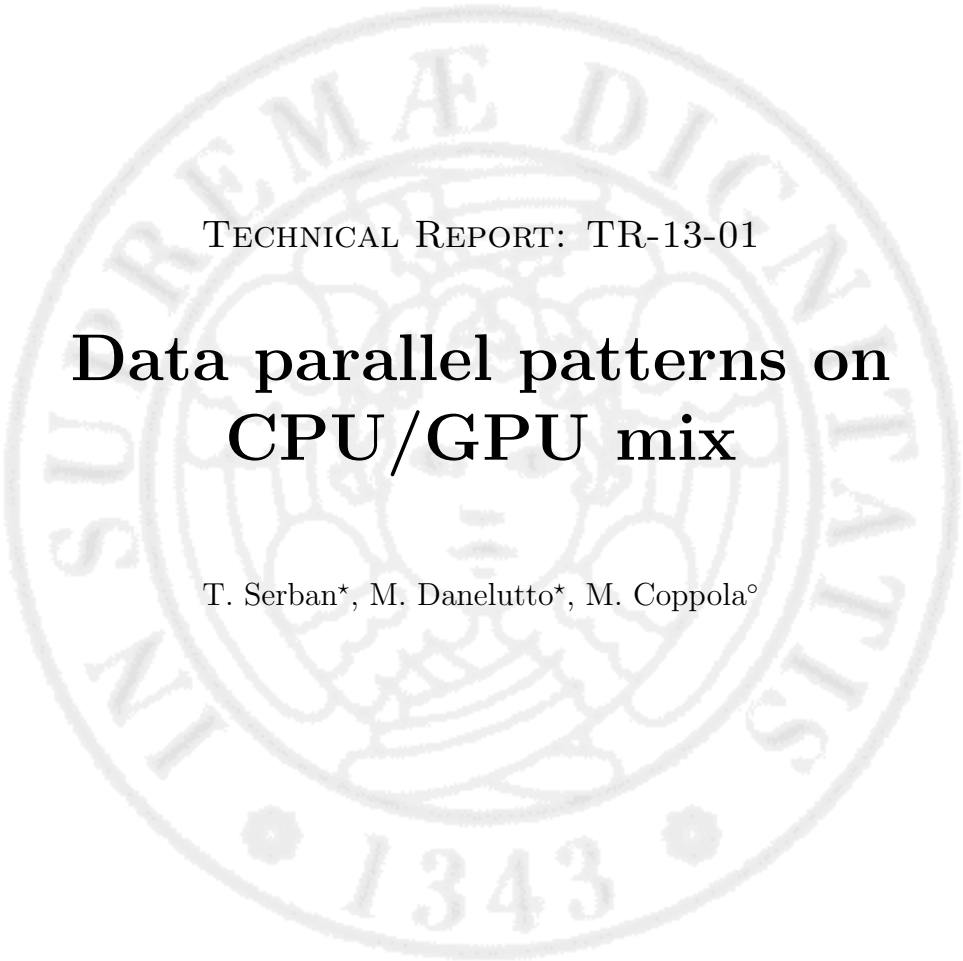
Università di Pisa
Dipartimento di Informatica



Technical Report: TR-13-01

Data parallel patterns on CPU/GPU mix

T. Serban*, M. Danelutto*, M. Coppola°

**Dept. Computer Science – Univ. of Pisa*

°*ISTI C.N.R. – Pisa*

Address: Largo B. Pontecorvo 3, 56127 Pisa, Italy. Tel: +39 050 2212700 Fax: +39 050 2212726

Data parallel patterns on CPU/GPU mix

T. Serban*, M. Danelutto*, M. Coppola°

**Dept. Computer Science – Univ. of Pisa*

°*ISTI C.N.R. – Pisa*

Abstract

We propose a model that uses a small set of quite simple parameters to devise a proper partitioning–between CPU and GPU cores–of the tasks deriving from structured data parallel patterns/algorithmic skeletons. The model takes into account both hardware related and application dependent parameters. It eventually computes the percentage of tasks to be executed on CPU and GPU cores such that both kind of cores are exploited and performance figures are optimized. Different experimental results on state-of-the-art CPU/GPU architectures are shown that assess the model properties.

Keywords: data parallelism, parallel design patterns, multicore, GPU.

1 Introduction

Current computing devices are characterized by the ubiquitous presence of multiple cores and of at least one GPU. Mobile phones currently sport quad cores with an integrated GPU, while, at the other end of the computer scenario, Top500 installations use nodes build out of multiple multicore CPUs paired with one or more GPUs [21].

Efficient programming of these computing devices poses new challenges: focus in program design shifts from "sequential efficiency" to "efficient parallelism exploitation", the available parallelism exacerbates the problems related to Von-Neumann bottleneck management, efficient programming of heterogeneous core mixes turns out to be quite hard to design and low level co-processor (GPU or many core) code and data management becomes a major source of errors and inefficiencies.

Several programming models have been recently proposed to target these new, highly parallel architectures, with the aim of providing the application programmers with more and more efficient and expressive tools and therefore of reducing the time-to-deploy of new parallel applications. POSIX threads, either explicit [8] or masked through annotations managed by proper compiling tools [9], libraries [20] and languages [7], as well as low level co-processor management languages [18, 17] and, more recently, co-processor related annotations and

compiling tools [16] have been used to program CPU/GPU mixes and currently represent state-of-the-art tools to program parallel applications exploiting CPU cores as well as the huge number of cores available on (GPU or many core) co-processors. The amount of target architecture and mechanism knowledge and the abilities required to application programmers to efficiently manage parallelism exploitation vary, depending on the tools used. At the very bottom–from the programmer productivity viewpoint–OpenCL/CUDA tools provide the finer and more efficient control mechanisms but also require a lot of expertise. At the high end of the scale, Cilk only requires application programmers to identify the precise structure of the divide&conquer (fork/join) algorithm used within an application while its compiler/run time system takes care of most of the low level details related to parallelism exploitation. However, in case of Cilk, no specific support is provided to target GPUs or other kind of many core co-processors.

