM level. The SM load balancing criterion is defined as follows:

$$LB_{SM} = 1 - \frac{\max\limits_{SM}(\text{TotActiveCycles}) - \text{AvgCycles}}{\max\limits_{SM}(\text{TotActiveCycles})}$$

where

$$\text{AvgCycles} = \frac{\sum\limits_{SM} \text{TotActiveCycles}}{\text{\#SM}}$$

The formula expresses the SM Load Balancing as the difference between the maximum execution cycles required among all SMs and the best case where all SMs take the same execution cycles.

4.7 L1/L2 Granularity ($Gran_{L1}$/$Gran_{L2}$)

GPU applications require optimized data access patterns and properly aligned data structures to achieve high memory bandwidths. In particular, efficient applications hide the latency of memory accesses by combining multiple memory accesses into single *transactions* that match the granularity (i.e., the cache line size) of the memory space[2]. Hides latency of memory accesses in CUDA is feasible by combining multiple memory accesses into a single transaction that match the granularity (cache line size) of the memory space. The proposed performance model includes two complementary criteria to describe the quality of memory access patterns:

$$Gran_{L1} = \frac{\#\text{L1\_transactions} \cdot 128}{\sum\limits_{T \in \{Mem\_instr\}} \#\text{Mem\_instr}_\text{T} \cdot \text{size}_\text{T}}$$

$$Gran_{L2} = \frac{\#\text{L2\_transactions} \cdot 32}{\sum\limits_{T \in \{Mem\_instr\}} \#\text{Mem\_instr}_\text{T} \cdot \text{size}_\text{T}}$$

2. This concept applied to the L1 cache is also known as *memory coalescing*.

The criteria take into account the number of actual transactions towards the L1(L2) memory, the cache line size (128 Bytes for L1, 32 Bytes for L2), the total number of memory instructions (`load` and `store`) to access the global memory, and the size of their accesses $size_T$ (1/2/4/8/16 Bytes). The ratio of useful data accesses to total data accesses is calculated by comparing the total size of the data required by threads with the number of transactions multiplied by the respective memory granularity.

4.8 Shared Memory Efficiency ($SMem_{eff}$)

This criterion measures the kernel efficacy to exploit the *data locality* concept through the on-chip shared memory. The shared memory allows high memory bandwidth for concurrent accesses, but it requires appropriate access patterns to achieve the full efficiency. On the other hand, an excessive and disorganized use of the shared memory leads to bank conflicts, which involve the memory instructions to be re-executed thus serializing the thread execution flow. This optimization criterion is defined as follows:

$$\text{SMem}_\text{eff} = \frac{\text{\#SharedLoadTrans + \#SharedStoreTrans}}{\text{\#SharedLoadAcc + \#SharedStoreAcc}}$$

The formula is defined in terms of total number of transactions towards shared memory for both load and store operations over the total number of accesses in shared memory for load and store instructions (which includes the re-executed memory instructions due to bank conflicts).

4.9 Computation Intensity (CI)

This criterion takes into account the amount of instructions that make use of arithmetic units, both integer and floating point, and load-store (instruction) units (LDST) for all memory spaces.

$$CI = \frac{\text{alu\_utilization} + \text{ld\_st\_utilization}}{2}$$

The formula expresses the computation intensity as the average of the utilization level of the ALU/FPU units and the LDST units. This criterion is strictly related to the code optimization. A high value of computation intensity criterion means that the ALU, FPU and LDST 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. A high value of the utilization level of an instruction unit indicates that the unit performs a high number of independent operations of the same type. As a consequence, all the operations can be run in parallel and saturate the computational throughput of the specific unit.

4.10 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:

$$DT = 0.5 + \frac{\text{Overlapping\_mem\_transf}}{\text{GPU\_allocated\_byte} + \sum \text{cudacopy\_size}} - \frac{\sum \text{cudacopy\_size}}{\text{GPU\_allocated\_byte} + \sum \text{cudacopy\_size}}$$

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.

