Optimization Space Pruning without Regrets

Ulysse Beaugnon, Antoine Pouille, Marc Pouzet, Jacques Pienaar, Albert Cohen

Optimization Space Pruning without Regrets

Ulysse Beaugnon
Antoine Pouille   Marc Pouzet
ENS, France
name.surname@ens.fr

Jacques Pienaar
Google, USA
jpienaar@google.com

Albert Cohen
INRIA, France
albert.cohen@inria.fr

Abstract

Many computationally-intensive algorithms benefit from the wide parallelism offered by Graphical Processing Units (GPUs). However, the search for a close-to-optimal implementation remains extremely tedious due to the specialization and complexity of GPU architectures.

We present a novel approach to automatically discover the best performing code from a given set of possible implementations. It involves a branch and bound algorithm with two distinctive features: (1) an analytic performance model of a *lower bound* on the execution time, and (2) the ability to estimate such bounds on a *partially-specified* implementation.

The unique features of this performance model allow to aggressively prune the optimization space without eliminating the best performing implementation. While the space considered in this paper focuses on GPUs, the approach is generic enough to be applied to other architectures.

We implemented our algorithm in a tool called *Telamon* and demonstrate its effectiveness on a huge, architecture-specific and input-sensitive optimization space. The information provided by the performance model also helps to identify ways to enrich the search space to consider better candidates, or to highlight architectural bottlenecks.

Categories and Subject Descriptors   D.3.4 [*Processsors*]: Optimization, Compilers

Keywords   GPU, search space exploration, performance model, branch and bound

1. Introduction

Graphic Processing Units (GPUs) offer massive parallelism whose raw processing power is an order of magnitude above contemporary multicores. Important domains like linear algebra can greatly benefit from such parallelism. For example, recent achievements in deep learning have largely relied on GPU acceleration (cuDNN).

However, choosing between the different optimizations and parallelization schemes available for a given algorithm is a difficult question. It requires a deep knowledge of the architecture and many trials and errors for the programmer to explore implementation alternatives at each level of parallelism, from thread-local optimizations to the mapping of computations across the entire processor.

Additionally, programmers have to carefully and explicitly specify memory transfers between the different memory spaces of various sizes and properties: global memory, different levels of cache, and scratchpad memories. Moreover, each targeted architecture and each problem may impose different implementation decisions to be taken. Consequently, programmers have to rely on proprietary libraries provided by hardware vendors to achieve competitive performance.

The problem is not limited to GPUs. On any complex architecture, the optimization space grows exponentially with the number of possible decisions: exhaustive enumeration is not an option. Multiple domain-specific approaches have been proposed, based on auto-tuning, such as SPIRAL (Puschel et al. 2005) for DSP codes on multicores, or MAGMA (Tomov et al. 2010) for linear algebra on GPUs. Auto-tuning drives the search through statistical techniques. However, the selected implementation may still be significantly less than optimal. This is one of the reasons vendor libraries outperform auto-tuning solutions. As a result, the production and the specialization to a given target or problem-size of highly tuned libraries remain a critical challenge and a costly effort for most hardware vendors.

This paper introduces a novel approach to efficiently discover the fastest code in a set of possible implementations. The essential novelty is to guarantee that we will find the best-performing implementation while only evaluating a small fraction of the given search space. We achieve this using a hybrid analytical and empirical strategy, relying on a branch and bound pruning algorithm that combines an analytic performance model with actual runs of the application. A critical idea is that the performance model may provide valuable information even when many implementation decisions are left open. This is made possible through a performance model that provides a lower bound on the execution time of the codes that may be generated from a partially specified implementation. Formally, if X is a region of the search space, $x \in X$ are the codes that can be generated by instantiating its remaining open decisions, $B(X)$ is the lower bound provided by the model and $T(x)$ is the execution time for $x \in X$ on the GPU, then:

$$\forall x \in X : \; B(X) \leq T(x) \qquad (1)$$

This bound allows to prune the search space without ever missing the best implementation. It also provides information on the limits of the optimizations expressed in the search space. It can even explain the nature of performance anomalies and their localization in the program. This information allows the domain expert to extend the search space with new optimizations, and to select a suitable architecture through design-space exploration.

Our approach is implemented in a tool named *Telamon*. The optimizations that can be expressed with Telamon make it well suited for working with basic linear algebra kernels on GPUs. That said, our approach is generic, and provided with a suitable

search space and performance model, it could be expanded to other architectures and application domains.

The paper is organized as follows. We specify the scope of Telamon in Section 2 and provide some background on GPU architectures in Section 3. Section 4 presents the search spaces and their representations. Our exploration algorithm is described in Section 5 and the performance model in Section 6. We validate the pruning efficiency of Telamon and its ability to generate competitive code in Section 7. Finally, Section 8 discusses the differences between our approach and existing solutions.

2. Problem Statement

GPU programs are composed of small functions, called kernels, that are called from the CPU and run on the GPU. The objective of Telamon is to find the best performing implementation of a GPU kernel in a given search space.

The search space describes the implementation and optimization decisions that may be taken to generate code, such as how code is parallelized, how loops are tiled, unrolled and vectorized or which instructions are used. The performance of the generated code is limited by the decisions exposed in the search space. We do not address the problem of finding the optimizations that may be applied which is a research subject in its own.

Telamon optimizes for a specific user-provided input data. However, the generated code will run correctly on all inputs. Given a search space, Telamon returns the candidate implementation with the best execution time for the provided input. We currently target NVIDIA GPUs, but other GPU architectures are similar enough that porting Telamon to them should not be problematic.

The search space is described using the intermediate representation (IR) introduced in Section 4. The originality of this IR is to represent whole sets of observationally equivalent implementations of computational kernels, up to dependence-preserving loop transformations, instruction selection, data mapping and transfer optimization.

We consider programs with static control-flow: there are no `if`-statements and loop counts depend only on kernel parameters. This allows us to focus on a class of problems that are known to run well on GPUs (Owens et al. 2008). Many application domains such as basic linear algebra fall into this category.

The search spaces handled by Telamon are too big for all possible implementations to be evaluated on the GPU in a reasonable amount of time. Even the plain generation of the candidates, without running them, is too time consuming for non-trivial examples. We propose an approach to drastically reduce the number of candidate implementations that need to be generated and evaluated, while still guaranteeing to discover the fastest one.

3. CUDA Programming Model

We now provide some background on the CUDA programming model and architecture (Nickolls et al. 2008). CUDA is developed by NVIDIA for their own devices. The programming model used by other GPU vendors is similar (Stone et al. 2010).

The basis of CUDA is data parallelism: a single code is executed in parallel by a large number of threads. The code executed by each individual thread is usually implemented in the CUDA programming language. This CUDA code is then compiled into *PTX*, a low-level intermediate representation dedicated to NVIDIA GPUs, and finally to a proprietary assembly by the CUDA driver. Telamon directly generates PTX to improve its control over the generated code.

