CPU+GPU Load Balance Guided by Execution Time Prediction

Jean-François Dollinger
University of Strasbourg, CNRS, INRIA (France)
jean-francois.dollinger@inria.fr

Vincent Loechner
University of Strasbourg, CNRS, INRIA (France)
vincent.loechner@inria.fr

ABSTRACT

We contribute a method to jointly use CPU and GPU in order to execute a balanced parallel code, automatically generated using polyhedral tools. To evenly distribute the load, the system is guided by predictions of loop nest execution times. Static and dynamic performance factors are modelled by two automatic and portable frameworks targeting CPUs and CUDA GPUs. The prediction methods comprise three parts: *static* code generation, *offline* profiling and *online* prediction. There are multiple versions of the loop nests, so that our scheduler balances the load of multiple combinations of code versions and selects the fastest before execution. This proposal is validated on the polyhedral benchmark suite, showing that CPU+GPU load balance is maintained and overhead is minimal.

1. INTRODUCTION

Efficient exploitation of heterogeneous computing resources is a difficult problem, especially when the processing units (PUs) run different compilation and runtime environments, in addition to different hardware. On multicore-CPUs, efficient computing relies on parallelization, load balance, cache locality exploitation, low-level optimizations. Efficient use of GPUs requires optimization of memory transfers between host and device, distribution of the computations on a grid of SIMD blocks with limitations on their sizes, explicit memory hierarchy exploitation, global memory accesses coalescing, etc.

Scientific codes are sensitive to their dynamic context. Dynamicity arises for two main reasons: the execution environment variations (*e.g.* hardware characteristics and availability, compiler optimizations) and input data size variation (*e.g.* from a call to a function to another). On the other hand, compilers have to take static decisions to generate the best possible performing code. But, as a result of the dynamic context, they miss many optimization opportunities.

In this work, we aim to address these issues automatically, namely to generate efficient code, that will run on multiple PUs and fully exploit the hardware, in a dynamic context. The most difficult problem is to achieve load balance between heterogeneous PUs. We rely on execution time predictions on each PU, based on a static code generator, an offline profiling and a runtime prediction and scheduling. Our current development platform targets shared memory cores (one- or multi-socket multicore CPUs) and one or multiple CUDA GPUs.

For achieving execution time prediction and distribution of computations, we target static control parts (SCoPs) of programs [5]. Computation distribution is made possible by polyhedral dependence analysis and scheduling, through Pluto [5] or PPCG [20] for instance. The outermost parallel loops are chunked into controllable size partitions and executed independently on different PUs.

The execution time of a chunk on a given PU is predicted at runtime, at each run of the target code. This computation is based on: (1) number of iterations and accessed data, evaluated using polyhedral tools; (2) the average execution time per iteration and per accessed data, based on tables that are generated automatically during profiling and depending on the context of the execution (number of blocks on GPUs, load balance between cores on CPU). Finally, load balance between different PUs is obtained by adjusting the size of the chunks such that their predicted execution times are as close as possible.

The main contribution of this work is an automatic framework for data-parallel workload partitioning and balancing on CPU+GPU systems. The runtime implements a low overhead dynamic scheduler, driven by pre-collected profiling data and by the program parameters. Completely ineffective PUs are automatically eliminated according to the prediction of their performance. Our system integrates a multiversioning mechanism capable to select the best performing combination of code versions, differing by their performance characteristics. Moreover, our implementation combines automatic polyhedral tools to tackle heterogeneous architectures (CPUs and GPUs) transparently.

A typical use-case of this framework is to compile a library, such as BLAS. Compile and profile time is not an issue, but performance on the machine hardware and adaptivity to different parameters are crucial: the user of this library wants to exploit efficiently all available resources of his machine, in all the calls he will make to the library, possibly using different parameters. For compiling such a library in our framework, one would first mark all computationally intensive SCoPs in the source, then call the script that generates the profiling and executable codes and distribute

IMPACT 2015
Fifth International Workshop on Polyhedral Compilation Techniques
Jan 19, 2015, Amsterdam, The Netherlands
In conjunction with HiPEAC 2015.

http://impact.gforge.inria.fr/impact2015

them. The user of the library would run the profiling code on his computer at installation time, and then the code of the library would automatically adapt to the hardware environment (multicore CPUs, number of GPUs, relative performance) and to the dynamic parameters of each call to the library at runtime.

In Sect. 2 we present some relevant related work. We detail the procedures of code generation in Sect. 3, profiling and execution time prediction in Sect. 4 and runtime scheduling in Sect. 5. In Sect. 6 we validate our proposal by running the polyhedral benchmark suite, and verifying that load balance is achieved with varying input data size on a testbed with a multicore CPU and one to four GPUs.

2. RELATED WORK

Our scheduling algorithm relies on [17] and [9] methods for predicting execution times on CPU and GPU (respectively). They were originally designed to portably select the best performing code amongst multiple, semantically equivalent versions. Python scripts generate profiling and prediction code. The profiling step simulates the target code execution and evaluates the throughput of host and device communications. At runtime, as the execution context is known, approximated execution times are computed and used as predictions.

Peng Di et al. [7] propose a technique to automatically select a good-performing tile size for codes targeting GPUs. They primarily focus on doaccross loops for which they extract inter-tile and intra-tile wavefront parallelism. Tile size selection is performed by comparing predicted execution times for different configurations of tile sizes. Even though the prediction model accurately approximates execution times, the generated code must follow a specific pattern and host-device communications are not considered.

Introduction of directive-based languages, such as OpenACC [14] or HMPP [8] have leveraged the programmer task to exploit accelerators. Although they provide means to easily target a single device, they lack in straightforward distribution of the computations onto all the available PUs. Komoda et al's work [11] is a first step towards using all resources of a machine with openACC. Still, it needs refinements and requires the implementation of scheduling to be efficient on heterogeneous systems.