4.11 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:

$$\mathbf{QM} = \frac{\begin{array}{l}\text{OCC} \cdot \text{W}_{\text{OCC}} + \text{HSync} \cdot \text{W}_{\text{HSync}} + \\ \text{DSync} \cdot \text{W}_{\text{DSync}} + \text{TDiv} \cdot \text{W}_{\text{TDiv}} + \\ \text{Gran}_{\text{L1}} \cdot \text{W}_{Gran_{L1}} + \text{Gran}_{\text{L2}} \cdot \text{W}_{Gran_{L2}} + \\ \text{LB}_W \cdot \text{W}_{\text{LB}_W} + \text{LB}_{SM} \cdot \text{W}_{\text{LB}_{SM}} + \\ \text{SMem}_{\text{eff}} \cdot \text{W}_{\text{SMem}_{\text{eff}}} + CI \cdot W_{CI} + DT \cdot W_{DT}\end{array}}{\begin{array}{l}\text{W}_{\text{OCC}} + \text{W}_{\text{HSync}} + \text{W}_{\text{DSync}} + \text{W}_{\text{TDiv}} + \\ \text{W}_{Gran_{L1}} + \text{W}_{Gran_{L2}} + \text{W}_{\text{LB}_W} + \text{W}_{\text{LB}_{SM}} + \\ \text{W}_{\text{SMem}_{\text{eff}}} + W_{CI} + \text{W}_{\text{DT}}\end{array}}$$

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.

5 Weighing of Optimization Criteria on the Overall Quality Metrics

5.1 Parallel Primitives

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* v1.8.1 [34], 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* v1.4.1 [35], provides a set of high performance parallel primitives for generic programming for both host and device programming layer. The third library, *CUDPP* v2.2 [37], 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* v1.1 (MGPU) [38], 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* v3 [39] 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 tune the weight of each optimization criteria in the overall quality metrics.

Table 2 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 to a subset of input data. The fourth class includes all 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.

5.2 Evaluation

For all parallel primitives, we firstly measured the value of each optimization criterion as proposed in Section 4. The evaluation of all primitives has been run on two different systems: a NVIDIA Kepler GeForce GTX 780 device with CUDA Toolkit 7.5, AMD Phenom II X6 1055T 3GHz host processor, and Debian 7 OS and a NVIDIA Fermi GeForce

Parallel Primitives		Library				
		ArrayFire	CUB	CUPDD	MGPU	Thrust
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

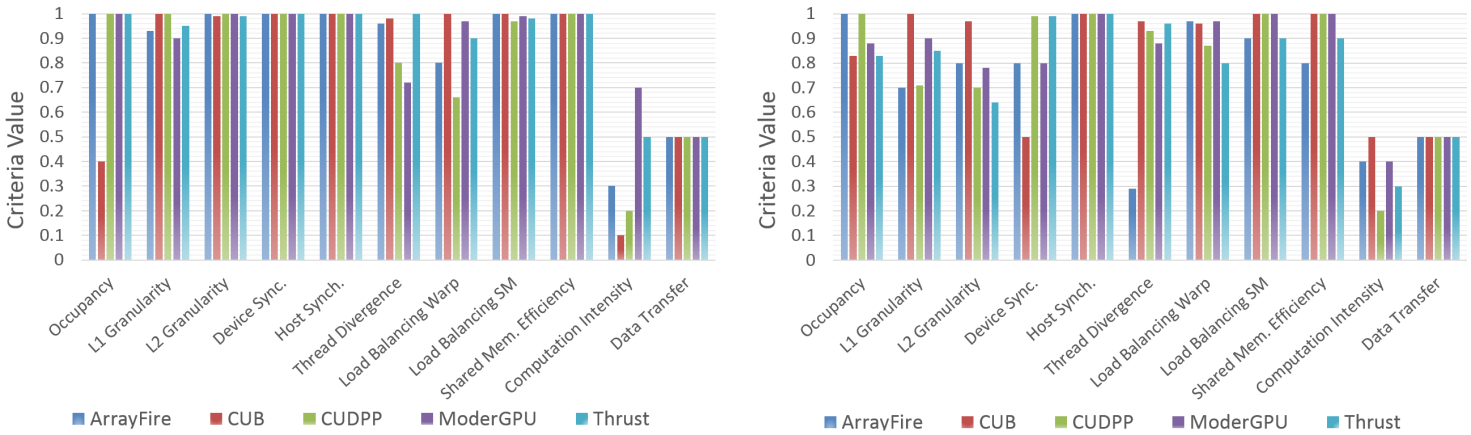
TABLE 2: Parallel primitives evaluated for the weight tuning. The table reports also the alternative names of primitives.

GTX 570 device with CUDA Toolkit 7.0, AMD FX-4100 1.4 GHz host processor, and Debian 7 OS. The dataset applied for the evaluation consists of a large set of random generated input data.