Threads are grouped into blocks, where they are organized into a 1, 2 or 3 dimensional grid. Similarly, blocks of threads are organized into another 1, 2 or 3 dimensional grid. Each block has the same thread configuration. The dimensions of these grids are hereafter called *thread dimensions* and *block dimensions*. Threads within the same block can synchronize with each other through barriers and communicate via a fast dedicated memory called shared memory. This is not possible among threads from different blocks.

There is one additional level of organization within a block: threads belong to *warps*, groups of 32 threads that are always executed simultaneously. The execution of warps on an SMX is interleaved in order to hide the latency of operations and to maximize throughput.

A GPU executes kernels by distributing blocks between several streaming multiprocessors (SMX). Each SMX contains a large number of small cores. Each core can execute at most one arithmetic instruction per cycle. The threads in a given block are dispatched to a single SMX, which can house a limited number of blocks based on the number of registers and the amount of shared memory used in each.

4. Search Space Representation

We represent the search space of possible implementation candidates of a given kernel using a specific IR. Its design is critical to efficiently find the optimal implementation of a GPU kernel. The central design feature of our IR is that a single instance can represent a whole set of candidate implementations. Indeed, an IR instance defines a region of the search space, i.e., as a partially specified implementation where some optimization choices are left free to be explored.

The search space can be fully specified as a single IR instance. In the following, we denote by X an IR instance and by $x \in X$ a candidate implementation represented by X. We first describe how unspecified implementation choices are represented and then list the optimization decisions available.

4.1 Representation of the Implementation Choices

The IR explicitly exposes the set of available alternatives for each unspecified implementation choice. For example, it lists the available cache directives for each memory access. Taking an implementation decision reduces to restricting an open choice to one of the listed alternatives. When each choice is specified, the IR instance represents a single implementation for which code can be generated. Thus, every candidate implementation $x \in X$ corresponds to a specific combination of decisions.

However, some combinations of choices are incompatible. To avoid the selection of mutually exclusive decisions, the IR guarantees that each available alternative of any choice is compatible with at least one alternative from every other choice. When a decision is taken, the IR recursively restricts the alternatives available for other choices accordingly.

Notice that lists of alternatives are only restricted and never increased, when making choices. All the optimizations that may be applied to the kernel are known in advance. This is critical for the performance model described in Section 6.

4.2 Available Decisions

An IR instance is composed of a list of instructions and loops. The loops, called *iteration dimensions* in our context, should have a fixed number of iterations that depend only on kernel parameters. Each instruction is nested into a specific list of iteration dimensions.

First, each instruction and each iteration dimension comes in multiple variants that must be chosen by the search space exploration:

- memory instructions can either use the L1 cache, the L2 cache, the Read-Only cache, or no cache at all;
- iteration dimensions can be implemented as plain for-loops, as vectorized or unrolled loops, or as thread or block dimensions.

Second, we list the possible ordering of each pair of iteration dimensions:

- one can be nested inside or outside the other;
- one can be executed before or after the other;
- the two dimensions may be fused.

This pairwise order can express many transformations, including loop fusion, loop interchange and coarse grain scheduling. In particular, we can express synchronization among the threads in a block: if a thread dimension is ordered before another, we introduce a barrier to ensure that all the computations of the first one complete before the second one begins.

The orders are encoded in a transitively closed adjacency matrix and are thus highly redundant. This allows us to represent complex non-scalar decisions and to easily access and restrict the list of available orders between any two dimensions. The redundancy does not impact the number of candidate implementations to consider, since the IR ensures that only distinct, compatible decisions are considered.

Finally, the IR can express point-to-point communications between two iteration dimensions: data produced at iteration i of one dimension may be consumed at iteration i of the second. This data transfer can be implemented in a register only if the two loops are fused or both unrolled. Otherwise an array is allocated in the GPU memory to buffer the data between the two loops. This array can be either in global or shared memory. Thus we also list how each data transfer can be implemented. In particular, this allows Telamon to generate copies to scratchpad memories if necessary.

In its current form, the IR does not capture stripmining decisions. To implement loop tiling (a.k.a. blocking), one needs to generate a separate IR instance for each available stripmining scenario. Practically, to capture the different stripmining and tiling combinations of a given kernel in the search space, Telamon is fed with multiple IR instances instead of a single one. For example, tiling a pair of nested loops involves stripmining both of them in the original IR instance, effectively exposing 4 nested iteration dimensions, and instantiating the permutation of the two intermediate dimensions further down the exploration of the search space. More generally, tiling opportunities are exposed in the search space by replacing each loop of size S tiled with a factor of T by two loops: one of size n/T and the other of size T. This allows both the number of tiling levels and the tile sizes of each dimension to be explored. As explained in Section 5, describing the search space with multiple instances impacts the time necessary to generate candidate implementations but not on the number of candidate implementation to evaluate on the GPU. The performance loss was minimal with the kernels we consider. This limitation is a temporary engineering decision to simplify the implementation, keeping the matrix size constant.

4.3 Sample IR Instance

Let us now present an IR instance associated with the (full) search space associated with the kernel $X \leftarrow \alpha.X$, where X is a vector of size N and α a scalar value.

Algorithm 1 shows the structure of the candidate implementations we consider. Each iteration dimension and instruction is labeled. The loop iterating over the array is stripmined twice so it can be mapped to the different levels of parallelism on the GPU. T_1 and T_2 denote the tile sizes. The inner stripmined loop encloses three innermost loops, one for each nested instruction: d_2, d_3 and d_4. This is necessary to vectorize memory accesses even if other instructions cannot be vectorized on the targeted GPU.

The instructions i_0 and i_1 compute a pointer to the first element of X to process at every iteration of d_0 and d_1. Memory accesses are indexed by closed-form induction variable expressions *index × constant + variable*. These expressions are derived from iteration

---

Algorithm 1: Structure of the kernel to optimize

Data: X a pointer to an array, α a scalar and N the size of X

Result: $\alpha.X$ is stored in X

d_0: **for** i **in** $0..N/(T_1 T_2)$ **do**
    i_0: $X_0 := $ move $4 T_1 T_2.i + X$;
    d_1: **for** j **in** $0..T_1$ **do**
        i_1: $X_1 := $ move $4 T_2 j + X_0$;
        d_2: **for** k **in** $0..T_2$ **do**
            i_2: $a := $ load $(4k + X_1)$;
        d_3: **for** k **in** $0..T_2$ **do**
            i_3: $b := $ mul $a[d_2 \rightarrow d_3], \alpha$;
        d_4: **for** k **in** $0..T_2$ **do**
            i_4: store $b[d_3 \rightarrow d_4], (4k + X_1)$;

---

dimensions and hoisted at the appropriate level accordingly. Then, i_2 loads a single value from X, i_3 multiplies it by α and i_4 stores it back into X. Both i_3 and i_4 feature point-to-point communication, respectively between the loop d_2 and d_3 (denoted as $a[d_2 \rightarrow d3]$) and the loops d_3 and d_4 (denoted as $b[d_3 \rightarrow d_4]$).