Recently, parallel design patterns have been recognized to be a useful abstraction to decouple application programmer duties from hardware targeting responsibilities in the new parallel computing scenario [6] and parallel programming frameworks designed following the parallel design pattern concepts [14] and exploiting the large experience in the algorithmic skeletons field [11, 13] have been designed [2, 10, 12]. In these parallel programming frameworks, the parallel structure of the application is somehow "declared" by the application programmer by means of proper design pattern (or algorithmic skeleton) composition, while the actual parallel implementation–the one efficiently targeting the parallel architecture at hand–is dealt with by the programming tools (compiler, libraries, run time systems). The complete information available in these programming frameworks relative to the high level parallel pattern(s) used to exploit parallelism allows the system programmer to devise proper parallelism exploitation strategies, also in case of CPU/GPU core mixes.

In this work, we discuss how general purpose data parallel patterns may be compiled to a mix of CPU and GPU threads such that the overall completion time of the pattern is minimized. The main contribution of this work consists therefore in

- a model–based on both architectural and application specific parameters–suitable to compute the ratio between the number of tasks to be executed on CPU and GPU cores to optimize the completion time, and
- the design of a prototype implementing the classical map and reduce patterns which uses CPU and GPU cores according to the ratio computed by the model.

Experimental results on state-of-the-art architectures demonstrate the feasibility and the efficiency of our approach.

2 Evaluating the CPU/GPU tradeoff

When implementing data parallel applications targeting architectures with multicore CPUs paired with one or more GPU, we are faced to a couple of rather

different problems.

First of all, we have to decide whether the application parallelism is suited to be exploited on GPUs. Data parallel exploitation patterns are in general suitable to be exploited–stream parallel patterns are not suitable, instead–but the some patterns are more suitable than others, depending on the "computation intensity", that is on the amount of computation needed to process a single data item fetched from the memory sub-system.

Second, in case a pattern demonstrates to be suitable for GPU execution, we have to decide how to use the CPU while the GPU is computing, that is, we have to decide if we can use the CPU to compute part of the tasks that in principle may be offloaded to the GPU. This is a kind of *Achilles and the tortoise* problem: on suitable problems, the GPU is much faster than the CPU, but the "waiting", slower CPU may be used to compute some tasks in such a way the GPU work is somehow shortened (even if by a relatively small amount) and therefore the overall execution time–that is the time needed to compute the "ensemble" of the tasks–is shortened.

The first problem may be solved by looking at only those parallelism exploitation patterns that somehow fit the GPU execution model, that is considering data parallel patterns only. In this work we concentrate on the second problem, actually.

We therefore concentrate on the study of a suitable "cost model" that will allow us to predict the performance of a data-parallel computation that utilizes both the CPU and the GPU together in the perspective of using such cost model to devise the amount of tasks of a data parallel computation to be directed to CPU and GPU cores in order to minimize the execution time.

2.1 Assumptions

Any discussion about a cost model, especially one involving several different components operating in parallel, becomes a too large task to tackle unless some simplifying assumptions are made. These assumptions are enough for us to be able to easily reason about what is going on, without being too restrictive such that they prohibit the utilization of the cost model in practice.

The first, huge but necessary assumption, is that the computations we are going to execute on CPU and GPU cores are expressed through parallel design patters/algorithmic skeletons. This is necessary to be able to clearly identify all the "parameters" characterizing the application, namely:

- The input data and the decomposition strategy leading to the "bag of tasks" that have to executed in parallel.
- The output data and the re-composition strategy building up the overall result of the data parallel computation out of the results relative to the single task computation.
- The code computing the single task.

It is worth pointing out that these information could be available through different formalisms. In case the data parallel application is expressed/programmed using parallel patterns/skeletons, it is immediately available. In case the computation is programmed using other, lower level programming frameworks (e.g. OpenMP), it can be still derived through a moderately more complex analysis, at least in certain cases.

Another set of simple assumptions are the needed to be able to provide a suitable formalization of the data parallel process on CPU and GPU core mixes. These further assumptions may be stated as follows:

- The single system is viewed as a distributed system with two nodes: one composed of the CPU and the main memory and another one composed of the GPU and the GPU memory.
- The first system is the one initially owning the data, part of which must be sent explicitly to the second system for processing, followed by retrieval of results.
- For data copy operations between main memory and GPU memory, we assume a two-step process: setup and effective data transmission.
- We will deal mainly with a computation that takes N elements in input, performs a computation of O(N) and produces N elements in output. The non-linear case will be briefly discussed in a dedicated section.
- We assume that measuring the computation speed for one core of the CPU allows us to generalize the measurement for K cores.
- Finally, we assume that the system on which the computations run is dedicated i.e. it does not have other processes in the background competing for CPU/GPU time.