Figure 2 reports, as an example, the values of the optimization criteria of the *reduction* and *prefix-scan* primitives. The figure shows that the *Host Synchronization* criterion reaches 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 *prefix-scan* execute few kernel calls to compute the respective algorithms involving negligible host-device synchronization overhead. The different implementations of the reduction also show a high value in almost all criteria except *Computation Intensity* and *Data Transfer*. This is due to two main reasons. First, the reduction primitive implements a highly regular computation on the input data, which does not cause work unbalance and, second, involves a simple memory access pattern that allows to achieve memory coalescing. In contrast, the *prefix-scan* primitive has been implemented, in all the evaluated libraries, through a two-phase algorithm that presents a more complex memory access pattern. This involves lower values of *L1 granularity* and *L2 granularity*. The algorithm requires also a sophisticated control flow that affects the *device synchronization* and the *thread divergence* criteria. Finally, all implementations of both algorithms shows a low value of *computation intensity* criterion because such primitives are clearly memory-bounded. This characteristic limits the opportunity to take advantage of the huge processing power of the GPU.

The impact of each criteria on the overall quality metric 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 metric 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 3 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



(a) Reduction (b) Prefix-Scan

Fig. 2: Optimization criteria evaluation of the *reduction* and *prefix-scan* primitives

Parallel Primitives	GPU vs. CPU Simulation speedup					Quality metrics value ([0, 1])				
	ArrayFire	CUB	CUDPP	MGPU	Thrust	ArrayFire	CUB	CUDPP	MGPU	Thrust
Compaction		67	24		14		0.87	0.75		0.59
Merge				32	21				0.93	0.78
Partition		44			6		0.83			0.58
PrefixScan	63	223	114	135	68	0.82	0.81	0.78	0.89	0.87
Reduction	1009	961	865	1074	1069	0.66	0.89	0.74	0.76	0.69
Segmented PrefixScan			26		4		0.93		0.85	0.62
Segmented Reduction	err	28		28	8	err	0.86			0.46
SetUnion	3			13	3	0.68			0.90	0.73
Sort	80	85	39	48	80	0.68	0.69	0.51	0.58	0.69
Unique	err	73			17	err	0.70			0.56
Vect. Binary Search				527	167				0.48	0.32

TABLE 3: Quality metrics values obtained with $W_{OCC} = 30$; $W_{Gran_{L1}} = 100$; $W_{Gran_{L2}} = 100$; $W_{\text{HSync}} = 40$; $W_{\text{DSync}} = 15$; $W_{TDiv} = 40$; $W_{LB_W} = 50$; $W_{LB_{SM}} = 30$; $W_{\text{SMem}_{\text{eff}}} = 30$; $W_{CI} = 100$; $W_{DT} = 50$ and the corresponding actual GPU vs. CPU simulation speedup. Blank cells indicate that the corresponding libraries do not support the parallel primitive, while the `err` notation means that the primitive execution returns a run-time error.

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.

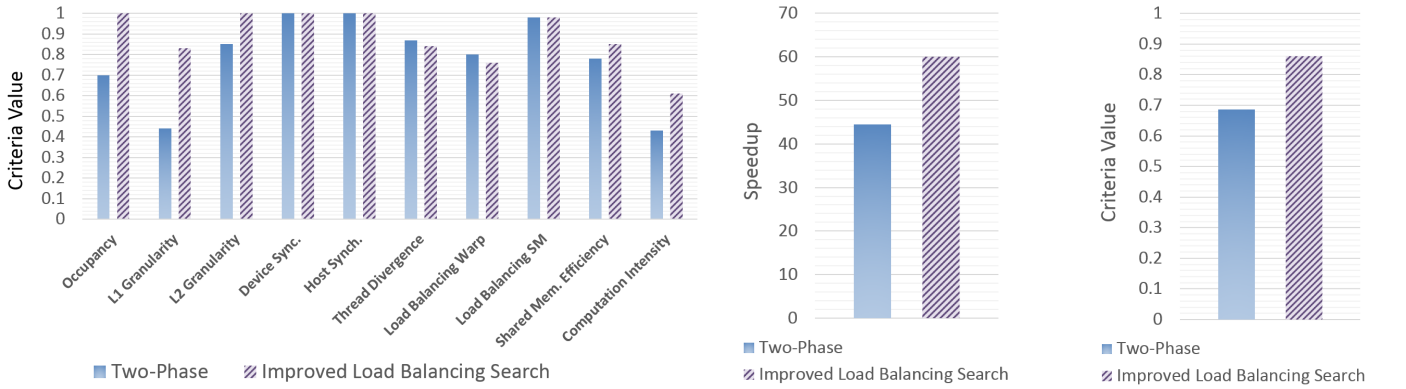
From the results reported in Figure 2, it is possible to compare different implementations of a given primitive in terms of performance and to understand which characteristics of such implementations lead to the corresponding speedup. As an example, the *Thrust* library provides the best implementation of the *reduction* primitive even though such an implementation presents a value lower than one for *load balancing warp* criterion. On the other hand, the code has been implemented by fully exploiting *L1/L2 Granularity* and by showing a good value of *computation intensity*, whose criteria values have more impact in the overall quality metrics. The *reduction* primitive implemented in *CUDPP* shows low values of *load balancing warp*, *thread divergence* and *computation intensity* that on average are worse than the other library implementations. This underlines that the threads organization and coordination presents many issues in the primitive. Another example is the very low value of *thread divergence* criterion obtained with the *prefix-scan* primitive of *ArrayFire*. The divergence issue indicates a high number of different execution paths among warp threads that, combined with a low value of *L1 Granularity*, represent the main performance bottlenecks.