Boyer et al.'s work [6] focuses on hardware availability and environment dynamicity while we focus on problem size influence. To train the scheduler, work groups aggregated into chunks, which size is exponentially increased at each step, are run on the target processor. Each chunk execution triggers data movements to the target processors. As an arbitrary threshold is reached the measurements are considered as relevant and the training stops. Due to its dynamic nature, the system is inclined to adapt to external performance-impacting events, for instance punctual clock rate scaling, processor load, etc. Based on the last execution times, the scheduler *linearly* dispatches the rest of the computations to the PUs. The authors evaluated their framework for problem sizes occupying the whole memory. We show in this paper that the execution times of chunks do not vary linearly with their size.

StarSs [15] extends the OpenMP directives with constructions to specifically handle and offload code portions. The scheduling strategy relies on a training phase during which tasks are run on the available processors, and it builds affinities between processors and tasks. Then, the tasks are associated to the processors so that the computational load and the overall execution time is minimized. This coarse-grained approach is best effort as it does not guarantee load balance.

Recently, there has been a lot of interest in providing multi-GPU accelerated mathematical libraries. *CuBLAS-XT* [13] is an Nvidia multi-GPU capable BLAS library, however, it is not clear whether load balance is maintained on heterogeneous GPUs configuration. MAGMA [18] is a similar linear algebra library that manages heterogeneous CPU+GPU platforms by relying on dynamic scheduling provided by StarPU.

The StarPU runtime system [2] schedules tasks onto the available computing resources. The programmer has to write the tasks as codelets, provide their dependencies, and decide which scheduling policy to employ. To maintain load balance, the *heft-tm* strategy [1] relies on a history-based time prediction to assign tasks to PUs, so that execution time is minimized. In order to characterize performance, multiple actual target code executions are required, for all execution contexts. Conversely, our framework profiles the code before the first execution of the application, thus immediately enabling maximum performance for any parameters values at runtime.

Overall, in comparison to the task-based StarSs and -StarPU systems, which may stall on dependencies, our runtime makes immediate and continuous use of all the hardware resources; and it is fully automatic, once the programmer has marked the region of (sequential) code of interest with a pragma. On the other hand our framework handles only SCoP codes, which can be handled by optimizing polyhedral compilers, while StarSs and StarPU can handle any parallel code that the programmer writes.

In the HDSS scheduling scheme [3] loops are decomposed into chunks. The first chunk performs a training phase to model the performance for each compute device with a logarithmic function. This function is then used to determine a weight, controlling the chunk size associated to each PU. While offering opportunities to adapt to external performance factors, training is required for each run, which may induce a large overhead, especially on short and frequently called codes. The performance model is coarse grain and may be imprecise on GPUs (see for example Fig. 2). Conversely, our scheduler does not require an online training, and partitions precisely the whole parallel loop nest before its execution.

In the SKMD system [12] the loop partitioning is computed by generating multiple combinations of compute device workload until they minimize execution time. To predict the performance, a table stores performance values expressed in work groups per millisecond. As mentioned in the paper, the target codes are hand-written. Moreover, host-device bandwidth is considered constant, which may imply prediction inaccuracies.

3. CODE GENERATION

In this section we present how we automatically generate OpenMP code for CPU and CUDA code for GPU. A set of python scripts orchestrates the code generation process for its execution on a heterogeneous configuration. To provide source code analysis and modification capabilities, the scripts implement a wrapper on pycparser [4], a C parser written in python. We extended it to handle C for CUDA

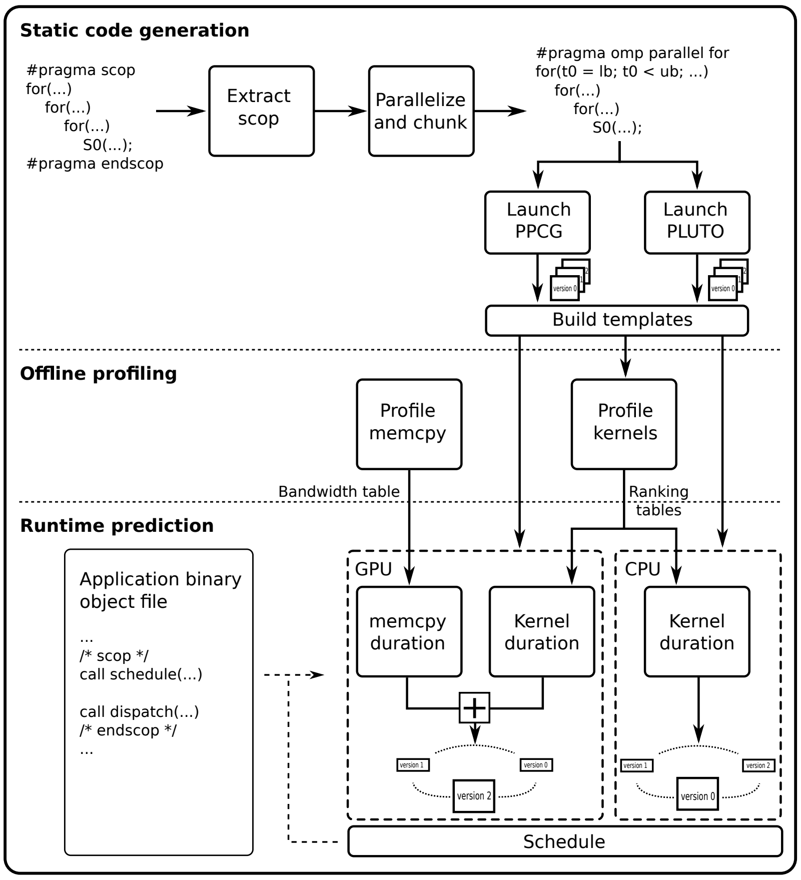

Figure 1: Global framework workflow overview.

and a pragma to mark the regions of code of interest. To build the target code, the generator makes extensive use of template files. For specializing the parallel loop nests we rely on Pluto [5] and PPCG [20], two source-to-source polyhedral compilers. Both compilers generate optimized parallel code from static control loops written in C. PPCG specializes in CUDA host and device code generation, from sequential loops.