2.2 CPU/GPU abstract execution model

The typical flow of a computation involving the GPU and using the CUDA programming model is as follows (see Fig. 1): the main program thread running on the CPU invokes the CUDA kernel (after copying the appropriate data to the GPU memory), which executes in parallel on the GPU. The CPU thread then continues execution until it calls either cudaThreadSynchronize() (API call that forces the host code to block until the pending GPU computations complete) or initiates a blocking memory copy operation from the GPU memory. Both these operations block until the previously launched GPU kernels have finished executing. Figure 1 clearly suggests that the computation allocated to the CPU cannot start until the data copy to the GPU has finished. However, we should not forget that we can always delegate this task to an additional I/O thread which does not really impact on the overall performance or use asynchronous memory operations. We can thus begin to compute data on the CPU as soon

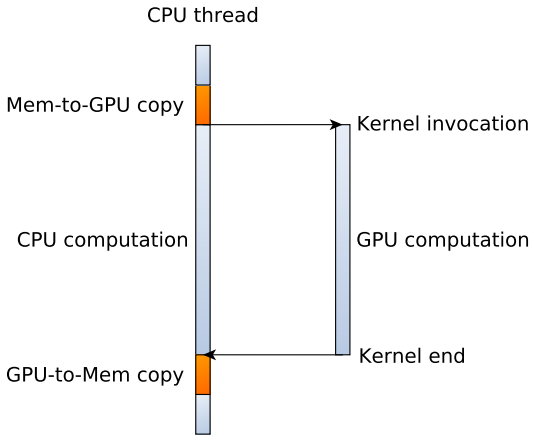


Figure 1: Execution model: CPU offloading tasks to GPU

as the data transfer to the GPU was initiated and up to the point when the transfer back of the GPU-computed chunk has finished.

2.3 Cost model

In this section we outline how suitable cost models may be developed such that in case of map-like (Sec. 2.3.1) and in case of reduce-like (Sec. 2.4 data parallel computations, an estimate of the percentage of tasks to be executed on GPU or on CPU cores may be devised.

2.3.1 Map cost model

In order to estimate the cost of the execution of tasks from data parallel patterns on CPUs and GPUs, we assume to use the parameters shown in Tab. 1

We also assume that our data parallel computation computes the same function f over all items of a data structure x. If f has type

$$f : \alpha \rightarrow \beta$$

then

$$map\ f : \alpha* \rightarrow \beta*$$

where the $*$ in the types denotes any kind of compound data structure. It is worth pointing out the data structure is preserved by the map functional, that is an input vector generates in output a vector (possibly of a different base type, e.g. vector(floats) $\rightarrow$ vector(ints)) and an input array generates in output an array. We can intuitively describe how the computation will take place. The proportion Q of data that will be processed by the GPU needs to be copied to the GPU memory, then the kernel invocation takes place and finally, when

Name	Description
D	The total amount of tasks to process
S	The size of each task (bytes).
P	The proportion of D processed by the CPU ($0 \leq P \leq 1$).
Q	The proportion of D processed by the GPU ($1 - P$).
C	The number of CPU cores available.
G	The number of GPU cores available.
T_r	Transfer rate to / from the GPU memory (in bytes/sec).
T_s	Data transfer setup time (CPU to/from GPU) (secs).
T_{f1}	CPU time to compute one task(secs).
T_{f2}	GPU time to compute one task (secs).

Table 1: Parameters of our cost model

the kernels finish executing, the output must be copied back to main memory. While this takes place, the CPU will process its own allocated part of the input (P).

Let us now express the CPU and GPU processing times as functions of the parameters in Tab. 1. First, for the data processed by the CPU, we have that the total execution time is:

$$T_1 = \frac{D \times P \times T_{f1}}{C}$$

that is, the sequential time needed to process the data divided by the number of cores. In parallel, we have two data transfers (back and forth) and a computation on the GPU. Each data transfer accounts for a:

$$t_s + \frac{size}{T_r}$$

that is for a setup time for the trasfer plus the actual transfer time, while the computation on the GPU may be evaluated with a time:

$$\frac{tasks \times T_{f2}}{G}$$

thus leading to a total time of:

$$T_2 = 2 \times (T_s + \frac{D \times S \times Q}{T_r}) + \frac{D \times Q \times T_{f2}}{G} \quad (1)$$

Since the computation is considered finished only when both the CPU and GPU have finished their assigned chunks, the cost model tells us that the total execution time will be

$$max\{T_1, T_2\} \quad (2)$$

This estimation would be reasonably accurate under the assumption that the computation takes an input of size N, performs a computation of complexity O(N) and produces an output of size N. A simplified formula is possible if we do not make the distinction between the speed of the CPU and the speed of the GPU, considering them to be equal. This, however, almost never holds simply because the CPU and GPU have totally different architectures. Furthermore, the GPU does not have to deal with context switches like the CPU does. All thread blocks run until completion and each thread has a separate set of registers. The CPU on the other hand has to do heavy context-switching and, on non-dedicated hardware that runs other processes in the background, this may have a severe impact on performance [15]. Therefore, it is safe to assume that the GPU is much faster than the CPU for raw number crunching. An accurate estimation needs to take into account the relative speeds of the GPU and CPU cores. We can now turn our attention towards determining the ideal P in order to maximize performance. From formula 2 it becomes clear that the optimal P is achieved when T1 and T2 are equal, that is:

$$\frac{D \times P \times T_{f1}}{C} = 2 \times T_s \times \frac{2 \times D \times S \times Q}{T_r} \times \frac{D \times Q \times T_{f2}}{G} \tag{3}$$

where $Q = 1 - P$. Solving the Eq. 3 we get:

$$P = \frac{\frac{2 \times T_s}{D} + \frac{2 \times S}{T_r} + \frac{T_{f2}}{G}}{\frac{T_{f1}}{C} + \frac{2 \times S}{T_r} + \frac{T_{f2}}{G}} \tag{4}$$

This is the percentage of tasks to be computed on the CPU according to our simplified model.