Along with the structure of the code, the IR stores the available choices for each decision. Table 1 shows how each iteration dimension can be implemented: as plain for-loops, as vectorized or unrolled loops or as block or thread dimensions.

Dimension	Implementation
d_0	Block, For-Loop
d_1	Block, Thread, For-Loop, Unrolled
d_2	Block, Thread, For-Loop, Unrolled, Vectorized
d_3	Block, Thread, For-Loop, Unrolled
d_4	Block, Thread, For-Loop, Unrolled, Vectorized

Table 1. Implementation choices for iteration dimensions

Similarly, Table 2 shows how memory instruction are implemented and Table 3 shows where the data is stored when it needs to be passed from one iteration dimension to the other.

Instruction	Cache Level
i_2	L2, RAM
i_4	L2, RAM

Table 2. Implementation choices for memory instructions

Data Transfer	Data Location
$a[d_2 \rightarrow d_3]$	Shared Mem, Global Mem, Register
$b[d_3 \rightarrow d_4]$	Shared Mem, Global Mem, Register

Table 3. Implementation choices for communications

Last, Table 4 lists the available ordering between each pair of instruction or iteration dimensions. The possible orderings are: Before (B), After (A), Nested Inside (I), Nested Outsize (O) and Fused (F). In practice, only half of the matrix is stored as the order between a and b can be deduced from the order between b and a.

5. Branch and Bound

The only way to know the exact execution time of a candidate implementation is to actually run it on the GPU with the input

	d_0	d_1	d_2	d_3	d_4	i_0	i_1	i_2	i_3	i_4
d_0	/	I, O	I, O	I, O	I, O	O	O	O	O	O
d_1	I, O	/	I, O	I, O	I, O	O, A	O	O	O	O
d_2	I, O	I, O	/	F, B	F, B	O, A	O		O, B	O, B
d_3	I, O	I, O	F, A	/	F, B	O, A	O, A	O, A	O	O, B
d_4	I, O	I, O	F, A	F, A	/	O, A	O, A	O, A	O, A	O
i_0	I	I, B	I, B	I, B	I, B	/	B	B	B	B
i_1	I	I	I, B	I, B	I, B	A	/	B	B	B
i_2	I	I	I	I, B	I, B	A	A	/	B	B
i_3	I	I	I, A	I	I, B	A	A	A	/	B
i_4	I	I	I, A	I, A	I	A	A	A	A	/

Table 4. Pair-wise ordering of instructions and dimensions

provided by the user. For non-trivial search spaces, however, it is impossible to evaluate every single candidate in a reasonable amount of time. Their number is exponential in the number of choices exposed in the IR, and even enumerating them without code generation is too expensive. In this section, we present a branch and bound algorithm that prunes the search space, without ever excluding the fastest candidate implementation.

Our pruning strategy is based on an analytic performance model which we apply to a given IR instance. It provides a lower bound on the execution time of every candidate implementation that is represented by the IR instance, even when some implementation choices were left unspecified. Formally, if X is an IR instance, $B(X)$ is the lower bound provided by the model and $T(x)$ is the execution time of $x \in X$ on the GPU, then:

$$\forall x \in X, \ B(X) \le T(x) \qquad (2)$$

The performance model is further detailed in Section 6.

The search space described by a single IR instance is explored by recursively building a search tree. The initial IR instance is the root of the tree: it represents the whole search space. The children of a node represent a partition of the search space described by the parent. They are obtained by picking an unspecified choice in the parent and creating one child for each of its possible values. At a given point, the structure of the tree specifies the choices that have been considered, as each node is a more constrained IR instance than its parent. By going deeper in the tree, we eventually reach its leaves: totally constrained IR instances that represent individual implementations. In fact, each node in the tree represents a search space that is exactly the set of the leaves beneath it.

When a leaf is encountered during the exploration, it is evaluated by actually running it on the GPU. We save the leaf with the shortest execution time as the current best candidate. Each time a new node is inserted in the search tree, it is evaluated with the performance model. We prune nodes for which the performance model returns a lower bound greater than the execution time of the current best candidate. With this pruning strategy, the best implementation candidate can never be pruned. Indeed, the performance model guarantees that there is not a single implementation in a pruned sub-tree that could run faster than the computed lower bound, for which we already know a better candidate.

When multiple nodes in the tree can be processed, we first look at the one with the lowest lower bound. This way we never consider a node that could have been later pruned. The exact algorithm is shown in Algorithm 2: $queue$ is a double-ended priority queue that orders the nodes with the lowest lower bound first.

As the performance model can highlight important performance bottlenecks even for a partially-specified IR instance, we are able to prune early in the search tree. Indeed, there is no need to generate the candidates in a sub-tree when the performance model detects its root may be pruned. Interestingly, our algorithm does not require the performance model to take all performance bottlenecks into account or to be too precise: the evaluation of candidates on

---

Algorithm 2: Search space exploration

Data: The initial search space s_0
Result: The best candidate implementation in s_0
$queue := [(s_0, 0)]$;
$best\_time := +\infty$;
$best_candidate := $ None;
while $queue$ is not empty **do**
  $node := queue.\text{pop\_first}()$;
  if $node$ is totally constrained **then**
    $time := $ evaluate $node$ on the GPU;
    if $time < best\_time$ **then**
      $best\_time := time$;
      $best\_candidate := node$;
      while $queue.\text{last.bound} > time$ **do**
        $queue.\text{pop\_last}()$;
  else
    $c := $ Pick an unspecified choice in $node$;
    foreach decision d available for the choice c **do**
      $new_node := $ apply d to $node$;
      $bound := \text{performance\_model}(new\_node)$;
      if $bound < best\_time$ **then**
        $queue.\text{insert}(new\_node, bound)$;

return $best\_candidate$;

---

the GPU ensures we always return the best performing implementation. The performance model only needs to be accurate enough for the search to complete in a reasonable amount of time.

When the search cannot be expressed as a single IR instance, because some implementation choices cannot be expressed, the user can provide multiple IR instances. In that case, the algorithm behaves as if the initial IR instances were all the children of some imaginary root node. This has a minimal performance impact provided the number of initial instances is not too big. Indeed, compared to a search tree with a single initial IR instance, the only overhead comes from the fact that some IR initial instances would not have not been generated if they could have been cut at a higher level in the tree. With multiple IR instances, the algorithm needs to evaluate each one of them once with the performance model to know which to keep. In our experiments, $250,000$ initial IR instances increased the search time by only one second (see Section 7.3).

The order in which choices are made is driven by a simple heuristic. We try to prioritize the decisions with the biggest performance impact. We first decide how iteration dimensions are implemented, then the memory space in which temporary arrays are allocated, then the ordering of dimensions and lastly the cache directives to use. More advanced heuristics could attempt to maximize the number of branches cut near the top of the search tree.