During a first stage, the code is parallelized using the *OpenMP C* backend of PPCG. Artificial parametric loop bounds are injected in the parallel loops to control the iteration domains. This enables the iteration domains of the parallel loops to be cut into chunks. At runtime, each chunk will be assigned a PU and sized to ensure load balance. To this end, Pluto and PPCG generate specialized versions of the parallel chunks, optimized towards CPU and GPU. As the chunks can be executed in any order, the code semantics is preserved and they can be safely distributed on the available PUs. Our scripts also compute the geometrical bounding box of the accessed arrays to generate minimal data communications between CPU and GPUs. This operation is performed through calls to the *isl* library [19] via its python bindings *islpy* [10] and a loop nest polyhedral representation extracted with *pet* [21].

One single version of a code may not perform well under all circumstances. Multiversioning is an adaptive technique consisting in generating semantically equivalent versions of a code, differing by their performance characteristics. The versions are built by PPCG and Pluto, launched with version-specific arguments, such as the tile size, tiling level, block size and schedule. The scripts generate all the version combinations by successively assigning the parallel and prediction code function pointers to internal structures. For each combination it places a call to the scheduler and the selection function. The combinations are scheduled at runtime, supposedly the best is executed on CPU+GPU.

4. EXECUTION TIME PREDICTION

Predicting execution times in heterogeneous contexts is a difficult problem. Performance of a code may vary according to compiler transformations, input dataset, hardware specifications and execution environment interactions. For this purpose, we rely on two techniques providing accurate execution time predictions of affine parallel loop nests on CPU and GPU.

Both frameworks share the same architecture as depicted in Fig. 1: static code generation, offline profiling and online prediction. Annotated loop nests are automatically extracted from the original source code and parallelized. An artificial lower- and upper-bound is injected for partitioning the outermost parallel loop into chunks. An architecture specific optimized version of the parallel loop nests is then produced by Pluto (for CPU) or PPCG (for CUDA GPU). A set of Python scripts generates the profiling and the prediction codes (*build templates*).

Prior to any execution of the target application, an *offline* profiling phase executes the annotated code on the target architectures. To perform relevant measurements, iteration domain sizes are controlled by adjusting loop nest parameters. The profiler outputs *ranking tables* filled with average statement execution times per iteration. We will abbreviate them to *execution times per iteration* in the following. An additional micro-benchmark builds up a *bandwidth table* modelling PCIe communication throughput.

At runtime, as the execution configuration is known, the scheduler calls the prediction functions by passing the parallel loop nests parameters as arguments. Simplified versions of the original loop nests, called *prediction nests*, compute an approximation of the execution times using the offline collected profiling results. Note that the required parameters and the prediction nests may differ between different classes of PUs.

Different types of performance factors will impact the parallel loop nests execution times. *Static* performance factors are constant during the execution (*e.g.* arithmetic instructions duration). Thus, they are naturally taken into account by the profiler. *External dynamic* performance factors originate from the execution environment (*e.g.* system scheduler, (co)processor load, I/O operations) and may impact the reliability of the profiling data. To adapt to these interactions, the ranking and bandwidth tables could be readjusted by the runtime or through renewed profiling sessions. Finally, *internal dynamic* performance factors fluctuate during the execution (*e.g.* cache effects, Thread Level Parallelism) and are inherent to the targeted code and architecture. They are depending on the input dataset and they parametrize the profiling tables. Internal dynamic performance factors are thoroughly described in Subsection 4.1 for CPU [17] and in Subsection 4.2 for GPU [9].

4.1 CPU execution time prediction

Two important dynamic internal performance factors were identified on CPUs. Cache effects are evicted by exponentially increasing the iteration domain size, until measurements become stable. This method is best effort and may stop prematurely in case of local stabilization. Triangular iteration domains or non-equal partitioning of the parallel iterations imply thread load imbalance which significantly impacts a code performance. To characterize load balance, the profiler builds a ranking table parametrized by the number of active threads. This allows to characterize the execution time per iteration according to the CPU cores usage. Note that during profiling, only full tiles are considered. In average the profiler requires a few seconds to evaluate one code version.

At runtime, the CPU prediction code computes the number of iterations per thread. Then, it determines the number of iterations processed in parallel, per thread quantity. The corresponding execution time per iteration is fetched from the ranking table and multiplied by the number of iterations to get the approximated loop nest execution time. The calculation can be synthesized as: $time = \sum_{i=1}^{C}(it_i - it_{i+1}) * rk_i$, where $time$ represents the approximated execution time of the loop nest, C the total number of cores, it_i the number of iterations per thread quantity i and rk_i the execution time per iteration for i active threads.

4.2 GPU execution time prediction

To make accurate predictions, distinction is made between

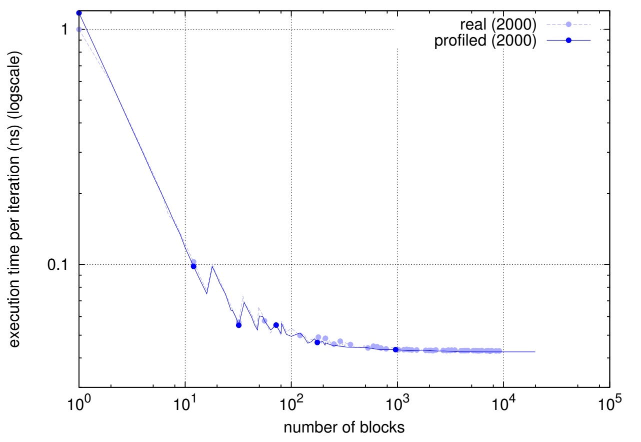

Figure 2: Comparison between profiled and measured execution times per iteration for *gemm*.

parallel and *sequential* loop nest parameters. The parameters that appear in parallel loop bounds are called the *parallel parameters*. All the other ones are *sequential parameters*: they bound sequential loops only, enclosing or enclosed in parallel loops.