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 analyze and improve the implementations of a *load balancing search* and a *matrix transpose*, as explained in the following section.

6 Case Studies

6.1 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 an 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.



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

Fig. 3: *Load balancing search* primitive evaluation

Among the libraries evaluated in this work, only *ModernGPU* provides an implementation of the *load balancing search* primitive. We applied *Pro++* to such a primitive to calculate the optimization criteria values, the CPU/GPU simulation speedup, and the overall quality metric value by considering the weights proposed in Section 5 (Table 3). Figure 3 reports the results (*MGPU* columns). Then, starting from the *ModernGPU* 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 criteria related to the memory coalescing. 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 both the *L1 Granularity* and *L2 Granularity* criteria values from 0.44 to 0.83 and from 0.85 to 1, 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 required the introduction of many extra control flow statements, which decreased the value of the *thread divergence* criterion with respect to the original *ModernGPU* implementation. Nevertheless, considering such a decrease and 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 an impact on the value of the *load balancing* criteria. 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 criteria. The modifications also slightly reduced the *thread divergence* and *Load Balancing Warp* values, which, on the other hand, still remains high. As a consequence, any further investigation or modification of the code, targeting thread divergence would not be worth to improve the overall quality of the primitive implementation. The *device* and *host synchronization* criteria had the highest values, 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 significantly improve step-by-step, in the optimization cycle, the performance of the *load balancing search* exploring the suggested guideline on the optimization criteria.

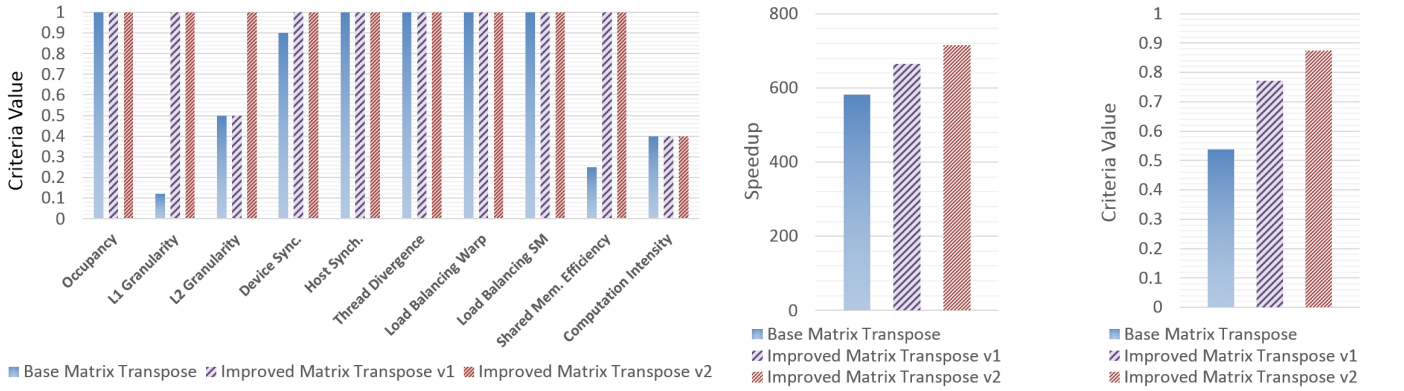
6.2 The *Matrix Transpose*

Transpose of a matrix is a basic linear algebra operation that has a deep impact in many computational science applications. The performance of matrix transpose is often compared with matrix copy due to the memory bottleneck. We analyzed the matrix transpose implementation presented in [40], which is characterized by data tiling in shared memory and thread organization in 2D hierarchical grids and blocks.