6. Performance Model

We designed a performance model that evaluates a lower bound on the execution time of all the candidates represented by a given IR instance. This model needs not provide an exact lower bound, but its precision influences the pruning efficiency. This is particularly important for fully constrained (instantiated) IR instances because evaluating a candidate on the GPU is more costly than generating the children of a node in the search tree.

The IR structure is strongly tied to the requirements of the performance model. Indeed, to evaluate a lower bound on the execution time of a kernel, the performance model should abstract over all possible optimizations. This is why our IR requires that available choices be listed explicitly.

Our performance model computes a lower bound on the pressure on execution and memory units at each level of parallelism as well as the latency of dependency chains to calculate a lower bound on the total execution time of partially specified kernels. The lower bound is correct even if not all performance bottlenecks are modeled. Additional bottlenecks can only make the code run more slowly.

The model first computes a lower bound on the average latency and number of execution and memory units used by each instruction. This information is used subsequently to compute the minimal execution time of a thread, then of a block, and finally of the whole kernel. These three execution times are respectively noted T_{thread}, T_{block} and T.

6.1 Individual Throughput and Latency Bounds

A GPU core has a limited number of resources of different kinds for executing instructions. For each instruction, we compute a lower bound on the number of resources it requires. We consider the number of Arithmetic and Logical Units (ALUs), the number of memory units, the number of instruction dispatchers and the number of Miss Status Holding Registers (MSHRs) that hold the status of pending memory requests.

If multiple alternatives are available for implementing an instruction, we compute the lower bound independently for each resource: we take the smallest usage of each kind of resource across all variants of the instruction.

The impact of memory instructions on GPU resources depends on how memory accesses are coalesced. A memory access is coalesced if multiple threads access the same cache line at the same time. In practice, coalescing depends on the mapping of iteration dimensions to GPU thread dimensions. We assume the most optimistic coalescing according to the available implementation choices of an IR instance. We do not yet have an accurate model of GPU caches and thus assume that all memory accesses hit the highest available cache. While this limitation impacts the number of candidates evaluated on the GPU, it does not alter the correctness of the lower bound.

Similarly, we obtain lower bounds on the average latencies of instructions and on the overhead in terms of resource use and instruction dependencies induced by iteration dimensions.

With this approach, we make independent optimistic assumptions for each instruction and for each iteration dimension. These assumptions may be mutually incompatible. For example, we might assume two iteration dimensions to be mapped to thread dimensions on the GPU, even if parallelizing both dimensions at the same time would create more threads than supported by the hardware and thus produce incorrect code. This alters the accuracy but not the correctness of the lower bound: relaxing the constraints on valid combination of choices can only lower the bound. When the IR instance is totally constrained, we make no assumption and the bounds should be exact within the limits of the model.

6.2 Thread Execution Time

We bound the execution time of a thread T_{thread} by multiplying the resources used by each instruction or iteration dimension, as computed in Section 6.1, by the minimal number of times it is executed within each thread. Then, we sum the resulting values across all the instructions to obtain a lower bound on the resources needed to execute the single thread. Finally, we divide this number by the resources that would be available for a thread at each cycle if it were run in isolation. This gives us a lower bound on the execution time of a thread based on the throughput of a single core.

$$T_{thread} \geq \frac{\text{Resources used per thread}}{\text{Resources available per thread}} \quad (3)$$

We may suppose that threads run in isolation as block-level contention is taken into account later in (6).

If the exact number of threads is left unspecified in the IR instance, we suppose maximal thread-level parallelism when computing the number of times instructions and iteration dimensions are executed. This assumption minimizes T_{thread}.

We also bound T_{thread} by the longest chain of data or control dependencies in a thread. The latency of data dependencies between instructions is computed as described in Section 6.1. Depending on how iteration dimensions are implemented, control dependencies may or may not be introduced. As always, we consider the most optimistic choice among those exposed in the IR.

$$T_{thread} \geq \text{Longest dependency chain} \quad (4)$$

6.3 Block Execution Time

The execution time of a block is always greater than the execution time of any thread within it.

$$T_{block} \geq T_{thread} \quad (5)$$

All threads within a block can be executed in parallel. Then, if $T_{block} \neq T_{thread}$, there is contention on the resources shared between the threads. We handled it as we did for threads in (3): T_{block} is limited by the resources used by a block divided by the resources available for a block running in isolation. However, threads in blocks may only be executed in warps (see Section 3). When the number of threads in a block is not a multiple of the warp size, phantom instructions are issued to complete the warp. While having no effect on the result of the computation, they still occupy resources, which we take into account by a *waste ratio*, that we define as the minimal number of phantom instructions issued per instruction.

$$T_{block} \geq \frac{\text{Resource used per block} \times (1 + \text{Waste ratio})}{\text{Resources available per block}} \quad (6)$$

When computing (3), we suppose maximal thread-level parallelism. However, we cannot suppose block-level parallelism to be maximal, as it may affect thread-level parallelism and thereby produce an incorrect lower bound. To avoid this problem, we compute T_{block} in (5) and (6) for maximum thread-level parallelism, as if block-level parallelism were minimal. We then divide the result by the maximum number of blocks that could be created (assuming block-level parallelism is maximal) for each block minimizing block-level parallelism (thus resulting in a minimal number of blocks). This provides a bound on the execution time of a block as if block-level and thread-level parallelism were both maximized.

$$T'_{block} \geq \frac{\text{Minimum number of blocks} \times T_{block}}{\text{Maximum number of blocks}} \quad (7)$$

6.4 Global Execution Time

First, as for blocks in (6), the global execution time T is bounded by the throughput of the GPU.

$$T \geq \frac{\text{Resources used by the kernel} \times (1 + \text{Waste ratio})}{\text{Resources available in the GPU}} \quad (8)$$

Second, T is bounded by the number of blocks that can be executed in parallel. This number is limited by the maximum number of active threads and blocks that the hardware can support, and by the amount of shared memory used by each block. Indeed, the combined use of shared memory by active blocks has to fit in the GPU shared memory.

$$T \geq \left\lceil \frac{\text{Maximum number of blocks} \times T'_{block}}{\text{Maximum number of parallel blocks}} \right\rceil \quad (9)$$

6.5 Strengths and Limits of the Model

Correctness of the Model Our performance model does not account for all GPUs bottlenecks. In particular, it does not model cache misses, nor the number of registers used per thread, nor the fact that not all ALUs can execute every kind of arithmetic instruction. Actually, performance bottlenecks may be missing from our analysis simply because their existence is hidden in the unpublished micro-architectural details of a GPU. The strength of our approach is that the lower bound is still correct as additional bottlenecks can only increase the execution time. Bottlenecks missing from the performance model are accounted for by the branch and bound algorithm by running candidate implementations on the GPU.

Partially Specified Implementations A particular strength of our model is its ability to bound the execution time of partially specified implementations, i.e., of entire regions of the optimization space. This is achieved by taking locally optimistic decisions whenever multiple options are available. The assumptions taken at two different points in the performance model algorithm do not need to be compatible with each other. Indeed, considering invalid combinations of choices in the performance model boils down to extending the search space with invalid candidate implementations and thus may only reduce the lower bound.