CUDA Thread Level Parallelism (TLP) latency hiding mechanisms have a strong incidence on performance. Relatively to the CUDA grid size, execution times per iteration typically follow a shrinking sawtooth-shaped curve. This curve is modelled by a piecewise affine function, built on the fly by the profiler. As a certain number of blocks is reached, the execution time per iteration becomes constant. The measurements are stopped as soon as the number of blocks exceeds hardware limitations. The profiling space is pruned in real time based on the affine shape of the execution time per iteration. Also, through experiments, we noticed that the grid dimensions are commutative for performance: $perf(sz_x * sz_y) \approx perf(sz_y * sz_x)$. Cache effects are subdued by gradually increasing the size of the sequential parameters, until stabilization. Through experiments, it appeared that the sequential loop parameters had a variable incidence on performance. This mainly stems from memory access contention and bank conflicts. As their impact is moderate, their influence is modelled via a linear regression function. All in all, the ranking table is parametrized by the size of the grid and the value of sequential parameters. Figure 2 shows that the ranking table values interpolations (plain line) overlap the measurements (dashed line) and thus captures the performance variations.

PCIe bandwidth is an asymmetric (read/write) and non-uniform function of the message size. A micro-benchmark builds a bandwidth table of piecewise affine functions of the message size, for each direction of the transfer. Similarly to Fig. 2, Fig. 3 shows the bandwidth table values interpolations overlapping the measurements. The measurements are performed for different message sizes until saturation of the GPU memory. Overall, the CUDA kernel profiling duration ranges from 15 minutes to 5 hours per code version, depending on the kernel durations and on the number of required kernel runs.

The GPU prediction code is executed on the CPU. The kernel execution time prediction function seeks, through the

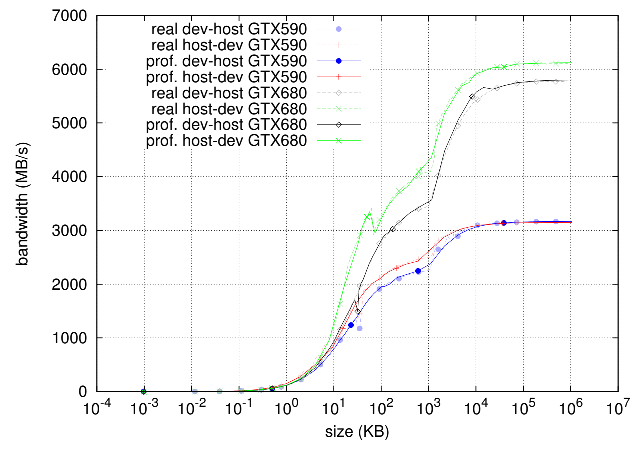

Figure 3: A comparison between profiled and effective transfer times.

values stored in the ranking table, the interval in which the kernel grid size belongs. The two linear regression functions of the sequential parameters that it gets from the table are then instantiated with an average of the sequential parameters values. We then calculate a linear interpolation to compute an accurate approximation of the execution time per iteration.

Memory transfer duration prediction takes as input the message size and the direction of the transfer. In the same way as the execution time computation, transfer time calculation requires linear interpolations on the bandwidth tables. Finally, the sum of the transfer and execution times approximates the global execution time of the kernel.

5. CPU+GPU RUNTIME

5.1 Scheduler

The scheduler implements a work sharing mechanism similar to the OpenMP static scheduling strategy. The quantity of work of each PU, controlled by the chunk size, is determined before the execution of the loop nest.

The scheduler relies on the execution time predictions to distribute iterations to the PUs. Let $T_i = t_i \times Q(P_i) + C(P_i)$ be the chunk i predicted duration, where $0 \leq i < n$, n being the number of PUs. Function $Q(P_i)$, generated with the Barvinok counting library [22], computes the number of iterations of the union of the statements iteration domains: it is a symbolic piecewise quasipolynomial instantiated with the parameters values P_i at runtime. t_i is the execution time per iteration. Function $C(P_i)$ is in charge of estimating the data transfer time on GPUs; for CPUs it returns 0.

For a given parallel loop nest, the load balance problem can be expressed as an equality between the chunks durations: $T_0 \approx T_1 \approx ... \approx T_{n-1}$. The upper parallel loop bound of each chunk i is the lower bound of chunk $i+1$. The execution time per iteration t_i of each chunk fluctuates non-linearly depending on the chunk size as described in Subsections 4.1 and 4.2. As a consequence, there is no direct method to compute the chunk sizes, but this optimization problem requires iterative refinements.

Through problem reformulation, achieving load balance comes down to make T_i tend to T_{all}/n, where T_{all} is the sum of the PUs execution times: $T_{all} = \sum_{i=0}^{n-1} T_i$. We implemented a low overhead iterative algorithm in three steps: initialization, refinement and partitioning. The refinement and partitioning stages are repeated until convergence is reached, or a maximum of 15 steps is attained. In the initialization phase, the iterations of the chunked loops are equally distributed between the PUs. No preliminary assumptions can be made concerning the execution times of the chunks.

