K-model: A New Computational Model for Stream Processors

Gabriele Capannini Fabrizio Silvestri Ranieri Baraglia
Institute of Information Science and Technologies
National Research Council
Pisa, Italy
g.capannini@isti.cnr.it *f.silvestri@isti.cnr.it* *r.baraglia@isti.cnr.it*

***Abstract*—We introduce K-model, a computational model to evaluate the algorithms designed for graphic processors, and other architectures adhering to the stream programming model. We address the lack of a formal complexity model that properly accounts for memory contention, address coalescing in memory accesses, or the serial control of instruction flows.**

We study the impact of K-model rules on algorithm design. We devise a coalesced and low contention data access technique for Batcher's networks, and we evaluate the effectiveness of this technique within our K-model. To evaluate the benefits in using K-model in evaluating solutions for streaming architectures, we compare the complexity of a sorting network built using our technique, and quicksort. Although in theory quicksort is more efficient than bitonic sort, empirically, our bitonic sorting network has been shown to be faster than the state-of-the-art implementation of quicksort on graphics processing units (GPUs). We use our K-model to prove that this observation should generally hold.

As a side result, our technique to perform a Batcher's network on GPUs improves the performance of one the fastest comparison-based solution for integers sorting.

***Keywords*-Parallel algorithms; Parallel processing; Modeling;**

I. Introduction

Nowadays computing architectures exploiting "many-cores" processors are available. Graphics Processing Units (GPUs), Sony's Cell/BE processor, and next-generation CPUs compute hundreds of arithmetic and logical operations at the same time. The key feature of all these architectures is the use of a massive collection of SIMD computing cores integrated in a single silicon chip. The computational elements of such cores share a small on-chip data-memory. Furthermore, all the SIMD processors are connected to a global off-chip memory. Such novel many-core processors are programmable accelerators classified as high-parallel shared-memory architectures capable of performing high performance computations [1].

An important aspect is that as processor building technology advances, GPUs and CPUs differently exploit the number of additional transistors available. GPUs are specialized for compute-intensive computations, and therefore designed to devote more transistors to data processing rather than data caching and flow control. As a result, the parallelism adopted by GPUs is devoted to augment roughly the number of arithmetical operations issued at the same time, and the bandwidth of the communication channels. On the other hand, CPUs offer more flexible programming models and techniques making CPU programming a task easier than GPU programming.

In particular, graphics processors refer to the so called *Stream Programming Model* [2]. Briefly, according to such model, a stream program organizes data as streams and expresses all computations as *kernels*. A stream is a sequence of similar elements. Each stream element is composed by a subset of the input dataset which can be, in turn, computed independently. A kernel consumes a set of input streams, performs a computation, and produces a set of output streams. Streams passing among multiple computation kernels form a stream program.

As it is shown in Section III, in the literature many computing models have been presented to describe and to analyze sequential and parallel algorithms. For many reasons these models are useless for our purposes. For instance, their target architectures are too different from "manycores", or the level at which they abstract the architecture is too high, they tend not to consider important aspects of computations. Just to make an example, the widely known Parallel Random Access Machine (PRAM) model [3] can be considered a unifying parallel paradigm. However, it is far from being accurate: for example it does not consider the cost of non-local memory references that, instead, has a great impact on performance. The PRAM model has been applied to some recently proposed GPU algorithms [4]. Indeed, as we describe in this paper, experimental results contradict theoretical results that are not done using a computational model specifically designed for GPUs.

Based on their historical development, three generations of parallel computational models have been proposed so far. Usually, they differentiate on the basis of the memory model exploited by their targeted parallel architectures. The first generation, developed since 1978, consists in the shared memory parallel computational models: examples are the PRAM model and the Queue-Read Queue-Write PRAM (QRQW PRAM) [5]; the second generation, developed since 1984, consists of the distributed memory parallel computational models, it includes the Bulk Synchronous Parallel (BSP) model [6]; the third generation consists of the hierarchical memory parallel computational models, it includes

models such as the PRAM, Parallel Hierarchical Memory Model (PHMM) and Parallel Block Transfer (PBT) [7], Uniform Parallel Memory Hierarchy (UPMH) [8], and so on. All these models have some features in common with the architectures we are considering, yet they fall short in targeting our reference architectures.