This approach is a source of inaccuracy when two incompatible assumptions are taken to avoid two different bottlenecks. One of those will always be present in the generated code, but the model assumes that both are avoided. However, this inaccuracy will only last while the two conflicting choices are left unspecified. The more choices are specified in the IR, the less conflicting assumptions are made and the more accurate is the model. For totally constrained instances, no assumption is made at all and the accuracy is maximal.

Interpretation of the Bounds The bound given by the performance model can be easily interpreted by looking at which equations were used to produce it:

- if T is bounded by (8), the bottleneck is the contention on a GPU resource shared between all the threads.
- if T is bounded by (9) and T_{block} by (6), the bottleneck is resource contention within a block.
- if T is bounded by (9), T_{block} by (5) and T_{thread} by (3), the bottleneck is the resource contention within a single thread.
- if T is bounded by (9), T_{block} by (5) and T_{thread} by (4), the bottleneck is a dependency chain within a thread.

Each of these bounds only considers a single bottleneck at once. This is a reasonable assumption in the context of GPUs, where a lot of threads run the same computation in parallel. Indeed, under such circumstances, parallelism hides the latency due to small bottlenecks. Only the most limiting factor cannot be hidden. This hypothesis does not hold for candidate implementations with too little parallelism, but such candidates usually perform so badly that they are anyway pruned away.

The accuracy of the performance model could be improved by bounding the execution time between each pair of synchronization primitives rather than looking at the latency of a whole thread when computing T_{thread} and T_{block}. This would handle the cases where block-level parallelism is too low. Similarly, we could better handle the cases where thread-level parallelism is too low by looking at the pressure on execution units between each pair of instructions to take local contention within a thread into account. Such improvements would, however, make the model more costly to run.

It is important to look at the throughput bottleneck at each level of parallelism as blocks and threads can only use a fraction of the resources available in a GPU. When we compute the bounds at the level of a thread or block, we assume it runs in isolation as contentions between threads or between blocks are taken into account by the bounds at a coarser level of parallelism.

7. Experiments

In this section, we present the performance of Telamon along three axes. In Section 7.2 we demonstrate that it can actually be used to automatically generate high-performance code for multiple GPU architectures. Then, in Section 7.3, we show the efficiency of our pruning strategy on a huge search space of possible implementations. times. Last, in Section 7.4, we evaluate the accuracy of the performance model used to compute lower bounds of execution

We choose to focus the experiments presented here on the matrix-matrix multiplication kernel SGEMM from the BLAS specification (bla 2002). This enables us to provide a more in-depth analysis of our results.

The SGEMM kernel computes the equation $C \leftarrow \alpha A.B + \beta C$ on single-precision floating-point values, where A, B and C are matrices and α and β are scalars. It is implemented by hardware vendors for all GPUs and is among the most-used functions in compute-intensive applications such as deep-learning. Section 7.1 describes the search space we used for our experiments.

7.1 Search Space

SGEMM has three inherent iteration dimensions: m, n and k. The first, m iterates over the rows of A and C, n iterates over the columns of B and C and k performs a reduction over the columns of A and rows of B.
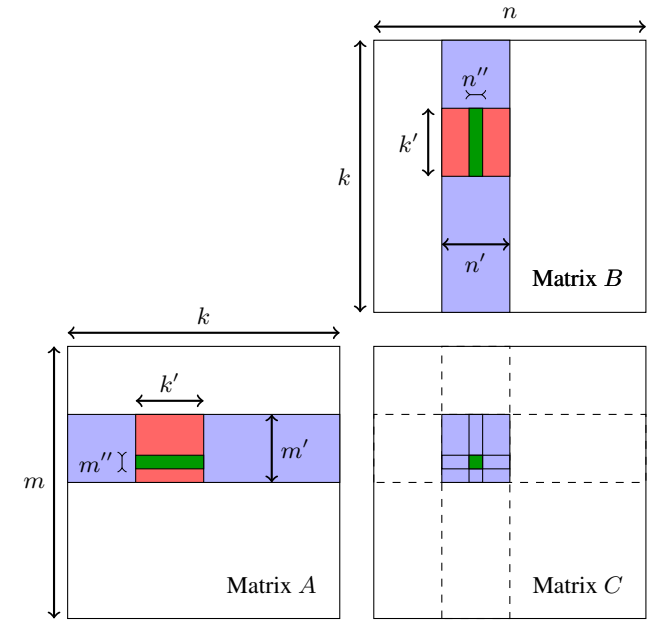

Figure 1. Tiling of iteration dimensions for SGEMM

Figure 1 shows the tiling of m, n and k. Our tiling strategy follows a standard procedure for parallelizing SGEMM on GPUs (Li et al. 2009). The dimensions m and n are tiled twice, creating four new dimensions iterating over the tiles of m and n: m', m'', n' and n''. The k dimension is only tiled once, creating a dimension k', as it offers less parallelism opportunities. We force m', n' and k' to have the same size to enable more loop fusion opportunities. Similarly m'' and n'' also have the same size.

We load all the data consumed by a tile $m' \times n' \times k'$ into two temporary arrays (one for A and one for B) to allow the reuse of the data of A and B, These chunks of data are shown in red in Figure 1.

Algorithm 3: Structure of our SGEMM implementation

> **for** i iterating over m, m', m'' **do**
>> **for** j iterating over n, n', n'' **do**
>>> $acc[i, j] \leftarrow 0$;
>
> **for** l iterating over k **do**
>> **for** i iterating over m **do**
>>> **for** j iterating over n **do**
>>>> Load a chunk from A into A_{tmp};
>>>> Load a chunk from B into B_{tmp};
>>>> Compute $A_{tmp} \times B_{tmp}$ into acc;
>
> **for** i iterating over m, m', m'' **do**
>> **for** j iterating over n, n', n'' **do**
>>> $C[i, j] \leftarrow \alpha.acc[i, j] + \beta.C[i, j]$;

Algorithm 3 presents the structure of computations in the search space. Our implementation of SGEMM is based on three main loop nests: the first one initializes an accumulator acc, the second one computes the matrix multiplication chunk-by-chunk and the last one adds the result to C. The loading of chunks and the $A_{tmp} \times B_{tmp}$ operation are each implemented by another loop nest. Each memory access also has its own loop nest on dimensions m'' and n'' to allow vectorization. Depending on how iteration dimensions are implemented, acc will be stored in registers or in a temporary array.

We expose the following implementation decisions in the search space:

- Each tiling level can take any size in $\{1, 2, 4, 8, 16, 32\}$. If the tile size is one, the tiling level is automatically removed.
- Dimensions in the same loop nest can be nested in any order.
- Dimensions in different loop nests can be fused together.
- We allow iteration dimensions to be implemented as plain loops, as fully unrolled or vectorized loops, or as thread or block dimensions. The real list of possibilities for each iteration dimension is actually smaller as the IR applies the following restrictions to ensure the correctness of the generated code: k and k' carry a reduction and thus cannot be parallelized, m and n have a non-constant size and thus cannot be fully unrolled and dimensions containing non-vectorizable instructions cannot be vectorized.
- A_{tmp} and B_{tmp} can be placed either in the shared memory or in any level of the cache hierarchy.