The refining stage starts by computing each chunk execution time T_i and their sum T_{all}. Each chunk execution time proportion $R_i = T_i/T_{all}$ must tend to $o = 1/n$ to achieve load balance. Note that an optimal predicted load balance is obtained for $R_i = o$ for all i. Each chunk size is then ajusted by multiplying it by o/R_i, to get closer to optimal load balance. However, these adjustments are computed independently for each chunk, and this leads to situations where the sum of the chunk sizes is not equal to the total number of iterations. Thus, the partitioning phase normalizes the chunk sizes so that all iterations of the chunked loop are processed. Iterations eliminated by integer rounding are assigned to an arbitrary PU (the CPU by default in our current implementation).

To get rid of very inefficient PUs faster, a chunk of less than x% of another chunk size is eliminated: $x = 10$ by default in our implementation. It can be increased if energy consumption is an issue: in that case one will want to eliminate inefficient PUs faster.

Algorithm 1: Scheduler algorithm

```
; // step 1: initialize to equal distribution
$chnk\_size \leftarrow (ub - lb)/num\_pu$;
for $i \leftarrow 0$ to $num\_PU - 1$ do
    $PUs[i].lb \leftarrow i * chnk\_size$;
    $PUs[i].ub \leftarrow PUs[i].lb + chnk\_size$;
end
; // step 2: refine
for $s \leftarrow 0$ to $MAX\_STEPS$ do
    $time \leftarrow 0.$;
    for $i \leftarrow 0$ to $num\_PU - 1$ do
        $PUs[i].size = PUs[i].ub - PUs[i].lb$;
        if $PUs[i].size \neq 0$ then
            $PUs[i].time\_val =$
            $PUs[i]$.time($PUs[i].lb$, $PUs[i].ub$);
            $time \leftarrow time + PUs[i].time\_val$;
        end
    end
    for $i \leftarrow 0$ to $num\_PU - 1$ do
        if $PUs[i].time\_val \neq 0$ then
            $adjst = time/(num\_PU * PUs[i].time\_val)$;
            $PUs[i].size = PUs[i].size * adjst$;
        end
    end
    ; // normalize the chunk bounds
    $(PUs, max\_card) \leftarrow$ normalize($PUs$) for $i \leftarrow 0$ to
    $num\_PU - 1$ do
        if $PU$.card($PU$)$/max\_card < 0.1$ then
            $PUs \leftarrow$ eliminate($PUs$, $i$);
        end
    end
end
```

The full algorithm is presented in Alg. 1, and Fig. 4 shows a typical example of the scheduler steps. Two PUs are considered: 1 GPU (on the left of each couple of bars) + 1 CPU (on the right). The lower-solid bars represent the

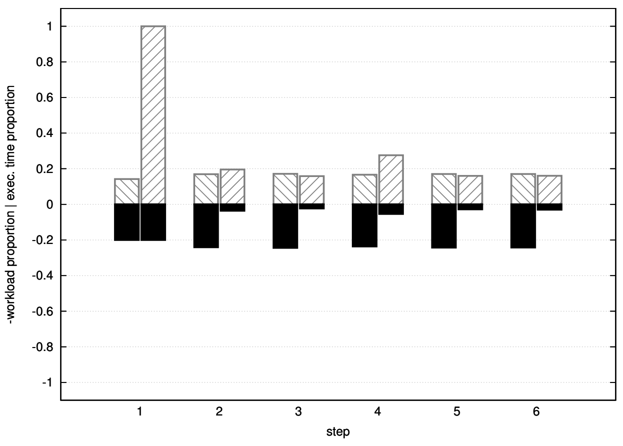

Figure 4: Example of scheduler algorithm steps for *gemm.*

size of the iteration domains of each chunk ($Q(P_i)$), and the upper-strided bars, the corresponding execution times (T_i). In Fig. 4, the GPU is assigned much more iterations for approximately the same execution time than the CPU, in 6 steps of the scheduler.

5.2 Dispatcher

The dispatcher is in charge of launching the codes on the different PUs. Each PU is assigned a thread using OpenMP parallel sections. Before launching the computation, a device initialization function is called. On CPUs it sets the number of threads required for the computation and activates nested parallelism. For GPUs, it selects the device and modifies the CUDA device scheduling policy. Indeed, we observed that the scheduling policy has an impact on the CPU threads performance. By default it will busy-wait if enough processing resources are available and yield the threads otherwise. To get rid of any overhead, we chose another strategy which blocks the polling threads until the device finishes its work (*i.e.* blocking synchronization). Due to device initialization purposes, the first called CUDA function (*e.g.* cudaMalloc(...)) consumes more time. To avoid noise in the measurements, we introduced a fake runtime call. At the end of the computation, the threads are synchronized using a barrier. All data movements are carried by the thread handling each PU.

6. EVALUATION

6.1 Benchmarks

The test platform is composed of two Asus GTX 590 plugged into an Asus P8P67-Pro motherboard. Each GTX 590 card is composed of two Fermi GPUs sharing 3 GB of GDDR5. Each graphics processor on the GTX 590 embeds a total of 512 Streaming Processors[1] ($16\ SM \times 32\ SP$). The motherboard provides a PCIe 2.0 x16 bus for connecting the peripherals. The two graphics cards individually support PCIe x16 and share half of the bus width (x8) in our configuration. The host processor is an Intel core i7-2700 (Sandy Bridge) with 4 hyperthreaded cores for which we enabled dynamic overclocking.

The benchmark programs that we run are taken from the Polyhedral Benchmark suite [16]. We used the *extra-large* dataset size by default, reducing it on some of the tested programs so that they fit the GPU memory. We did not consider some programs because they are very inefficient on GPU: the CPU version is much faster in any case, and there is no point in trying to exploit a GPU version of them. We did however include some benchmarks that fall in this category in our experiments (*gesummv*, *mvt* and *gemver*). All loop nests of depth 1 are ignored by our framework.