There are two main contributions in this paper:

1) We propose *K-model*, a computational model developed to capture all the distinguishing features of GPUs, and alike architectures. An algorithm, in the K-model, is evaluated by measuring the *time complexity* and the *computational complexity*. The former is the *latency* of each instruction an algorithm calls, summed over all the instructions called. The computational complexity, instead, can be thought of as the classical sequential complexity assuming we are simulating the execution of the algorithm on a serial RAM. To the best of our knowledge this is the first paper attempting to propose a computational model specifically targeting GPUs (and manycore architectures in general).
2) On the basis of the K-model rules, we devise a partitioning function to exploit efficiently a butterfly networks through a stream program on graphics processors, and then we applied such function to the Batcher's sorting network [9]. Our sorting algorithm has been proven to have a lower latency with respect to another solution based on the quicksort algorithm. Our approach is particularly efficient because we designed it to optimize all the metrics introduced in the K-model complexity functions. In particular, our solution reduces the memory contention in accessing the shared memory from the set of computational elements computing on a stream element. Furthermore, it exploits all the computational elements available, and finally it issues *coalesced* sequences of non-local memory references so as to maximize the communication bandwidth.

We present the architectural features details in Section II. Section III and Section IV discuss related work and introduce the K-model, respectively. We address the butterfly network mapping function in Section V, showing that for the most part of the cases it entails no loss of performance with respect to the architectural capability. In Section VI we evaluate two sorting algorithms, one based on quicksort and the other on the bitonic sorting network, which exploits our techniques. In the same section, test results show the results obtained implementing the solution on a real manycore device.

II. Architectural Details

GPU, Sony's Cell/BE processor, and next generation CPUs permit computing hundreds of arithmetical operations at the same time. The key features joining these architectures is integrating several computing cores on a single chip, grouped into SIMD processors. We focused in particular on GPUs.

Such architectures are built around a scalable array of streaming multiprocessors (MPs). Each MP comprises one instruction unit, a collection of single precision pipelines executing scalar instructions, and a local data-memory called *shared memory*. This memory is divided into banks, that can be accessed simultaneously. If two addresses of a memory request fall in the same memory bank, there is a bank conflict and the accesses have to be serialized, penalizing the latency of the overall requests.

All MPs create, manage, and execute concurrent "threads" referred to as kernels in hardware with zero scheduling overhead, and implement a barrier for threads synchronization. Nevertheless, only threads concurrently running on the same MP can be synchronized.

Regarding the shared memory, each device is equipped with its own global memory that is off-chip, and can be addressed by each MP during the computation. Due to the number of memory transactions issued to access to the different memory segments addressed, the accesses to the global memory should be made in a *coalesced* manner to minimize the number of memory transactions, i.e. the latency of data transfers.

The organization of the other type of processors is similar to GPUs. Cell architecture exposes on the same chip both the CPU and the set of SIMD processors, called PPUs, and equipped with a local memory. In this case all processors share the system main memory that is unified with the analog GPU global memory.

A peculiarity modelling SIMD processors is the *serial control*. As in the standard RAM model, having one instruction unit, conditional branches are scalar primitives. Serial control has important advantage simplifying the algorithm design: the programmer only needs to worry about a single program. On the other hand, serial control is more restrictive than parallel control. This aspect can also significantly impact the effective instruction throughput by causing threads on the same processor to "diverge", that is to follow different execution paths. If this happens, the different executions paths have to be serialized, increasing the total number of instructions executed, as well as the latency of the kernel execution.

III. Related Works

For our analysis, we discuss models able to abstract computer architecture having a complex memory hierarchy, or some other analogies with stream processors. Moreover, we refer to such architectures in order to evaluate the models considered. In the following we analyze the Hierarchical Memory with Block Transfer (HMM-BT) and Cache-oblivious models, from the sequential class of computing models, and PRAM, BSP, Vector RAM (VRAM), PHMM,

PBT, QRQW PRAM, and UPMH, from the parallel class of computing models.

HMM-BT has been proposed by Aggarwal *et al.* [10]. In the HMM [11] model, the authors assume k levels of memory hierarchy, each containing 2^k locations, and to access to memory location x takes $f(x)$ time. The BT model differentiates from HMM by incorporating block transfers into its cost function. By using the block transfer, the memory access to a data segment with length l at address x costs $f(x) + l$. Parallel versions of these models, namely PHMM and PBT, are easy to build by replicating the sequential model p times, then by connecting them at the lowest memory level.