Going back to our $N \rightarrow O(N) \rightarrow N$ computation assumption, let us see what happens in the case that it does not hold. For instance, squaring all the numbers of an array fits this pattern. However, a computation like matrix multiplication takes in input $2N^2$ elements, has $O(N^3)$ complexity, and produces N^2 elements in output. Therefore, to better reflect this distinction in formula 2 , we should introduce two different terms to represent the amount of data that is received in input and the amount of data that is produced as output.

Obviously, each particular computation will have a slightly different version of the formula we are considering here. For example, for (square) matrix multiplication, let us denote by N the number of elements on one dimension of the three matrices (A, B and C, where $C = A * B$). In this case, the completion time would look like:

$$\begin{aligned} T = max \quad & \{\frac{N^3 \times P \times T_{f1}}{C}, \\ & 2T_s + \frac{2 \times N^2 \times S \times Q \times N^2 \times S}{T_r} + \frac{N^3 \times Q \times T_{f2}}{G}\} \end{aligned} \tag{5}$$

leading to a percentage of tasks to be executed on the CPU:

$$P = \frac{\frac{2 \times T_s}{N^2} + \frac{3 \times S}{T_r} + \frac{N \times T_{f2}}{G}}{\frac{N \times T_{f1}}{C} + \frac{2 \times S}{T_R} + \frac{N \times T_{f2}}{G}} \tag{6}$$

2.4 Reduce cost model

With similar kind of reasoning, we may develop a cost model for "reduce" data parallel computations as well.

The cost of a reduce executed on the CPU may be evaluated as:

$$T1 = \frac{D \times P \times T_{f1}}{C} + C \times T_{f1}$$

Intuitively, C cores compute a sub tree of the reduction tree[1] and then one node "reduces" the sub tree results sequentially.

The cost of a reduce executed on the GPU includes the times needed to transfer input data and results to and from the GPU:

$$T2 = (T_s + \frac{D \times Q \times S}{T_r} + T_s) + (\frac{D \times Q \times T_{f2}}{G} + log(D) \times T_{f2})$$

As for the map case, we should look for the

$$T = max\{T1, T2\}$$

The formula does not have an immediate, reasonable solution, but after some simplification[2] we eventually find as a solution:

$$P = \frac{\frac{2 \times T_s + T_{f1} \times C + log(D) \times T_{f2}}{D} + \frac{S}{T_r} + \frac{T_{f2}}{G}}{\frac{T_{f1}}{C} + \frac{S}{T_r} + \frac{T_{f2}}{G}} \tag{7}$$

2.5 Cost model assessment

We have pointed out that each particular data parallel computation corresponds to a different formula. Different types of maps produced different models. Reduce produced a model eventually different from the ones derived when maps were taken into account, which is a fairly reasonable result. However, the kind of process used to derive P in the different cases clearly identifies a general *methodology* to derive the P formula, once the base parameters of the application (data parallel pattern) and architecture (CPU and GPU) are known. We therefore expect other kind of data parallel computations may be modelled with negligible additional complexity.

3 Experimental results

In order to assess the models discussed in Sec. 2 we used three different systems equipped with GPUs, whose features are summarised in Tab. 2.

[1] we assume the reduce operator to be associative and commutative, here, as it usually happens

[2] see [19] for the complete discussion regarding the solution

	CPU	GHz	#core	GPU	#core
S1	Xeon E5400	2.70	2	Geforce GTX285	240
S2	i3 2310M	2.10	2+2 (HT)	Geforce GT540	96
S3	AMD Opteron 6176	0.8	24	Tesla C2050	448

Table 2: Configuration of the systems used for the experiments

3.1 Map

We tested the efficiency of our P estimate model using two different benchmarks:

B1 is a simple benchmark computing a $map(sq)$ over a matrix, that is computing the matrix whose elements are the square of the corresponding elements in the input matrix. In this case P is computed according to Eq. 4, sq : `float` $\rightarrow$ `float` and $map(sq)$: `float matrix` $\rightarrow$ `float matrix`.

B2 is the simplest matrix multiplication algorithm (three nested loops, no blocking, no further optimization). In this case P is computed according to Eq. 6 and the computation happens to be a map applying the inner product function (ip) to all the pairs made of a row of the first matrix and of a column of the second one. In other words, our computation is a $map(ip)$ and ip : `float vector` $\times$ `float vector` $\rightarrow$ `float`. The generation of the vector pairs and the re-construction of the result matrix is made using proper loop indexes, and in fact it does not impact the $map(ip)$ complexity.

The two benchmarks differ in their "computational intensity", i.e. in the ratio among the memory size of the data and the amount of computation performed. In B1, the amount of data is N^2 and the amount of computation is N^2 and therefore the computational intensity is 1. In B2, however, the amount of data is N^2 and the computation performed is N^3, thus leading to a computational intensity of N. This means the data transfer overhead in B2 is negligible with respect to the same overhead measured in B1.

In fact, out model estimates P as shown in Fig. 2. In case of benchmark B1 (left plot) we get values indicating that is will be convenient to execute part of the tasks on the CPU while the GPU is handling the rest of the tasks. In case of benchmark B2, however, the values indicate that possibly it is not worth executing tasks on the CPU.

Fig. 3 plot the completion times relative to execution of benchmark B1 and B2, respectively, on different target systems. While B1 plot clearly evidences the fact we achieved a decent speedup using CPU cores while running tasks on the GPU ones, the B2 plot clearly shows it is not worth using CPU cores at all.