We compiled the benchmarks using gcc version 4.4.6 with *-O3 -march=native* optimization flags. On GPUs, the codes were compiled with the CUDA 5.5 compilation tools. The GPU on-chip memory partitioning was set to 48 KB of shared memory and 16 KB of L1 cache. In our first experiment presented in Fig. 5, only one code version was generated. To generate CUDA code, the minimum loop nest fuse flag was provided to PPCG and automatic cache management code generation was enabled. The CUDA block and tile sizes have been set to the default provided by PPCG. Communications between host and device are handled with synchronous non-pinned memory copies. Similarly, we run PLUTO with the default parameters, and disabled tiling for *mvt*, *gemver* and *gesummv* as these tiled CPU codes are strongly affected by performance fluctuations. For the multiversioning experiments presented in Fig. 7 and 9, we generated CUDA codes with couple *(block size, tile size)* equal to $(32 \times 16, 32 \times 16)$, $(32 \times 16, 64 \times 64)$ and $(16 \times 16, 16 \times 16)$, respectively for the (c1, c2, c3), (c4, c5, c6), (c7, c8, c9) combinations. CPU versions were generated with one level of tiling, of size 32, 64 and 128, respectively for the (c1, c4, c7), (c2, c5, c8), (c3, c6, c9) combinations. We averaged all measurements on five runs.

Figure 5 depicts the speedup obtained by using different combinations of PUs compared to the execution time on CPU alone or on GPU alone. Our system achieves a maximum speedup of 20x for *gemm* and a speedup of 7.1x on average comparing the best and worst execution times. These results show that *gemm*, *2mm*, *3mm*, *syrk*, *syr2k* (the five on the left of Fig. 5) better suit the GPU while *doitgen*, *gesummv*, *mvt*, *gemver* better suit the CPU. Note that *doitgen* better suits the CPU because of a lower computation time on CPU than on GPU, and not because of the data transfer times. It is interesting to notice that combined CPU+GPU execution provide noticeable benefits for three benchmarks (*syr2k*, *doitgen* and *gemver*). When the GPU version is faster, the average speedup of our system on CPU plus 4 GPUs against 1 GPU, is 3.49x and is greater than 4x for *syrk*; for the programs where the CPU version is faster, the average speedup on CPU plus 4 GPUs against 1 CPU is negligible, as in these case GPUs were generally unused. Also notice that, apart from *2mm* (for which CPU was unused), the 1 CPU + 1 GPU version is always faster to the CPU alone and GPU alone versions. Figure 6 shows the imbalance ratio between the total longest and shortest execution times of the different PUs. In *2mm* the CPU was eliminated, and in *gesummv* and *mvt* the GPU was eliminated[2]. Figure 6 shows an average of 10% load imbalance. The imbalance is mostly due to prediction inaccuracies rather than bad scheduling decisions. In fact, the sum of the absolute

[1]SM: Streaming Multiprocessors, SP: Streaming Processors

[2]Figure 5 shows small variations of the CPU+nGPU versions due to measurements inaccuracies.

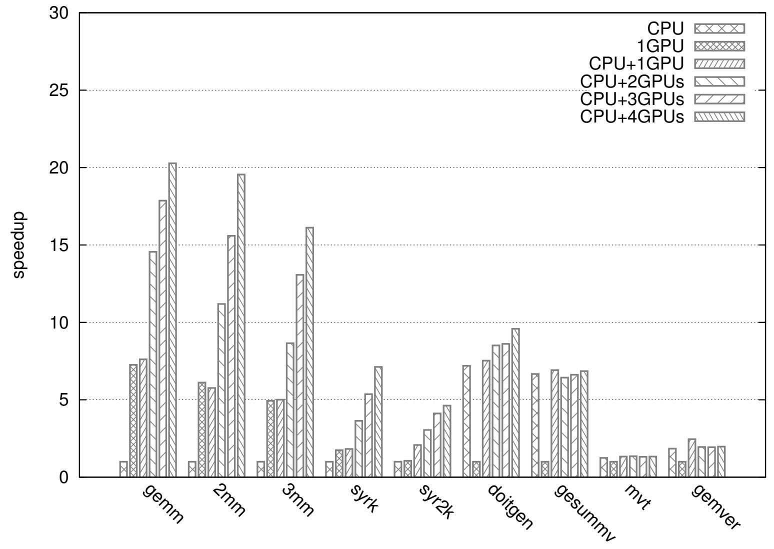

Figure 5: Speedup to execution time on CPU or GPU alone.

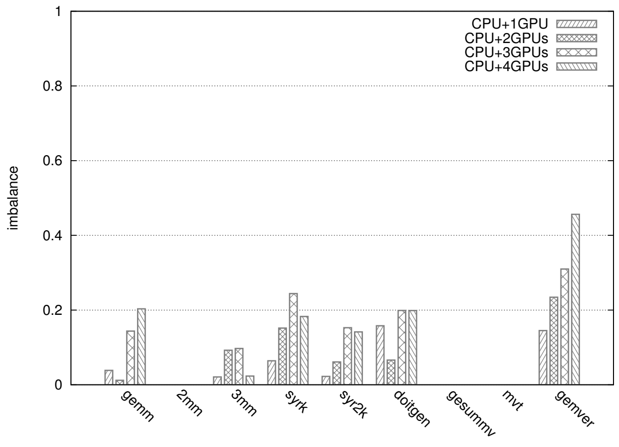

Figure 6: Execution time imbalance ratio for several combinations of PUs.