Cache-Oblivious (CO) [12] algorithms are designed for multilevel memory hierarchy without knowing any parameters of the hierarchy, and only knowing the existence of a hierarchy. To measure the complexity of an algorithm, one needs to measure its running time in the RAM model, and the number of cache misses it incurs as a function $Z()$ of both the size and the line length of the cache.

A consideration concerning serial models is that they expose a hierarchical memory similar to a target architecture that, in turn, does not suffer from issues like shared memory contention and serial control, typical of our architectures. In fact, these aspects are not negligible, in particular for computation-intensive algorithm in order to avoid unworthy serializations in computing a stream element. On the other hand, accesses to lower memory levels of the hierarchy are considered, but not as carefully as they should. Besides, coalescing aspects do not emerge. In brief, BT function $f(x) + l$ well formalizes the access latency to disk storage devices, yet it fails to model global memory accesses.

Cache-Oblivious $Z()$ function explicitly considers cache equipped machines. Unfortunately, the latency of a cache line replacement is considered to be constant disregarding the coalescing of locations in each cache line. In our case, latency is heavily affected by the coalescing of the addresses of moved data.

A PRAM [3] is a shared memory abstract machine made up of p processors. PRAM, indeed, neglects issues such as synchronization and communication. This leads to unrealistic expectations for algorithms designed in this model, and thus we are prone to consider as theoretically optimal, algorithms that are not optimal when ported onto a real machine [13], [4].

The BSP [6] model explicitly requires processors to synchronize every a fixed number of computing steps. A number of p processor-memory components, a router that delivers point-to-point messages and the facilities for synchronization are required to build a BSP parallel computer. The program executed on the BSP model consists of a sequence of *supersteps*, which perform the combination of local computations on local data and message passing.

The VRAM [14] computational model consists of a serial RAM with a vector memory and a vector processor. A program for a VRAM is evaluated by counting the number of either scalar or vector primitives invoked. To compute complexities within VRAM, parallel vector processors are considered to replace non-vector processors in a serial RAM. This effective approach points out the effect of the serial control, but lacks of the management of the contention on the on-chip shared memory. On the other hand, the QRQW PRAM model does not take into consideration such aspect, but it sufficiently formalizes the cost of on-chip shared memory access: it permits concurrent reading and writing, but at a cost proportional to the number of readers/writers to a memory "bank" in a given step. A model obtained by merging VRAM and QRQW PRAM model would correctly capture all the features characterizing a stream multiprocessor.

The Uniform Memory Hierarchy (UMH) model allows for simultaneous data transfers, while the HMM and BT models only allow one data transfer at a time. The concept of "communication efficiency", i.e. the ratio of RAM complexity and the UMH complexity, is used to assess the algorithm complexity under the UMH model. UPMH connects p sequential model through a tree structure. It refines the traditional algorithm analysis method by including the cost of data movement throughout the memory hierarchy. However, the memory cost functions depend on the memory level, and do not depend on the addresses at which a data is located. In this model both the shared memory contention and the serial control aspects cannot be evaluated.

The following table resumes the main features (i.e. serial control, contention for on-chip shared memory accessing, and coalescing for off-chip memory accessing respectively).

	Serial	Contention	Coalescing
PRAM	–	–	–
VRAM	✓	–	–
(P)HMM	–	–	–
(P)BT	–	–	✓
QRQW PRAM	–	✓	–
BSP	✓	–	–
UPMH	–	–	✓
CO	–	–	✓

IV. K-MODEL

K-model is designed to have an abstract machine able to capture the key features of the novel generation of the graphics processing units. This class of processors is specialized for compute-intensive, highly parallel computation, exactly what graphics rendering is about, therefore more transistors are devoted to data processing rather than data caching and flow control. These features well fit with the stream programming paradigm [2]. The salient aspect of this paradigm is splitting the overall computation into a set of independent computations. This set forms a stream of

elements, which are independently computed by one of the available SIMD processors [15].

Due to the homogeneity of available processors and the unpredictability with which a stream element is assigned to a processor, the abstract machine referred in the K-model appears simplified with respect to the real architecture. The K-model restricts the set of identical processors to only one. In fact, the benefit of having more SIMD processors on the real architectures does not concern the design of the algorithm which should aim at the effective computation of the single stream element. In other words, the core of the K-model is the evaluation of the "parallel-work" performed on each stream element by one SIMD processor, disregarding the number of real multiprocessors.