3.2 Reduce

In order to assess the reduce cost model, we used two benchmarks, differing in the reduce operator. We choose fine grain operators, such as sum (benchmark

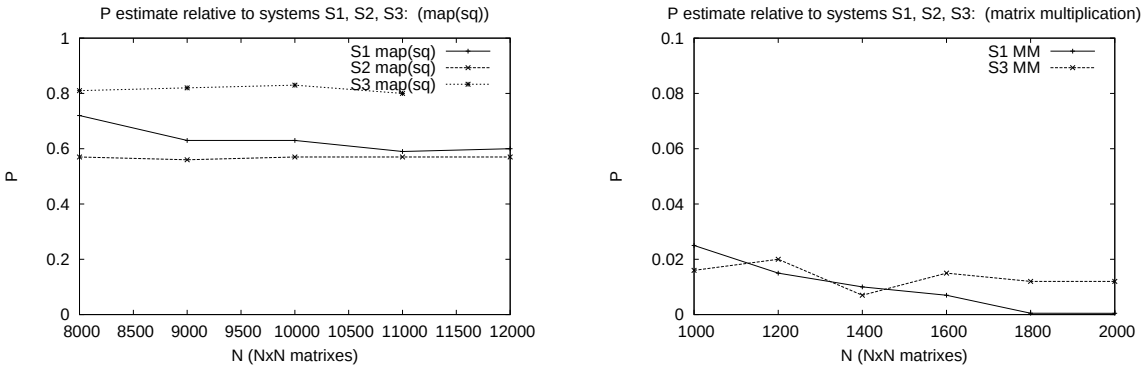


Figure 2: Model assessment: P values for different benchmarks (map(square), left, matrix multiplication, left) and target architectures (right) (Sx plots in the two graphs)

R1) and minimum (benchmark R2).

The kind of results achieved are shown in Fig. 4.

4 Exploiting results: a FastFlow CPU/GPU map module

After verifying that our simple model was actually producing good results, we pick up a state-of-the-art structured parallel programming framework and we integrated in in the framework a data parallel algorithmic skeleton that uses our model(s) to properly schedule data parallel tasks to CPU/GPU core mixes. As a framework, we choose FastFlow, which is being developed by Computer Science departments at our university and at the university of Torino [5, 4, 1]. The prototype implementation we are going to discuss in this Section has been developed in the perspective of investigating the feasibility of a completely transparent and efficient usage of GPU and CPU cores once the data parallel pattern used in the application has been clearly qualitatively identified by the application programmer.

According to the description of FastFlow written by its authors [1]

> FastFlow is a parallel programming framework for multi-core platforms based upon non-blocking lock-free/fence-free synchronization mechanisms. The framework is composed of a stack of layers that progressively abstracts out the programming of shared-memory parallel applications. The goal of the stack is twofold: to ease the development of applications and make them very fast and scalable.