It is important to note that the performance of the generated code will be limited by the optimizations we expose in the search space. Our IR does not support texture memory and the programming language exposed by NVidia, PTX, does not allow expressing decisions such as register allocation. We believe, however, that the search space is big enough to achieve competitive performance and to show the validity of the approach. Tackling those limitations would require a significant implementation effort and is out of the scope of a research paper.

7.2 Generated Code Performance

We show that Telamon can be used to automatically generate competitive code for different architectures and different input sizes by finding the best implementation for two matrix sizes on three GPUs. The same search space, described in Section 7.1, is used for all the configurations.

The first GPU we use is a Quadro K4000, a desktop graphics card based on the Kepler micro-architecture, with 768 CUDA cores, a peak performance of 1.2 TFLOPS and 125 GB/s of memory bandwidth. The second GPU is a Tesla K20m, a passively

cooled GPU built to be used in data-centers and based on a variant of the Kepler micro-architecture optimized for non-graphic applications. The Tesla K20m features 2496 CUDA cores, a peak performance of 3.5 TFLOPS and a memory bandwidth of 208 GB/s. The last GPU is a GeForce GTX 470 based on the Fermi micro-architecture, with 448 cores, a peak performance of 1.1 TFLOPS and a memory bandwidth of 134 GB/s.

We compare the code generated using Telamon to a *Cublas*, native hand-written library, and to a state-of-the-art parallel code generator, *PPCG*. Cublas is the highly optimized implementation of BLAS that NVidia provides for its GPUs, and PPCG (Verdoolaege et al. 2013) is a compiler that can generate parallel code for GPUs from C-like source code. PPCG leverages polyhedral compilation strategies to express parallelism in the provided code, and relies on a combination of heuristics and exhaustive search to find efficient optimization parameters.

Table 5 shows the execution time for each configuration. We consider square matrices of size 256 and of size 1024. The timings do not account for the data transfers between the GPU and the CPU. Unfortunately, we could not run PPCG on the Tesla K20m as we did not have a direct access to this GPU. Each value is averaged over 10 runs. The standard deviation was below 0.03 in all cases.

On 1024×1024 matrices, we reach a performance within reasonable distance of Cublas. The slowdown indicates that our search space does not cover all the target-specific optimizations implemented in Cublas. Further investigation indicates that support for texture memory is the main missing dimension in the space. Texture memory provides a very fast read-only cache, enabling host-side loops to synchronize blocks. This limitation is not fundamental to our approach and is considered for future versions of the tool. We do not believe supporting texture memory will bring new insights to the performance model and search algorithm however.

On 256×256 matrices, we beat Cublas by a wide margin ($4.2\times$ speedup on average) thanks to Telamon's ability to adapt the generated code to the size of the input, while libraries can only provide a few implementations of each function.

We outperform PPCG on all configurations. This shows the strength of our approach over a combination of heuristics and exhaustive search to find the best optimization parameters. Indeed, apart from the cache directives, we do not introduce optimizations that could not be generated with PPCG.

Telamon performs similarly well on all platforms, thus proving our performance model is generic enough to support multiple micro-architectures and can adapt the generated code to the targeted hardware.

7.3 Search Space Pruning

We now evaluate the pruning efficiency of Telamon on the search space presented in Section 7.1. This search space contains 2.7 billion candidate implementations and the corresponding search tree 5.4 billion nodes. To obtain these numbers, we generated the full search tree on a dual-socket computer equipped with two Intel Xeon E5-2630v2 processors, each of them featuring 6 hyperthreaded cores running at 2.6 GHz. Enumerating the search tree without generating code or evaluating candidate implementations on the GPU took more than 5 hours using 24 threads.

Let us now focus the search space exploration on 1024×1024 matrices on the Quadro K4000 GPU. Other sizes and GPU architectures present similar results.

Exhaustive evaluation is clearly impracticable with the considered search space: it would take 149 days to run all the candidates on the GPU, assuming they are all as fast as the fastest implementation. The actual number would be much higher as our performance model shows the bad performance of most candidates.

| | Quadro K4000 | | Tesla K20m | | GeForce GTX 470 | |
	256×256	1024×1024	256×256	1024×1024	256×256	1024×1024
Cublas	0.410 ms	4.05 ms	0.373 ms	1.43 ms	0.494 ms	3.57 ms
Telamon	0.0940 ms	5.16 ms	0.0866 ms	1.61 ms	0.133 ms	4.92 ms
PPCG	0.500 ms	15.4 ms	-	-	0.219 ms	14.6 ms

Table 5. Performance of the code generated by Telamon

Finding the best implementation using Telamon takes only 13 minutes. This shows that Telamon can efficiently explore large search spaces. More precisely, only $17,664$ candidate implementations were compiled and run on the GPU and only $121,335$ nodes of the search tree were visited. This is an example of a reduction by a factor of 1.5×10^5 in the number of candidate implementations to evaluate, and a reduction by a factor of 4.8×10^4 in the number of nodes visited in the search tree compared to exhaustive search.

The reduction in the number of candidates to be run on the GPU is critical to achieve good exploration performance. However, the ability to prune high in the search tree also plays an important role. Indeed, the exploration takes only 13 minutes compared to the 5 hours necessary to enumerate the search space.
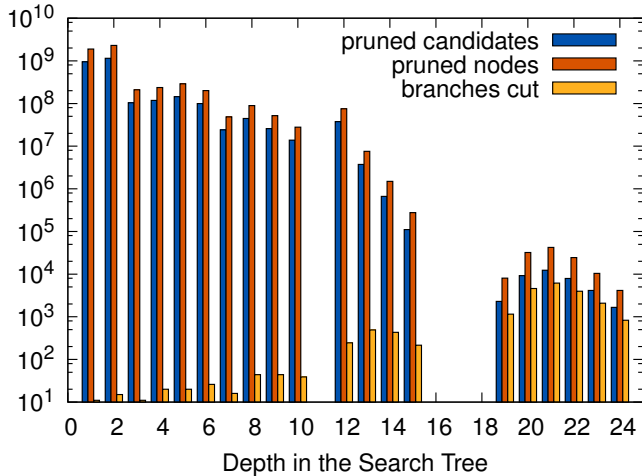
7.4 Performance Model Accuracy

To measure the accuracy of the performance model, we evaluate all of the candidate implementations in a search space on the GPU and compare their execution times with the lower bounds provided by the model.