Before discussing the notion of complexity, we introduce the K-model to study the complexity of algorithms. This model, which is illustrated in Figure 1, consists of a computer with an array of k scalar execution units ($E[1..k]$) linked to a unique instruction unit (*IU*) which provides to dispatch the instructions to the scalar units. This forces the work of the scalar units to be synchronized, because only one instruction is issued at each step. The evaluation of an instruction issued by IU in the different execution paths of the program is based on two criteria: *latency* and *length*. Length is simply the number of scalar units that an instruction issued by the IU must run, then it will be in the range $[1..k]$. Latency, instead, is measured in terms of the type of instruction issued.

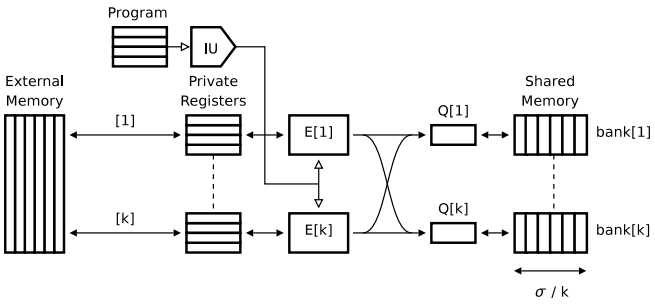

Figure 1. Representation of the architecture.

The memory hierarchy consists of an *external memory* and a local memory made of a set of *private registers*, and a *shared memory* of σ locations equally divided into k parallel modules. The K-model evaluates the latency of a data-access instruction proportionally to the level of contention it generates. Whenever an instruction addresses a shared memory location with no bank conflict or a register, the latency of the instruction has a cost 1. Otherwise, the latency of a data-access instruction is equal to the highest number of requests incoming to one of the k memory banks. In other words, assuming to have a queue ($Q[1..k]$) for each bank which collects the requests incoming at each step to the related memory addresses, the latency for accessing the shared memory is equal to such queue length. Since all execution units are synchronous, the next instruction will be performed only when all requests are satisfied, i.e. when the longest queue is empty.

Regarding the external memory, its latency is evaluated separately. The peculiarity of the external memory is the coalescing of the accesses. Whenever the scalar units access a contiguous set of words (i.e. for each $i \in [1..k]$, $E[i]$ accesses the i-th word of a segment in the external memory) the related requests are coalesced into one memory transaction, thus the transfer-time of k accesses overlap. When accesses can not be coalesced, the architecture considers one memory transaction for each length-k segment addressed.

The latency of the arithmetical instructions has a cost equal to 1.

Performance Indicators: Given a algorithm, first we measure two performance indicators, namely $T()$ and $W()$, to evaluate the work performed inside the processor. In particular, both $T()$ and $W()$ are function of latency and length of the instructions previously defined. $T()$ represents the latency per instruction call, summed over the instructions issued, and $W()$ is the length induced by each instruction, summed over the instructions issued.

Note that $T()$ can be thought as the parallel complexity of the algorithm assuming a collection of k scalar processors, whereas $W()$ can be thought as the serial complexity, assuming we are simulating the execution on a serial RAM. Generally, we are interested in the minimization of $T()$, so as to obtain faster execution; $W()$, instead, is more relevant during the efficiency analysis of the solution we are evaluating. For example, the value of $\frac{W()}{k \cdot T()}$ gives us the level of exploitation of the architecture performing the computation of a stream element: values closer to 1 are indicative of a good efficiency.

Eventually, in order to evaluate an algorithm in the K-model, we have to evaluate the stream of elements to compute by multiplying the complexities of a generic stream element, i.e. $T()$ and $W()$, by the total number of elements in the stream. Whenever the algorithm is made of more streams, its complexity is defined as the sum over the complexities of the different streams.

However, the evaluation of the "internal-work" induced by a stream element is not the unique purpose of K-model. A relevant aspect to take into consideration is the pattern used to access the external memory in order to fetch the input data and then to flush the results. Since the cost of accessing the external memory is one order of magnitude higher than the other type of data-accesses, and due to the parallelism among the instructions performed on the local memory and the access to the external one, we measure the number of *memory transactions* issued to access to the external memory. This measure is represented by the function $G()$ and, as said, it aims at taking into account coalesced memory

accesses by k scalar processors, i.e. accesses to locations lying in the same size k segment within a single memory transaction.