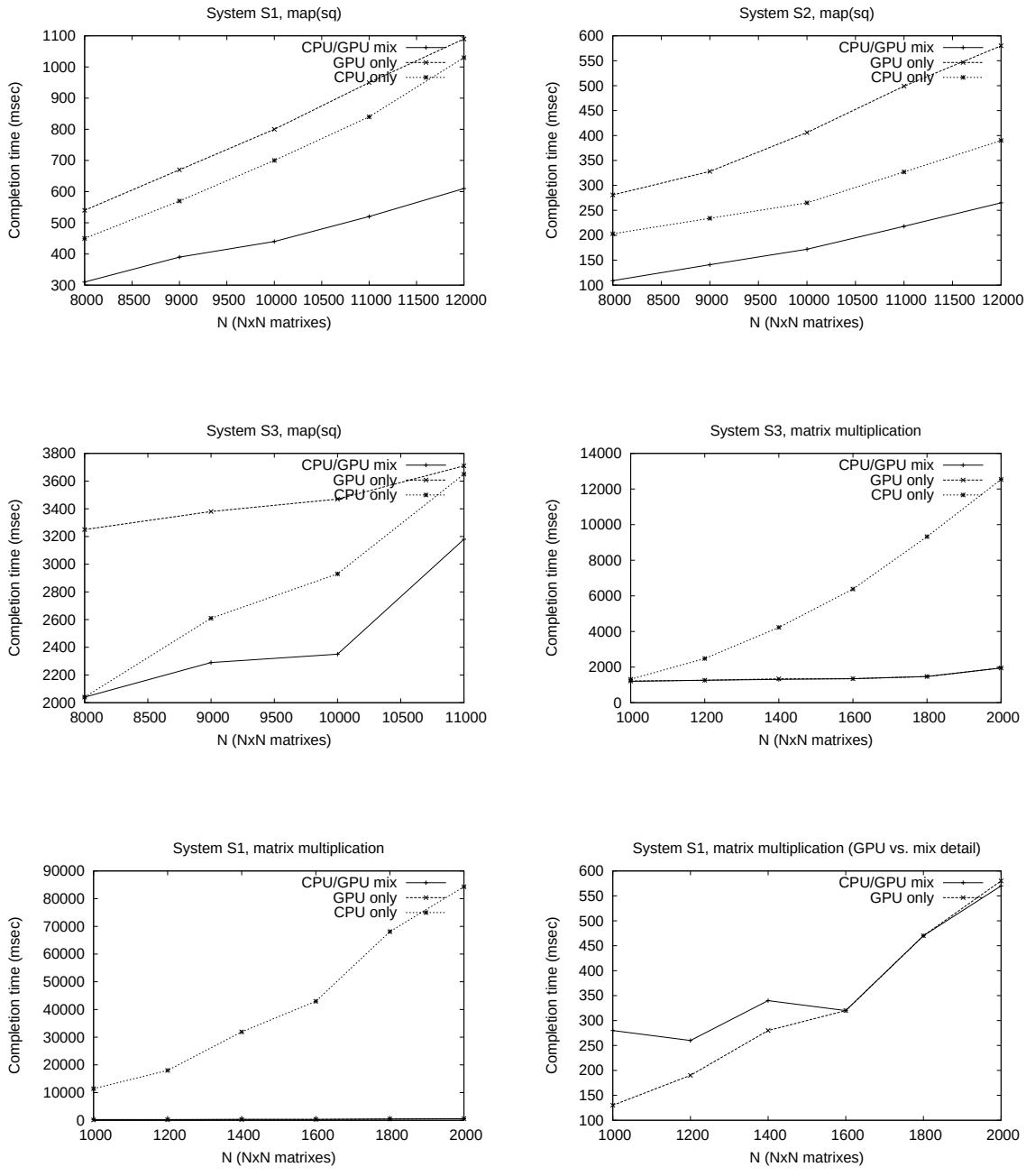


Figure 3: Model assessment: map model

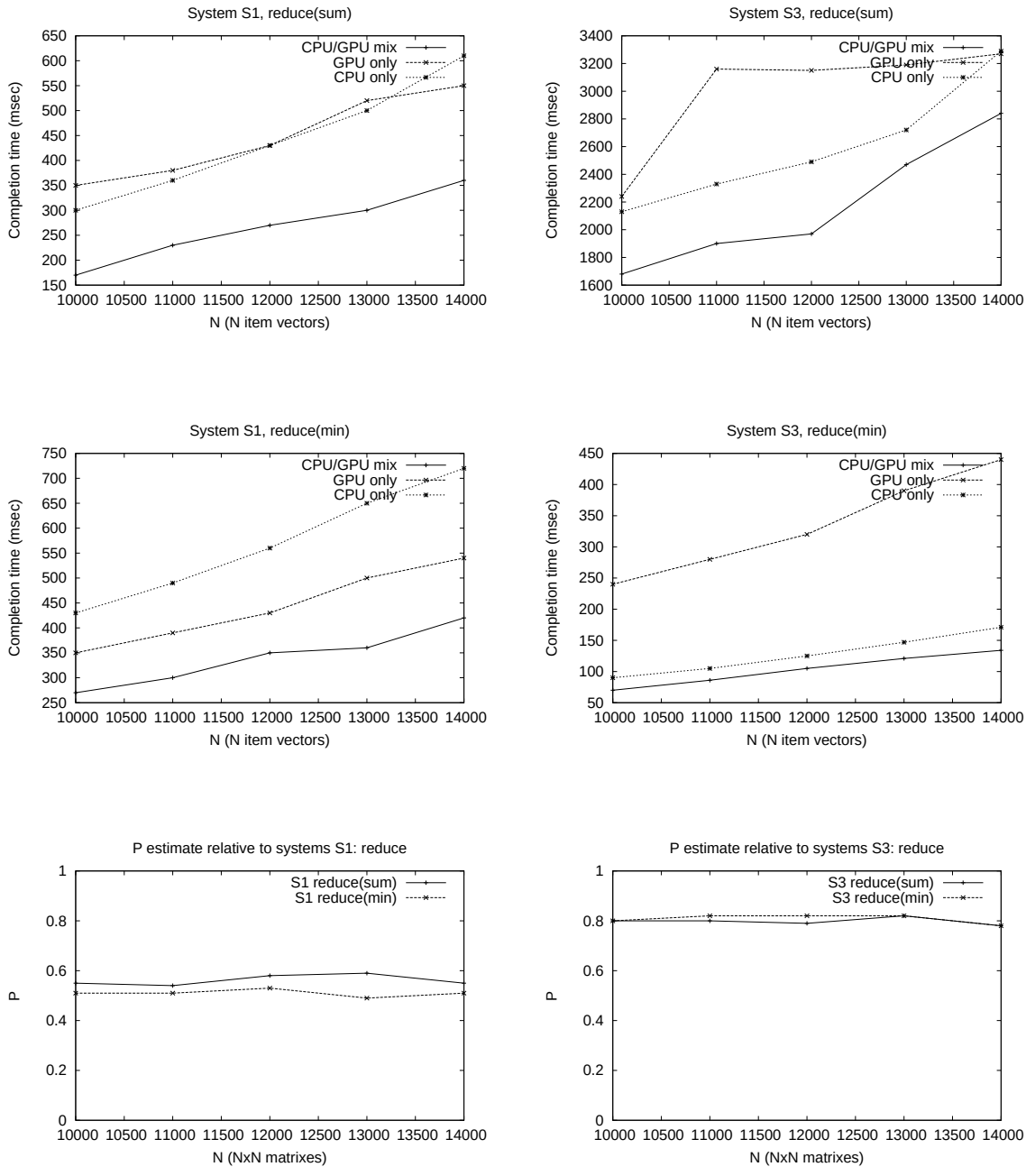


Figure 4: Model assessment: reduce

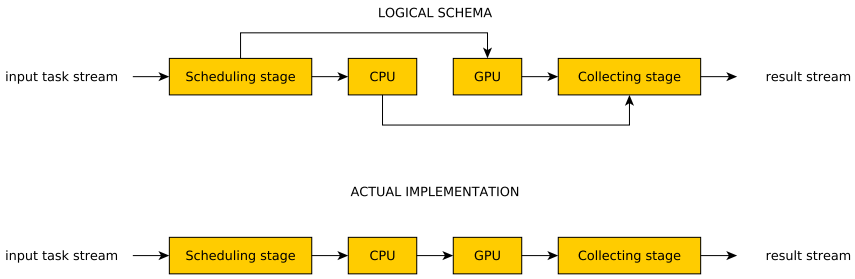


Figure 5: Intelligent FastFlow map (reduce) module: logical (top) and actual (bottom) implementation schema

> FastFlow is particularly targeted to the development of streaming applications.