Figure 4 shows the results. The original code already provides values close to the maximum for the *host* and *device synchronizations*, *thread divergence*, *Warp/SM load balancing* and *occupancy* criteria. This is due to the fact that the application algorithm relies on very regular and independent tasks.

All the other criteria have very low values (between 0.1 and 0.5), thus we investigated to improve the code by considering memory related criteria both for global and shared memory spaces.

In the first optimization (Version1), we focused on improving the shared memory bank conflicts (*shared memory efficiency* criterion) by applying the *memory padding technique*.



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

Fig. 4: *Matrix Transpose* evaluation

The optimization has been designed mainly for the NVIDIA Fermi architecture that as a low number of independent memory banks. The memory padding has less impact on NVIDIA Kepler architecture and the gained speedup is marginal. We also improved the *device synchronization* criterion by removing barriers and by re-organizing the execution flow in order to assign an independent task to each warp .

In the second optimization (Version3) we have taken into account the memory access patterns to improve the *L1* and *L2 granularity* criteria. Their low values suggest that the memory accesses do not match the granularity of the respective caches, thus involving a waste of the memory bandwidth. We fully optimized both the criteria by simply re-organizing the thread block configuration and by resizing the memory *tiles*.

7 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 two standard GPU primitives, i.e., the load balancing search and the matrix transpose.

References

[1] "Hybrid System Architecture - HSA Foundation," http://www.hsafoundation.com.

[2] F. Bistaffa, A. Farinelli, and N. Bombieri, "Optimising memory management for belief propagation in junction trees using gpgpus," in *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, vol. 2015-April, 2014, pp. 526–533.

[3] N. Bombieri, F. Fummi, and S. Vinco, "On the automatic generation of gpu-oriented software applications from rtl ips," in *2013 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2013*, 2013.

[4] V. Bertacco, D. Chatterjee, N. Bombieri, F. Fummi, S. Vinco, A. Kaushik, and H. Patel, "On the use of gp-gpus for accelerating compute-intensive eda applications," in *Proceedings -Design, Automation and Test in Europe, DATE*, 2013, pp. 1357–1366.

[5] S. Vinco, D. Chatterjee, V. Bertacco, and F. Fummi, "Saga: Systemc acceleration on gpu architectures," in *Proceedings - Design Automation Conference*, 2012, pp. 115–120.

[6] "OpenACC - Directives for Accelerators," http://www.openacc-standard.org/.

[7] D. R., B. S., and B. F., "Hmpp: A hybrid multicore parallel programming environment," 2007.

[8] 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.

[9] 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.

[10] "Spiral - Software/Hardware Generation for DSP Algorithms," http://www.spiral.net/bench.html.

[11] 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.

[12] "NVIDIA CUDA ZONE - GPU-accelerated libraries," https://developer.nvidia.com/gpu-accelerated-libraries.

[13] "CLPP - OpenCL Parallel Primitives Library," http://gpgpu.org/2011/06/03/opencl-parallel-primitives-library.

[14] N. Bombieri, F. Busato, and F. Fummi, "An enhanced profiling framework for the analysis and development of parallel primitives for gpus," in *Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2015 IEEE 9th International Symposium on*, Sept 2015, pp. 1–8.

[15] U. Lopez-Novoa, A. Mendiburu, and J. Miguel-Alonso, "A survey of performance modeling and simulation techniques for accelerator-based computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 1, pp. 272–281, 2015.

[16] R. Dietrich, F. Schmitt, R. Widera, and M. Bussmann, "Phase-based profiling in gpgpu kernels," in *Proc. IEEE ICPPW*, 2012, pp. 414–423.

[17] B. Coutinho, D. Sampaio, F. Pereira, and W. Meira Jr., "Profiling divergences in gpu applications," *Concurrency Computation Practice and Experience*, vol. 25, no. 6, pp. 775–789, 2013.