The search space presented in Section 7.2 is far too big to be fully evaluated and must thus be restricted. We first fix the size of tiles: m', n' and k' are set to 16, and m'' and n'' to 4. Second, we force m' and n' to be mapped to the thread dimensions of the GPU. Last we choose to allocate the temporary arrays A_{tmp} and B_{tmp} in shared memory. These settings ensure that the search space is small enough to enumerate ($73,728$ candidates) and that candidate implementations run fast enough to be evaluated exhaustively.
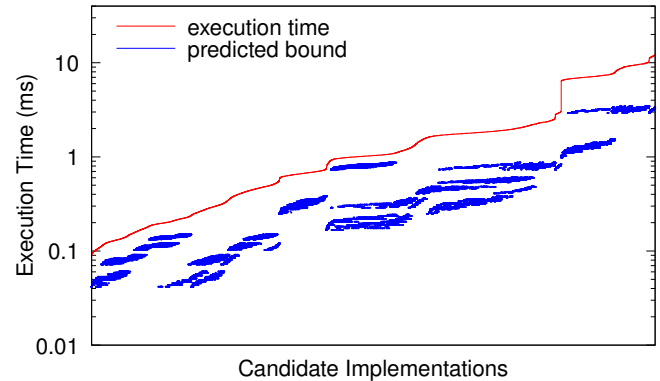

Figure 2. Nodes pruned at each level of the search tree

Figure 2 shows the effect of the pruning at each level of the search tree. It displays the number of candidate implementations and the number of nodes contained in the sub-trees that are cut at a given depth of the search tree. It also gives the number of cuts performed.

The main conclusion from this graph is that most candidates are pruned by a cut which is high in the tree: 77% of the nodes are cut in the two first levels. It should be noted that the y axis uses a logarithmic scale. Even the small step between levels 2 and 3 shows a decrease in the number of pruned candidates by a factor of 10, while 6×10^6 times fewer candidates are pruned in the last level than in the first. Thus, the performance model detects some major performance bottlenecks early in the exploration, and already prunes the search space with only a few cuts in the first levels. More cuts are necessary in the lower levels as bottlenecks become harder to detect, but their number is still reasonable. In the end, only 20448 cuts were needed to prune 2.7×10^9 candidate implementations.

The levels 11, 16, 17 and 18 do not show any cut as the performance model was unable to discriminate between the alternatives of the choices made at these levels.

Figure 3. Performance model accuracy

The execution time and the lower bound computed for each candidate are shown in Figure 3. The y-axis represents the actual and predicted execution time, and the x-axis the candidate implementations sorted by their execution time.

The lower bound is correct on all candidates: it is always below the measured execution time. Second, the bound follows the execution time and allows to prune a part of the search space: all candidates whose lower bound is higher than the best execution time are pruned. For the purpose of modeling the accuracy of the performance model, we have restricted the search space so that it only contains fast enough implementations. In practice, a much bigger fraction of the search space is pruned.

The bound is not tight. However, in our case, the ability to derive information from partially specified instances matters more than the tightness of the bound on fully specified instances. Figure 2 shows that our bound is efficient at pruning high in the search tree, where each cut has a huge impact and reduces both the number of nodes to visit and candidates to evaluate. A tighter bound would help reduce the number of candidate evaluated but would have a small impact high in the search tree, where the bound is loose anyway because of the missing implementation decisions.

8. Discussion and Related Work

To the best of our knowledge, Telamon is the first program representation, search algorithm and tool capable of constructing the best performing implementation out of a large search space, in presence of global transformations involving loops and arrays.

Super-optimization is a class of algorithms to find the optimal instruction sequence for a given computation. However, these algorithms are limited to straight-line code (or assimilated to it) (Phothilimthana et al. 2016). Our search space is structurally very different and too large even for plain enumeration.

Existing basic linear algebra libraries rely either on manually tuned code, such as Goto BLAS (Goto and Geijn 2008) on CPUs or Cublas on GPUs, or on autotuning, or both. Manual optimization requires a careful understanding of the performance anomalies of each implementation, and a lot of effort to produce these and to deal with architecture variants.

Autotuning involves statistical methods and predictors to drive the search towards candidates that are more likely to perform well (Agakov et al. 2006; Jia et al. 2013; Ansel et al. 2014). While it avoids the hurdles of manual optimization, autotuning has its own limitations. First it does does not provide any guarantees on the distance to the optimal code. The search algorithm might find a local optimum whose performance would still be far from the best implementation in the search space. Second, the structure of the search space must be carefully crafted for the autotuner to be efficient. Indeed, stochastic methods rely on a notion of locality between pairs of candidate implementations: Indeed, stochastic methods rely on the notion of locality: similar implementations are likely to have a similar execution time. As explained in (Ansel et al. 2014), this locality information is hard to preserve for complex search spaces. On the opposite, our branch-and-bound approach applies naturally to non-homogeneous spaces defined by globally interacting constraints. And indeed, our IR exposes complex interdependent decisions, that may not naively be mapped to the predefined constructs of autotuning frameworks. For example, the nesting order of loops cannot be represented by a simple permutation: it also depends on loop fusion and distribution choices, and on how the loops are mapped to the different levels of parallelism of the GPU. Furthermore, the fusibility of loops is not a transitive relation when fusion is enhanced with enabling transformations such as shifting or statement reordering (Pouchet et al. 2011).

Another approach, used in ATLAS (Whaley and Dongarra 1999) on CPUs and in MAGMA on GPUs (Tomov et al. 2010), is to combine manual optimizations and exhaustive search rather than statistical exploration. These libraries achieve remarkable performance, but their approach is kernel-specific and cannot be easily adapted to new problems. LGEN (Spampinato and Püschel 2014) proposes a more generic approach to compile basic linear algebra kernels, but still relies on a combination of heuristics and exhaustive search and thus cannot explore many alternatives. Yotov et al. pioneered the analytical modeling of the performance of BLAS kernels (Yotov et al. 2003), but scaling their approach to complex parallel architectures has remained an open problem.

SPIRAL (Puschel et al. 2005) generates high-performance code for DSP linear transforms from a high level representation. It explores a collection of rewrite rules for a few well-defined operators, which serve as the generators of a search space. The exploration of the space yields a Directed Acyclic Graph (DAG) which is similar to our search tree, but where confluent rewritings will lead to the same implementation. (De Mesmay et al. 2009) proposes to detect the most promising branches to be explored using a Monte-Carlo approach: a branch is evaluated by randomly selecting a few candidate implementations among its descendants and by evaluating them on the GPU. The authors show it is possible to prune high in the search tree—or DAG in this case—and still achieve excellent

performance; yet they cannot guarantee the best implementation will be found. (Remmelg et al. 2016) also show promising performances on matrix-matrix multiplication using rewrite rules. However, their approach is based on heuristics and a fixed parallelization scheme that offer no guarantees on the execution time and may not adapt to other kernels.