In particular, the FastFlow programming framework provides the application programmers with a *pipeline* and a *task farm* skeleton, that can be used, alone or in combination, to model the stream parallel structure of the application at hand.

Thinking to our map and reduce patterns and to their cost models, we decided to use the FastFlow pipeline to implement a map and a reduce skeleton where (see Fig. 5):

- A first stage (the *Scheduling* stage) computes the cost models, after a specific profile phase, and schedules task sets to either the second or the third stage.
- The second stage (the *CPU* stage, either map or reduce) computes the P portion of the tasks. This stage is a parallel stage, targeting CPU cores through OpenMP.
- The third stage (the *GPU* stage, either map or reduce) computes the Q portion of the tasks. This stage ia actually a front-end for the GPU.
- The fourth stage (the *Collecting* stage) rebuilds the map (or reduce) results out of the results computed on CPU and GPU cores.

The resulting pipeline looks like a little bit odd, with the CPU stage not communicating directly to the following stage, i.e. the GPU one. In reality what happens is that the communications of the scheduling stage to the GPU stage happen through CPU stage, and communications of the CPU stage to the collecting stage happen through the GPU stage. This because what we move are shared memory pointers rather than actual data exploiting the ultra-efficient FastFlow communication infrastructure that succeeds moving a message in close to nano second time [3].

While implementing the schema in Fig. 5, we solved different problems:

```
#define define_map_gpu(func_name, out_type, in_type, arg_name) \
   __device__ out_type func_name(in_type arg_name); \
   __global__ void map_kernel(in_type *input, out_type *output, \
                              size_t size) { \
       int blockId = blockIdx.y * gridDim.x + blockIdx.x; \
       int k = blockId * blockDim.x * blockDim.y + threadIdx.y \
               * blockD im.x + threadIdx.x; \
       if(k < size)  output[k] = func_name(input[k]); } \

       out_type* func_name(in_type* input, size_t size) { \
          const int CHUNK_SIZE = 128 * 1024 * 1024;\
          out_type* output = new out_type[size]; \
          int num_items = CHUNK_SIZE / sizeof(in_type); \
          int chunks = (size + num_items - 1) / num_items; \
          dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE); \
          int threadsPerBlockTotal = BLOCK_SIZE * BLOCK_SIZE; \
          int numBlocksX = 256; \
          int numBlocksY = (num_items + threadsPerBlockTotal * \
                            num BlocksX - 1) / (threadsPerBlockTotal
                            * numBlocksX); \
          dim3 dimGrid(numBlocksX, numBlocksY); \
          in_type *d_input;\
          out_type *d_output;\
          cudaMalloc(&d_input, num_items * sizeof(in_type)); \
          cudaMalloc(&d_output, num_items * sizeof(out_type)); \
          for(int ind = 0; ind < chunks; ind++) {\
             if(ind == chunks - 1) {\
                int remains = size % num_items;\
                cudaMemcpy(d_input, input + num_items * \
                ind, remains * sizeof(in_type), \
                cudaMemcpyHostToDevice);\
                map_kernel<<<dimGrid, dimBlock>>>(d_input, \
                                                  d_output, remains);\
                cudaThreadSynchronize();\
                cudaMemcpy(output + num_items * ind, d_output, \
                           remains * sizeof(out_type), \
                           cudaMemcpyDeviceToHost);\
             } else { \
                cudaMemcpy(d_input, input + num_items * \
                           ind, num_items * sizeof(in_type), \
                           cudaMemcpyHostToDevice);\
                map_kernel<<<dimGrid, dimBlock>>>(d_input, \
                                        d_output, num_items);\
                cudaThreadSynchronize();\
                cudaMemcpy(output + num_items * ind, d_output, \
                           num_items * sizeof(out_type), \
                           cudaMemcpyDeviceToHost);\
             }\
          }\
          cudaFree(d_input); cudaFree(d_output); \
          cudaError_t error = cudaGetLastError();\
          const char* lerror = cudaGetErrorString(error);\
          cout << lerror << endl;\
          return output; \
       } \

__device__ out_type func_name(in_type arg_name)
```

Figure 6: Sample CUDA code for the map template

- We managed to use a small fraction of the data parallel tasks to *profile* an estimate of the T_{f1} and T_{f2} parameters. This phase computes actual results contributing to the overall result of the data parallel skeleton. However, the tasks are computed on both CPU and GPU to figure out the T_{fx} estimate.
- We implemented the pipeline in such a way the Scheduling stage flags messages directed to the CPU stage as either "cpu tasks" or "gpu tasks". The CPU stage computes the "cpu tasks" and simply passes "gpu tasks" to the next stage. Similarly, CPU stage generates "result" messages that are simply routed to the Collecting stage by the GPU stage.
- CPU stage uses OpenMP to compute tasks. Tasks are usually made of a sequence of data items such as $\langle x_1, \ldots, x_n \rangle$ which are subject to a computation such as $\forall i \in [1, n]\ y_i = f(x_i)$ or $\forall i \in [1, n]\ sum = sum \oplus f(x_i)$ The two loops are parallelized through a `#pragma omp parallel for`. Similarly, the GPU stage uses CUDA kernels on the GPU to execute tasks.
- In order to be able to use the same "business code" for f and $\oplus$ in the two cases (CPU/OpenMP and GPU/CUDA kernel), we ask the programmer to write f as a standard C function. CPU OpenMP code simply invokes the f code, while in order to execute (bunch of) f computation on the GPU, the f code is prefixed with the `__device__` keyword (which tells the compiler to make the code available/compiled on the GPU) and eventually the "map" computation is executed using a code such as the one shown in Fig. 6.