V. Butterfly Networks Mapping

In this section we describe a K-model efficient technique to decompose the computation of a butterfly. In particular, we devise our version of Bitonc Sorting network (BS) [16] by considering the BS as a sequence of butterfly networks. In the following, Algorithm 1 describes the sequential version of the BS algorithm.

Algorithm 1 BitonicSort (A)

1: $n = |A|$
2: **For** $s = 1$ **To** $\log_2 n$ **Do**
3: **For** $c = s - 1$ **Down To** 0 **Do**
4: **For** $r = 0$ **To** $n - 1$ **Do**
5: **If** $\frac{r}{2^c} \equiv \frac{r}{2^s} \pmod 2 \wedge A[r] > A[r \oplus 2^c]$ **Then** Swap $(A[r],\ A[r \oplus 2^c])$

We refer to the comparisons performed in the loop at line 4 of Algorithm 1 using the term $step_c$, where the parameter c refers the related variable in BS. At each step, all elements of A are compared exactly once. In particular, given a $step_c$, the element at position i is compared with the one at position j, if and only if the binary representation of i and j differ only by the c-th bit. Let us do the same by considering two steps, for example $step_c$ and $step_{c-1}$, and let us merge the pairs defined at $step_c$ with the ones belonging to $step_{c-1}$ that share at least one element. We obtain that a new subset contains the elements whose binary representation differ only by the bits at position c and $c-1$. Successively, we can iterate such process with the pairs of $step_{c-2}$ and so on. Let us note that these sets define a partition of A. This follows by straightforward Algorithm 1. In particular, given a single $step_c$, the distinct pairs compose a partition of A. Successively, the only operation we perform is merging of two existing subsets, then the resulting subsets are still a partition of A.

From the discussion above, we can assert the following claim.

Claim 1. *Let Γ be a valid set of values of "c" for the BS relative to an array A, with $|A| = 2^x$ for some integer x. The partition P_Γ generated by merging the pairs related to the steps whose indexes are in Γ, has $2^{x-|\Gamma|}$ parts and $2^{|\Gamma|}$ elements per part.*

Since the shared memory is made of σ memory locations, a part of P_Γ can contain at most σ elements. From Claim 1 it follows that $|\Gamma| = \log_2(\sigma)$, this means that Γ can contain $\log_2(\sigma)$ distinct values of "c".

Let us define the sequence of P_Γ partitions that defines the mapping of the network computation. At the begin of the computation, since $s = 1$, $step_0$ is performed, successively $s = 2$ and the algorithm performs $step_1$, then $step_0$ and so on, until the last $step_0$ when $s = \log_2 |A|$. Let us count $\log_2(\sigma)$ different values of c occurring in the sequence of $step_c$ to perform. Each group of different c identifies a set Γ, thus the P_Γ partition.

Considering the K-model, in order to reduce the number of memory transactions, we initially create parts of k consecutive elements, so as to coalesce their transfer into one memory transaction. Algorithmically, this means that each Γ initially is not empty, but it contains the values $[0, \log_2 k - 1]$. Nothing changes in the rest of the algorithm, with the exception of the number of $step_c$ performable by using P_Γ. Since Γ already contains $\log_2 k$ values, it has just $\log_2(\sigma) - \log_2 k$ available positions. Counting the succeeding values of c in the sequence of $step_c$ to perform, it can push at most $\log_2(\sigma/k)$ that is the minimum number of steps coverable. In particular, if the next values of c coincide with some of the initial $\Gamma = [0, \log_2 k - 1]$, then the number of steps coverable augments up to $\log_2(\sigma)$ steps. Then, each P_Γ partition is executed by computing all parts as a stream. Thanks to the mapping adopted, each part is modeled as bitonic network.

In the following, the related pseudo-code is presented, see Algorithm 2. It permits us to maintain always active the k executors, and at the same time, to balance the contention on the k banks. Note that balancing the contention, the latency of the access is reduced because the maximum contention is lower.