Compared to manual optimization, one collateral damage of autotuning is the understanding of performance anomalies, and the characterization of the remaining bottlenecks that the selected optimizations did not address. This information is crucial to understand why a program does not run faster and to understand if and how the considered search space can be extended to include better implementations. Telamon is able to explain its choices and the performance bottlenecks it faces, from the bounds produced by the performance model. For example, the performance model shows that the limiting factor in the code generated for the 1024×1024 SGEMM kernel on a Quadro K4000 is the number of blocks that can be run in parallel on a single SMX. This number could be increased by using the GPU's texture memory to load matrix B instead of copying it to a temporary array, thus reducing the amount of shared memory used per block. Alternatively one could use another GPU with a larger shared memory. While performance counters found in most GPUs can provide accurate information on a specific implementation, we provide insights valid on large regions of the search space, independently of the optimizations that may subsequently be chosen. We illustrated this explanatory capability of Telamon on the construction of an SGEMM-specific optimization space, refining and extending the space incrementally. The same capability could be extended to design-space exploration, by including hardware exploration in the space.

Our performance model remains inexpensive and easily interpretable because it considers each bottleneck independently. It does not try to model the interaction between multiple resource constraints. The idea of looking independently at each bottleneck to bound the execution time is developed in the roofline model (Williams et al. 2009) and known to work well on highly parallel architectures and deep memory hierarchies, where parallelism or a dominating factor hides secondary bottlenecks. In (Lai and Seznec 2013), Lai et al. show the effectiveness of this approach: they provide a bound at 77% percent of the actual execution time of the matrix multiplication kernel just by analyzing the different throughput constraints of a highly optimized code.

The originality of Telamon is to bound the execution time of partially specified implementations. It allows the branch and bound algorithm to prune high in the search tree without missing the best implementation. This ability plays a critical role on the reduction of the exploration time.

Other analytical performance models, such as (Hong and Kim 2009) or (Baghsorkhi et al. 2010), have been developed to drive the search for good optimization parameters on GPUs. These models might be more accurate than the one presented here. However, they tackle a different problem as they try to estimate the execution time rather than providing a lower bound on it. Code generators based on such models, such as (Samadi et al. 2012), are highly dependent on the accuracy of the model they use. Instead, Telamon does not require the performance model to be too precise, as we guarantee to find the best implementation even if some bottlenecks are not accounted for. This is crucial as the exact micro-architecture of GPUs is not disclosed by hardware vendors.

Existing performance models require all optimization decisions to be specified and thus cannot prune early in the search tree. Our intermediate representation is the key ingredient to operate on a partially specified implementation. A single instance of the IR captures whole regions of the search space, abstracting optimistically all the transformations that may be applied to the considered kernel

when computing the performance lower bound. In contrast, SPI-RAL (Puschel et al. 2005) relies on a set on rewrite rules for a few well-defined operators. With the rewrite rule approach, one must first apply a rule to find which further optimizations may be applied. An optimization which depends on specific rewrite rules may be completely missed by a non-exhaustive search. With the IR developed for Telamon, available optimizations are known upfront and can be applied in any order.

9. Conclusion

We present a performance model, an associated exploration strategy, and a tool called Telamon to find the best performing implementation out of a large program optimization space. The model is able to predict the expected performance of partially specified implementations. This crucially allows us to terminate the exploration of sub-optimal parameter space branches at an early stage, and makes the search for the optimal implementation tractable. Telamon demonstrates that reliable information about the execution time of a whole region of the optimization space can be provided before the implementations in this region are fully specified. Furthermore, as the model provides a lower bound on the execution time, the best candidate implementation is guaranteed not to be pruned. We validated our approach on the automatic generation of a high-performance GPU kernel.

Interestingly, we discovered that evaluating performance early in the exploration was critical to enable agressive pruning strategies. High accuracy comes next and is not strictly necessary to find the best implementation. For example, Telamon can explore large search spaces even if the current model assumes all memory accesses are cached at the highest level of the hierarchy. Still, we are currently designing an optimistic model of GPU caches to further reduce the number of candidates evaluated.

We also discovered that the performance model provides key insights to the designer of a code generator and autotuner. It can pinpoint which performance bottlenecks should be tackled to achieve better performance, hinting at missing dimensions in the optimization space. This point has been of tremendous help when building the search space for the GEMM kernel and refining it to compete with vendor libraries. We believe it can also help supporting design-space exploration, to select the most appropriate platform for a given computational domain.

Telamon is dedicated to basic linear algebra on GPUs and currently evaluated on GEMM only. Yet our branch and bound approach is generic and could be applied to other domains and hardware. The intermediate representation would need to be extended to capture that domain's computational structure and to adapt the performance model to consider the bottlenecks of the target platform. When porting our approach to another computational domain, the main hurdle is related with the design and implementation of the intermediate representation. The programmer has to consider every interaction between the different optimizations, how they compose and whether they contradict or restrict each other. One interesting possibility would be to generate domain-specific intermediate representations and the associated exploration scheme from a high level description of the optimization requirements.

References

An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Softw.*, 28(2):135–151, June 2002. ISSN 0098-3500.

F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization*, CGO. IEEE Computer Society, 2006.

J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. Opentuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*, 2014.

S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. Hwu. An adaptive performance modeling tool for GPU architectures. In *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, volume 45, pages 105–114. ACM, Jan. 2010.

cuDNN. NVIDIA cuDNN GPU accelerated deep learning. `https://developer.nvidia.com/cudnn`.

F. De Mesmay, A. Rimmel, Y. Voronenko, and M. Püschel. Bandit-based optimization on graphs with application to library performance tuning. In *Annual International Conference on Machine Learning*, pages 729–736. ACM, 2009.

K. Goto and R. A. Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):12, 2008.

S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 152–163. ACM, 2009.

W. Jia, K. A. Shaw, and M. Martonosi. Starchart: hardware and software optimization using recursive partitioning regression trees. In *Conference on Parallel architectures and compilation techniques*, pages 257–268. IEEE Press, 2013.

J. Lai and A. Seznec. Performance upper bound analysis and optimization of SGEMM on Fermi and Kepler GPUs. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–10. IEEE, 2013.

Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning GEMM for GPUs. In *International Conference on Computational Science, ICCS'09*. Springer, May 25-27 2009.

J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, Mar. 2008. ISSN 1542-7730.

J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati. Scaling up superoptimization. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS. ACM, 2016.

L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. Loop transformations: Convexity, pruning and optimization. In *38th ACM Symposium on Principles of Programming Languages (POPL)*, Austin, Texas, Jan. 2011.

M. Puschel, J. M. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, et al. Spiral: Code generation for DSP transforms. *IEEE*, 93(2), 2005.

T. Remmelg, T. Lutz, M. Steuwer, and C. Dubach. Performance portable GPU code generation for matrix multiplication. In *Workshop on General Purpose Processing using Graphics Processing Units (GPGPU)*. ACM, 2016.

M. Samadi, A. Hormati, M. Mehrara, J. Lee, and S. Mahlke. Adaptive input-aware compilation for graphics engines. *ACM SIGPLAN Notices*, 47(6):13–22, 2012.

D. G. Spampinato and M. Püschel. A basic linear algebra compiler. In *Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 23. ACM, 2014.

J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36 (5-6):232–240, June 2010.

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), Jan. 2013.

R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 2009.

K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2003.