4.1 Experimental validation

Our FastFlow module implementation had been tested on the target architecture S3 (see Tab. 2). We run two experiments, one applying a map of a function computing 8K trigonometric functions out of each element of a float vector, and the other one findind the minimum of a float vector. Fig. 7 shows the results achieved, that are fully in line with those discussed in Sec. 3.

5 Conclusions

Considering the theoretical and practical results highlighted in the previous sections, we can conclude that we have an encouraging start towards developing a cost model that can accurately predict the way computations should occur on system having both a CPU and a GPU. Although not entirely accurate, the model we have so far is good enough to predict potential running time reductions by additionally using the GPU. It can, of course, be improved in several ways, including by taking into account the behavior of the memory hierarchy as well as a more detailed analysis of the GPU execution model. Having said this, there may be immediate practical benefits resulting from this model, especially applicable to streaming data-parallel computations:

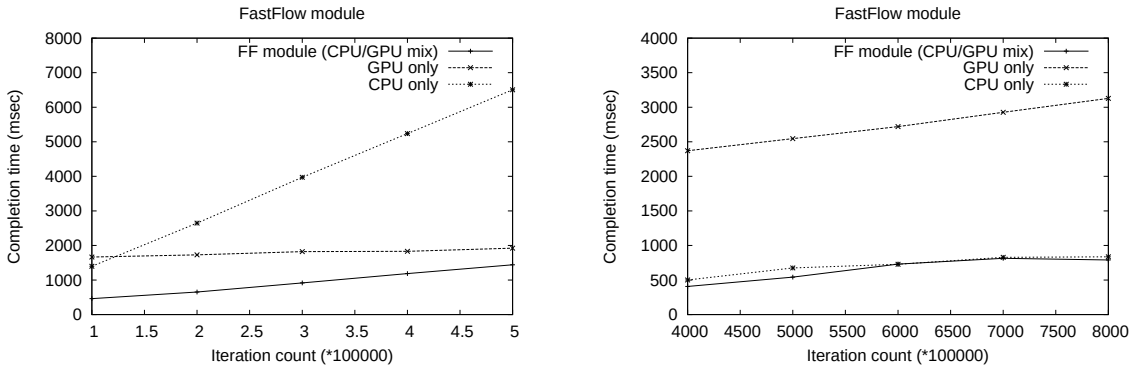


Figure 7: FastFlow module assessment: map(sin^{8K}) (left) and reduce(min) (right) results

- Firstly, we can, of course, use the GPU to improve the execution time of a single work task. After receiving a data-parallel computation from the stream, a specialized module (with or without memory of previously executed computations) can calculate the value of P and determine how to best exploit the system (see Sec. 4).
- Secondly, if actually splitting the computation is too much trouble or incurs too much overhead, the value of P can be used in a more transparent manner simply as an "affinity index (since sometimes it tends to be inaccurate) linking a particular computation to a particular execution unit (GPU or CPU). Thus, if the value of P is closer to 1, we should run the computation on the CPU and if it is closer to 0 we should run it on the GPU. An intelligent scheduler can be implemented to make this decision. For example, in the matrix multiplication case the GPU is very well suited to handle the entire computation, as P was generally less than 0.1. This of course, does not hold for any system, as a weak GPU and a stronger CPU will give a different value of P.

Acknowledgements

This work has been partially supported by EU FP7 STREP "ParaPhrase".

References

[1] M. Aldinucci and M. Torquati. FastFlow home page, 2012. http://calvados.di.unipi.it/dokuwiki/doku.php?id=

`ffnamespace:about`.

[2] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. Accelerating code on multi-cores with fastflow. In *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, Euro-Par'11, pages 170–181, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. An efficient unbounded lock-free queue for multi-core systems. In *Proc. of 18th Intl. Euro-Par 2012 Parallel Processing*, LNCS, Rhodes Island, Greece, August 2012. Springer. To appear.

[4] Marco Aldinucci, Marco Danelutto, and Massimo Torquati. Fastflow tutorial. Technical Report TR-12-04, Università di Pisa, Dipartimento di Informatica, Italy, March 2012.

[5] Marco Aldinucci, Massimiliano Meneghin, and Massimo Torquati. Efficient smith-waterman on multi-core with fastflow. In Marco Danelutto, Tom Gross, and Julien Bourgeois, editors, *Proc. of Intl. Euromicro PDP 2010: Parallel Distributed and network-based Processing*, Pisa, Italy, February 2010. IEEE.

[6] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[7] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[8] Richard H. Carver and Kuo-Chung Tai. *Modern Multithreading: Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs*. Wiley-Interscience, 2005.

[9] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[10] Philipp Ciechanowicz, Michael Poldner, and Herbert Kuchen. The mnster skeleton library muesli: A comprehensive overview. ERCIS Working Papers 7, Westflsche Wilhelms-Universitt Mnster (WWU) - European Research Center for Information Systems (ERCIS), 2009.

[11] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406, March 2004.

[12] Johan Enmyren and Christoph W. Kessler. Skepu: a multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[13] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, November 2010.

[14] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004.

[15] nVidia. CUDA C best practices guide, 2012. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

[16] OpenAAC home page, 2012. `http://www.openacc.org/`.

[17] OpenCL home page, 2012. `http://www.khronos.org/opencl/`.

[18] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[19] T. Serban. Data Parallel Patterns targeting CPU/GPU mix, 2012. Graduation thesis for the Master in Computer Science and Networking, Univ. of Pisa and Scuola Superiore St. Anna.

[20] Thread building block home page, 2012. `http://threadingbuildingblocks.org/`.

[21] Top500 home page, 2012. `www.top500.org`.