Algorithm 2 RunStreamElement (A_{part}, Γ)

1: **Foreach** $id \in [0, k-1]$ **Do** `//in parallel`
2: $n = \log_2(\sigma)$
3: **For** $i = 0$ **To** $n - 1$ **Do**
4: $c = \Gamma[i]$
5: **For** $j = id$ **To** $n/2 - 1$ **Step** k **Do**
6: $p =$ InsAt $(j, 0, c)$
7: $q =$ InsAt $(j, 1, c)$
8: Compare $(A_{part}[p], A_{part}[q])$

The pseudo-code discards some side aspects, to focus the main technique. It considers the case with no optimization for reducing the number of memory transaction, so Γ has only values referring a $step_c$. The procedure InsAt (N, x, p) inserts the bit x at the position p of the binary representation of N, for example InsAt $(7, \mathbf{0}, 1) = 11\mathbf{0}1 = 13$. The procedure Compare performs the comparisons between the argument elements, and, in case, swaps them.

VI. Evaluation

Let us compare the complexity of the proposed algorithm with the complexity of another comparison-based solution for integers sorting, namely quicksort [4]. Both the solutions

have been proposed for GPUs. However, it exists some further advantages that can lead to prefer BS. The first one is its space complexity, because BS does not require auxiliary space during the computation with respect to some other solutions. The second advantage regards the portability of the method. Driven by the K-model rules, that are superset of the rules presented in other models, and due to the applicability of the butterfly network, the mapping function proposed can be efficiently adopted on cache-based processors[1].

Bitonic Sorting Networks: Let us count the number of *memory transaction* needed to compute a sorting network for an array of size n. The number of steps to perform are $(\log^2 n + \log n)/2$. We have to know the number of partitions needed to cover all the network, in order to know how many stream elements are computed, then the number of fetch/flush phases, and the number of memory transactions. The first transaction covers the first $(\log^2 \sigma + \log \sigma)/2$ steps, it follows from Claim 1, and because in such steps $c < s \leq \log \sigma$, thus c loops on the same values.

Let us call $stage_s$ the loop at line 2 of Algorithm 1. In the remaining steps $s > \sigma$, $\log n - \log \sigma$ stages remain, and each of them has the last P_Γ partition that covers $\log \sigma$ steps, namely when $\Gamma = \{0, .., \log \sigma - 1\}$. On the other hand the $s - \log \sigma$ steps are performed with partitions covering $\log(\sigma/k)$ steps. Resuming, the number of partitions needed to cover all the network is

$$1 + \sum_{s=\log \sigma + 1}^{\log n} \left(\left\lceil \frac{s - \log \sigma}{\log(\sigma/k)} \right\rceil + 1 \right) = O\left(\frac{\log^2 n}{\log k} \right)$$

Since, each element fetches and flushes only coalesced subset of elements, the number of transactions is

$$O\left(\frac{n}{k} \cdot \frac{\log^2 n}{\log k} \right)$$

Regarding $W()$, it is known and it corresponds to the number of comparisons computed for the same algorithm performed in the RAM model, that is

$$O(n \log^2 n)$$

Whereas, thanks to Algorithm 2 which equally spreads the contentions among the k memory banks and maintains active all elements, the $T()$ is

$$O\left(\frac{n \log^2 n}{k} \right)$$

Quicksort: It splits the computation in $\log n$ steps. For each step it performs three kernels. In the first one, it equally splits the input and counts the number of elements greater than the pivot, and the number of the elements smaller than the pivot. In the second, it performs twice a parallel prefix sum of the two set of counters in order to know the position where to write the elements previously scanned. In the final kernel, it accesses to the data in the same manner that in the first kernel, but writes the elements to the two opposite heads of an auxiliary array beginning at the positions calculated in the previous kernel.

The first kernel coalesces the access to the elements, and because the blocks are equally sized, also the computation is balanced. Then the counters are flushed, and the second kernel starts. Supposing that $n/k < \sigma$, each prefix sum can be computed within a unique stream element. Consequently, for each prefix sum we need n/k^2 memory transactions to read n/k counters. This means that $T()$ is logarithmic in the number of addends, on the contrary $W()$ is linear. Last kernel is similar to the first one, except for flushing the data into the auxiliary array. In particular, because each thread accesses to consecutive memory locations, the main part of the requests is not coalesced, requesting one memory transaction per element.

The following table contains the evaluation of the three type of kernel in the K-model. In order to compute the complexity of the whole algorithm, the sum of such formulas have to be multiplied by $\log n$.