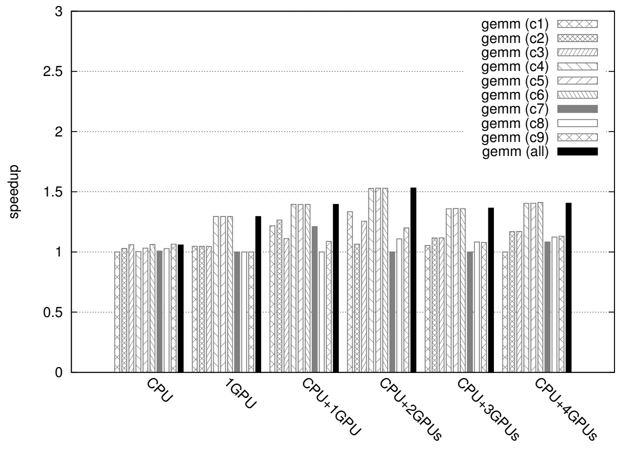

Figure 7: Speedup to execution time of slowest code version combination for *gemm*.

prediction errors drives imbalance. In particular, this affects fast codes with shallow loop nests, such as *gemver* for which the imbalance peaks at 45% for CPU+4GPUs. These codes are strongly affected by multiple running GPUs.

Gesummv, *mvt* and *gemver* are also noticeable due to the elimination of the GPU for certain loop nests computation. For *gesummv* and *mvt* the whole computation is run by the CPU. This happens when the ratio between the communication and the computation times is too high. Also, reducing the problem size may eliminate non-necessary PUs. Remark that there are performance interactions between CPU and GPU, especially on the host code side as memory transfers get through the CPU caches. As an example, the spinning CUDA runtime scheduling policy, set by default, impacts CPU performance by 30% on *gemm* with 4 GPUs running. The opposite effect of using blocking scheduling is that small codes repeatedly executed tend to run slower.

Our system overhead (including all prediction and scheduling calls) is low: it caps at $2ms$ for *doitgen*, that is to say 0.02% of the execution time. It tends to average below $1ms$ for most of the codes, which is a reasonable figure for codes that are suited to run on GPUs (executing for more than a second).

6.2 Multiversioning

Our framework is able to generate multiple versions of the CPU and GPU codes and to select the best performing combination at runtime. The scheduler is called for each combination and returns its predicted execution time. The runtime selects the scheduled combination of versions which minimizes the execution time. As the number of combinations grows exponentially, we limited our experiments to 3 versions per PU (9 combinations). Note that for *syr2k*, the maximum scheduler overhead was of $450 * 9 = 3600\mu s$, that is to say less than 0.01% of the execution time. Figures 7 and 9 show speedups to the slowest combination of versions. Usefulness of multiversioning is emphasized by the perfor-

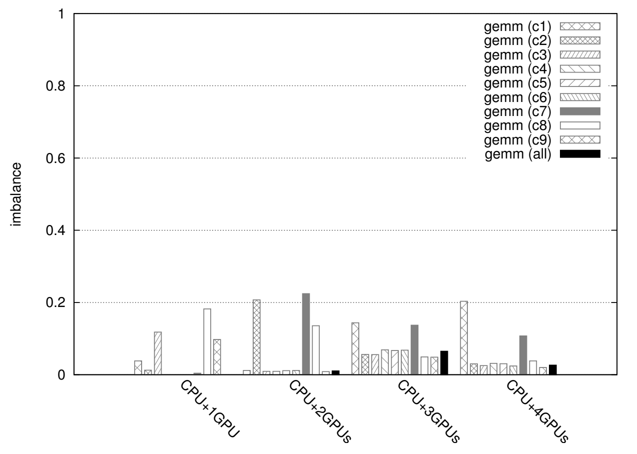

Figure 8: Execution time imbalance ratio for several combination of code versions for *gemm*.

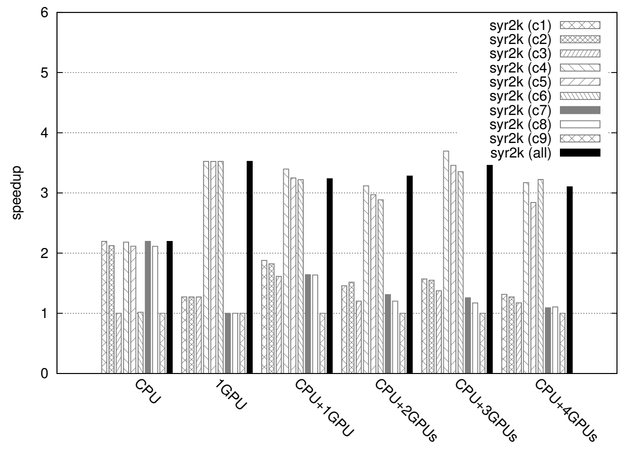

Figure 9: Speedup to execution time of slowest code version combination for *syr2k*.

mance variations of the *GPU only* executions in the (c1), (c4), (c7) combinations. The (all) bars refer to the final combination selected by our runtime system. At best, it was able to achieve a 1.53x speedup for *gemm* and a 3.46x speedup for *syr2k* against the slowest combination. For *gemm*, it is noticeable that best performance were obtained when CPU was evicted. On the opposite, combinations benefit from the use of CPU and GPUs for *syr2k*.

Imbalance shown in Fig. 8 and 10 mainly results from fluctuations in CPU time predictions. Despite the good accuracy of predictions, as confirmed by Fig. 11, slight changes in the partition size can significantly alter the predicted execution time and mislead the scheduler. This behavior is highlighted in *gemm (c7)*, for which the imbalance reaches 22%. However, imbalance is acceptable as it averages out at 5% and 8% for all the combinations of *gemm* and *syr2k*. Accuracy of our execution time prediction methods for *syr2k* is shown in Fig. 11. The plotted prediction errors are derived from the average error for all the PUs combination. Those results validate our methods for accurately predicting execution times.