[18] P. Guo and L. Wang, "Accurate cross-architecture performance modeling for sparse matrix-vector multiplication (spmv) on

gpus," *Concurrency Computation*, vol. 27, no. 13, pp. 3281–3294, 2015.
[19] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 152–163, Jun. 2009.
[20] K. Kothapalli, R. Mukherjee, M. Suhail Rehman, S. Patidar, P. Narayanan, and K. Srinathan, "A performance prediction model for the cuda gpgpu platform," in *Proc. of IEEE HiPC*, 2009, pp. 463–472.
[21] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A performance analysis framework for identifying potential benefits in gpgpu applications," in *Proc. of ACM SIGPLAN PPoPP*, 2012, pp. 11–22.
[22] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 353–364.
[23] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 163–174.
[24] S. Collange, M. Daumas, D. Defour, and D. Parello, "Barra: A parallel functional simulator for gpgpu," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 351–360.
[25] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: a simulation framework for cpu-gpu computing," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 335–344.
[26] M. Zheng, V. Ravi, W. Ma, F. Qin, and G. Agrawal, "Gmprof: A low-overhead, fine-grained profiling approach for gpu programs," in *Proc. of IEEE HiPC*, 2012.
[27] A. Kerr, G. Diamos, and S. Yalamanchili, "Modeling gpu-cpu workloads and systems," in *Proc. of GPGPU*, 2010, pp. 31–42.
[28] K. Sato, K. Komatsu, H. Takizawa, and H. Kobayashi, "A history-based performance prediction model with profile data classification for automatic task allocation in heterogeneous computing systems," in *Proc. of IEEE ISPA*, 2011, pp. 135–142.
[29] C. NVidia, "C best practices guide," *NVIDIA, Santa Clara, CA*, 2012.
[30] D. B. Kirk and W. H. Wen-mei, *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.
[31] J. Cheng, M. Grossman, and T. McKercher, *Professional Cuda C Programming*. John Wiley & Sons, 2014.
[32] F. Busato and N. Bombieri, "BFS-4K: an efficient implementation of BFS for kepler GPU architectures," *IEEE Transactions on Parallel Distributed Systems*, vol. 26, no. 7, pp. 1826 – 1838, 2015.
[33] S. Xiao and W. chun Feng, "Inter-block gpu communication via fast barrier synchronization," Dept. of Computer Science Virginia Tech, Tech. Rep., 2009.
[34] J. Hoberock and N. Bell, "Thrust: A parallel template library," 2014. [Online]. Available: http://thrust.github.io/
[35] D. Merrill, "Cub," 2015. [Online]. Available: http://nvlabs.github.io/cub/
[36] M. Billeter, O. Olsson, and U. Assarsson, "Efficient stream compaction on wide simd many-core architectures," in *Proceedings of the HPG 2009: Conference on High-Performance Graphics 2009*, 2009, pp. 159–166.
[37] M. Harris, J. Owens, S. Sengupta, Y. Zhang, and A. Davidson, "Cudpp: Cuda data parallel primitives library," 2014. [Online]. Available: http://cudpp.github.io/
[38] S. Baxter, "Modern gpu," 2014. [Online]. Available: http://nvlabs.github.io/moderngpu/
[39] J. Malcolm, P. Yalamanchili, C. McClanahan, V. Venugopalakrishnan, K. Patel, and J. Melonakos, "Arrayfire: a gpu acceleration platform," 2014. [Online]. Available: http://arrayfire.com/
[40] G. Ruetsch and P. Micikevicius, "Optimizing matrix transpose in cuda," *Nvidia CUDA SDK Application Note*, vol. 18, 2009.

Nicola Bombieri received the PhD in Computer Science from the University of Verona in 2008. He is Professor Assistant at the Dept. of Computer Science of the University of Verona. His research activity focuses on high performance computing, design and verification of embedded systems, and automatic generation and optimization of embedded SW. He has been involved in several national and international research projects and has published more than 70 papers on conference proceedings and journals.

Federico Busato received the Master degree in Computer Science from the University of Verona in 2014. Currently he is a Ph.D. student at the University of Verona, Department of Computer Science. His research activity focuses on high performance computing and graph theory.

Franco Fummi is Full Professor and the Head of the Department of Computer Science of the University of Verona. His main research includes electronic design automation methodologies for modeling, verification, testing and optimization of embedded systems. He received the PhD in Electronic Engineering from Politecnico di Milano in 1995. He is an IEEE member and a member of the IEEE test technology technical committee. He is also co-founder and active project leader of EDALab srl.