	$G(n,k)$	$W(n,k)$	$T(n,k)$
kernel #1	$n/k + 2$	n	n/k
kernel #2	$4n/k^2$	$2n/k$	$4 \cdot \log \frac{n}{k}$
kernel #3	$n/k + n$	n	n/k
Overall	$O(\frac{n}{k} \log n)$	$O(\frac{n}{k} \log n)$	$O(n \log n)$

By comparing the evaluation of the function $T(n,k)$ of the QS proposed in [4] with the related PRAM complexity computed by the authors[2] nothing is changed. However, considering also the number of memory transaction, $G(n,k)$, it emerges a considerable waste of bandwidth in the access to the off-chip memory. Since it involves extra-chip communications, this kind of operations is the slowest taken into account by the K-model.

Likely, the waste of bandwidth is the main key feature to consider for this type of algorithm, which can not be properly classified as computational-intensive. The rationale suggests that analysing an algorithm exposing an higher number of operations per memory location, $T()$ probably increases its relevance.

To conclude this section, we present the actual, i.e. measured, performance of BS and quicksort (QS) in terms of number of memory transactions, memory contention, number of divergent paths, and elapsed computational time. For tests we implemented the version of BS proposed in Section V and we performed the implementation of QS used by Cederman *et al.* [4]. We performed experiments on a Linux-based platform equipped with an NVidia GeForce

[1]Exploiting also the SIMD instruction set extension (SSE) if present on the real architecture.

[2]It results $O(n \cdot \log n)$, that is the same to cosider only $T()$ in the K-model.

8800 GT video card having 14 stream multiprocessors. We used CUDA SDK 2.1 as the toolkit to develop our solution.

Problem Size		Memory Transactions	Memory Contention
2^{20}	BS	796800	34350
	QS	4446802	123437
2^{22}	BS	4119680	151592
	QS	18438423	379967
2^{24}	BS	20223360	666044
	QS	85843422	1379155
2^{26}	BS	101866786	2912926
	QS	N.A.	N.A.

		Divergent Paths	Elapsed Time (ms)
2^{20}	BS	0	22.837
	QS	272904	31.816
2^{22}	BS	0	111.399
	QS	1126038	172.543
2^{24}	BS	0	532.866
	QS	1126038	1027.270
2^{26}	BS	0	2654.649
	QS	N.A.	N.A.

In Figure 2 a plot of the elapsed time of sort algorithms measured varying the problem size is shown. We compare three different algorithms: cpu sort is the stdlib's qsort() function, quick sort is the GPU-based quick sort, and bitonic sort is our implementation of the Batcher bitonic sorting network. We are reporting the elapsed time also for a CPU-based solution to remark the benefits in using GPUs for this kind of problem. Obviously, better sorting algorithms for CPUs can be compared but it is out-of-scope of our work.

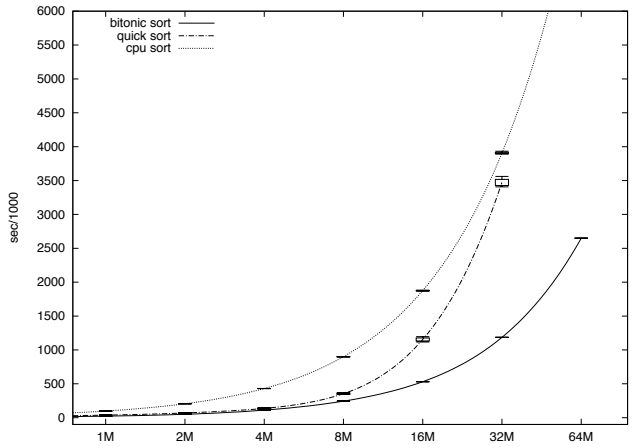

Figure 2. A plot showing the elapsed time of sort algorithms measured varying the problem size. We represent variance of elapsed times by using candlesticks.

Note that the adherence of the model to the experimental results obtained is still valid considering the theoretical results. In particular the theoretical evaluation of the number of memory transactions made for BS results higher than the one computed for QS. The motivation because the experimental results shows the opposite are represented in Figure 3. In practice, considering the specifications of real architectures, which related to the parameter k of the K-model, and considering the capacity of the external memory available on real devices (order of Gigabytes), QS results to be slower than BS.