Overall, the experiments show that our multiversioning

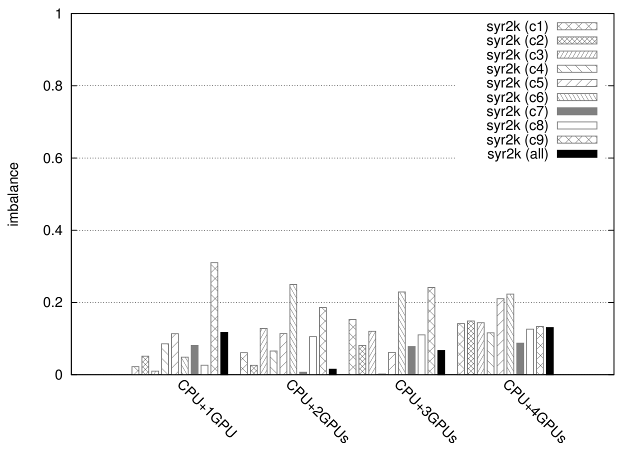

Figure 10: Execution time imbalance ratio for several combination of code versions for *syr2k*.

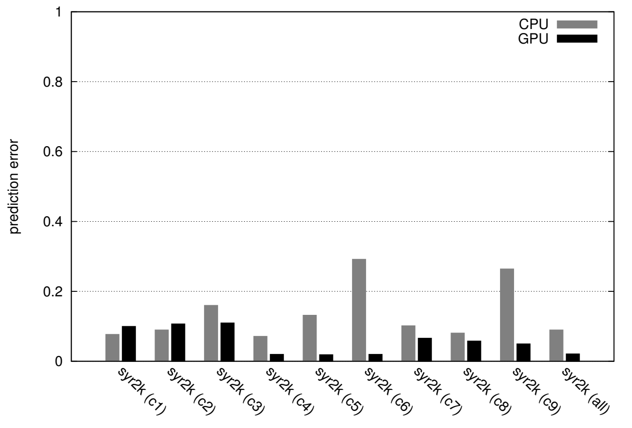

Figure 11: Average prediction error ratio of CPU and GPU for *syr2k*.

system was systematically selecting the best version, thus improving performance. The design and low overhead of our runtime system allows the comparison of multiple schedules, combining different code versions, during execution.

7. CONCLUSION

We presented an original method for achieving load balance between CPUs and GPUs in a dynamic context. It is based on an accurate prediction of the CPU and GPU execution times of codes, using the results of a profiling of those codes. We implemented it using Python scripts, calling several polyhedral compilation tools, and we tested it on the polyhedral benchmark suite, showing that it is effective on a platform composed of one CPU and 4 GPUs.

Our future plans include extending this work to handle other types of hardware, for example Xeon Phi processors and larger systems including 10's of GPUs and 100's of cores. Finally, the current system is focused towards performance, but with slight modifications it could be adapted to improve energy consumption by deactivating the inefficient PUs.

8. REFERENCES

[1] C. Augonnet, S. Thibault, and R. Namyst. Automatic calibration of performance models on heterogeneous multicore architectures. In *3rd Workshop on Highly Parallel Processing on a Chip (HPPC 2009)*, Delft, Netherlands, Aug. 2009.

[2] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In *EuroPar 2009*, LNCS, Delft, Netherlands, 2009.

[3] M. E. Belviranli, L. N. Bhuyan, and R. Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Trans. Archit. Code Optim.*, 9(4):57:1–57:20, Jan. 2013.

[4] E. Bendersky. pycparser. https://github.com/eliben/pycparser, 2010.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI '08*, pages 101–113. ACM, 2008. http://pluto-compiler.sourceforge.net.

[6] M. Boyer, K. Skadron, S. Che, and N. Jayasena. Load balancing in a changing world: Dealing with heterogeneity and performance variability. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 21:1–21:10, New York, NY, USA, 2013. ACM.

[7] P. Di and J. Xue. Model-driven tile size selection for doacross loops on gpus. In E. Jeannot, R. Namyst, and J. Roman, editors, *Euro-Par 2011 Parallel Processing*, volume 6853 of *Lecture Notes in Computer Science*, pages 401–412. Springer Berlin Heidelberg, 2011.

[8] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A hybrid multi-core parallel programming environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.

[9] J.-F. Dollinger and V. Loechner. Adaptive runtime selection for GPU. In *42nd International Conference on Parallel Processing - ICPP*, Lyon, France, 2013. IEEE.

[10] A. Kloeckner. islpy. https://pypi.python.org/pypi/islpy, 2011.

[11] T. Komoda, S. Miwa, H. Nakamura, and N. Maruyama. Integrating multi-GPU execution in an OpenACC compiler. In *42nd International Conference on Parallel Processing - ICPP*, Lyon, France, 2013. IEEE.

[12] J. Lee, M. Samadi, Y. Park, and S. Mahlke. Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 245–256, Piscataway, NJ, USA, 2013. IEEE Press.

[13] NVIDIA Corporation. cuBLAS-XT. https://developer.nvidia.com/cublasxt, 2014.

[14] OpenACC corporation. OpenACC. http://www.openacc-standard.org/, 2012.

[15] J. Planas, R. Badia, E. Ayguade, and J. Labarta. Self-adaptive OmpSs tasks in heterogeneous environments. In *IEEE 27th Int. Symposium on Parallel Distributed Processing (IPDPS)*, pages 138–149, 2013.

[16] L.-N. Pouchet. PolyBench 3.1. http://www.cse.ohio-state.edu/~pouchet/software/polybench/, 2011.

[17] B. Pradelle, P. Clauss, and V. Loechner. Adaptive Runtime Selection of Parallel Schedules in the Polytope Model. In *19th High Performance Computing Symposium - HPC 2011*, Boston, USA, Apr. 2011. ACM/SIGSIM.

[18] University of Sydney. Magma. http://magma.maths.usyd.edu.au/, 1993.

[19] S. Verdoolaege. isl: An integer set library for the polyhedral model. In K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, editors, *ICMS*, volume 6327 of *Lecture Notes in Computer Science*, pages 299–302. Springer, 2010.

[20] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, Jan. 2013.

[21] S. Verdoolaege and T. Grosser. Polyhedral extraction tool. In *International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*, Jan. 2012.

[22] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 48(1):37–66, 2007.