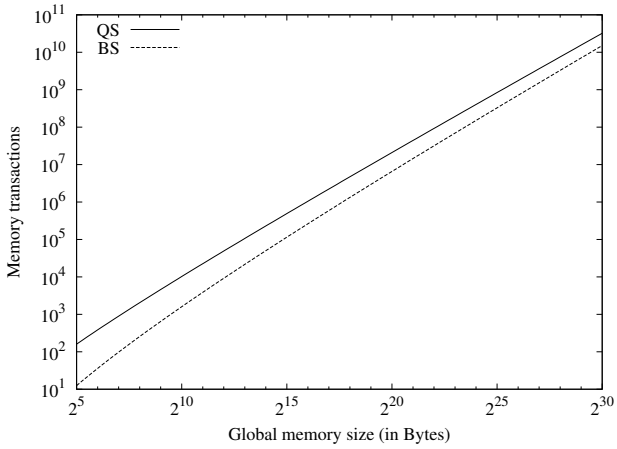

Figure 3. Theoretical number of memory transactions for BS and QS considering the specifications of real architectures, i.e. $k \simeq 16$, and the capacity of the external memory available as order of Gigabytes.

Eventually let us note that due to the space complexity of QS and due to the size of available memory on the device, QS is not able to perform the sorting of arrays containing 64M of integers.

VII. Conclusions

We have defined a novel computational model, namely the K-model. We wanted to consider all the important features of manycore architectures (in particular GPUs) in order to allow the design of efficient algorithms for this kind of hardware. We considered three main metrics: $T()$ and $W()$ complexity functions, and the number of memory transactions, $G()$.

We have shown the validity of our model by computing the complexity of a novel mapping of butterfly networks (i.e. Bitonic Sorting Networks) onto GPUs. The mapping scheme has been proven to be efficient in the K-model. In particular, we evaluated both theoretically and empirically our BS network mapping on GPUs. Empirical results showed that, due to a very effective exploitation of the K-model features, we are able to attain an efficient solution, to date, to comparison-based sorting.

Some further considerations regarding the portability of the K-model are valuable. By merging different models

existing in literature, in fact, K-model permits the reuse of some parts of the algorithms devised for such models.

REFERENCES

[1] J. Bovay, B. Henderson Brent, H. Lin, and K. Wadleigh, "Accelerators for high performance computing investigation," White paper, High Performance Computing Division - Hewlett-Packard Company, 2007.

[2] Brucek Khailany, William J. Dally, Ujval J. Kapasi, and Peter Mattson, "Imagine: Media processing with streams," *IEEE Micro*, vol. 21, no. 2, 2001.

[3] Steven Fortune and James Wyllie, "Parallelism in random access machines," in *STOC '78*, New York, NY, USA, 1978, ACM.

[4] Daniel Cederman and Philippas Tsigas, "A practical quicksort algorithm for graphics processors," in *ESA '08*, Berlin, Heidelberg, 2008, Springer-Verlag.

[5] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran, "The qrqw pram: accounting for contention in parallel algorithms," in *SODA '94*, Philadelphia, PA, USA, 1994, Society for Industrial and Applied Mathematics.

[6] Leslie G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, 1990.

[7] Jeffrey Scott Vitter and Elizabeth A. M. Shriver, "Algorithms for parallel memory ii: Hierarchical multilevel memories," *Algorithmica*, vol. 12, no. 2/3, pp. 148–169, 1994.

[8] Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker, "The uniform memory hierarchy model of computation," *Algorithmica*, vol. 12, 1994.

[9] K. E. Batcher, "Sorting networks and their applications," in *AFIPS '68*, New York, NY, USA, 1968, ACM.

[10] Alok Aggarwal, Ashok K. Chandra, and Marc Snir, "Hierarchical memory with block transfer," in *SFCS '87*, Washington, DC, USA, 1987, IEEE Computer Society.

[11] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir, "A model for hierarchical memory," in *STOC '87*, New York, NY, USA, 1987, ACM.

[12] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran, "Cache-oblivious algorithms," in *FOCS*, 1999.

[13] Lawrence Snyder, "Type architectures, shared memory, and the corollary of modest potential," pp. 289–317, 1986.

[14] Guy E. Blelloch, *Vector models for data-parallel computing*, MIT Press, Cambridge, MA, USA, 1990.

[15] Vipin Kumar, *Introduction to Parallel Computing*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[16] Majed Z. Al-Hajery and Kenneth E. Batcher, "Multicast bitonic network," in *SPDP*, 1993, pp. 320–326.