Search Space Generation and Pruning System for Autotuners

Piotr Luszczek* Mark Gates* Jakub Kurzak* Anthony Danalis* Jack Dongarra*†‡
*University of Tennessee, USA
†Oak Ridge National Laboratory, USA
‡University of Manchester, UK

***Abstract*—This work tackles two simultaneous challenges faced by autotuners: the ease of describing a complex, multidimensional search space, and the speed of evaluating that space, while applying a multitude of pruning constraints. This article presents a declarative notation for describing a search space and a translation system for conversion to a standard C code for fast and multithreaded, as necessary, evaluation. The notation is Python-based and thus simple in syntax and easy to assimilate by the user interested in tuning rather than learning a new programming language. A large number of dimensions and a large number of pruning constraints may be expressed with little effort. The system is discussed in the context of autotuning the canonical matrix multiplication kernel for NVIDIA GPUs, where the search space has 15 dimensions and involves application of 10 complex pruning constrains. The speed of evaluation is compared against generators created using imperative programming style in various scripting and compiled languages.**

I. Introduction

The BEAST project follows the classic recipe for automated software tuning. First, a computational kernel is implemented and parameterized with a set of tunable parameters (tiling sizes, implementation options, hardware switches), which define the search space. Then pruning constraints are applied to trim the search space to a manageable size. Then the variants that pass the pruning process are compiled, run and benchmarked, and the best performers are identified.

In the course of working on the matrix multiplication kernel, known in the HPC community by its GEMM name in the BLAS library, we discovered that the search space generation and pruning process poses some serious challenges in terms of ease of use and speed of evaluation. This led us to the development of a *declarative*, Python-based language for describing the search space, with pruning constraints, and a translation system that converts that description to a standard C code, which can then be compiled with a C compiler, executed at high speed, and multithreaded for extra performance.

We decided to present the language in the context of the GEMM kernel, which has the largest and most complex search space, and the largest and most complex set of pruning constraints, that we have ever encountered in the course of our work on many different GPU kernels. Also, we discuss the work in the context of NVIDIA CUDA, which has been the main vehicle of our implementations so far, and also, for simplicity, we focus specifically on the Kepler architecture, which for many months has been the accelerator of note for High Performance Computing. We assume that the reader has some familiarity with CUDA and GPU architecture, as it is difficult to include these basics due to space limitations.

II. Motivation

The ultimate goal of the BEAST project is to explore the search space without introducing any arbitrary constraints, but only those that have sound technical justification. To start with, we want to have a very large search space, to ensure that best performing kernels are not missed. We also want to apply aggressive pruning, to explore that space in the shortest time, interactively if possible. Thus, we want to make sure that pruning eliminates only those kernels that have absolutely no chance of achieving good performance.

Performance engineers commonly apply arbitrary constraints to the problem dimensions when tuning GPU kernels, for example, using data sizes and index strides that are a power of two, and setting upper limits of loops to powers of two. Power-of-two sizes could be considered reasonable for all kinds of parameters, because they correlate to hardware specs. What we strive to accomplish in the BEAST project is to eliminate this kind of educated guesswork based on the developer's intuition.

In our view, it is better to replace such arbitrary decisions with a set of derived constraints that have a direct relation to performance. One of the best examples here is the GPU occupancy, which is a function of multiple variables, including: the number of threads in a block, the number of registers required by each thread and the amount of shared memory required by each block. Occupancy threshold is a very effective and safe pruning constraint, as most kernels have no chance of achieving good performance at low occupancy levels. One can think about it as an automated occupancy calculator, which becomes an integral part of the pruning process, alongside other constraints.

III. Background and Contributions

In the past, we used the BEAST methodology to tune GEMM kernels for the NVIDIA Fermi architecture [1], [2], and achieved substantial performance improvement over cuBLAS for the double precision complex case (ZGEMM). We also tuned GEMM kernels for the NVIDIA GTX 680 consumer card, which was the first available card with the Kepler architecture [3], and achieved substantial performance improvement over cuBLAS for the single precision complex case (CGEMM). Recently, we used the BEAST approach to study energy consumption trade-offs of the GEMM kernel [4].

We also used the BEAST system to produce kernels other than standard BLAS. Recently, we implemented and tuned the fastest kernels for the batched Cholesky factorization and triangular solve for large sets of very small matrices [5] and achieved between 3× and 5× performance improvement over cuBLAS. Finally, we applied the methodology to a much more exotic kernel, the alternating least square algorithm for collaborative filtering [6] and achieved significant speedups over CPU implementations of the same operation.

A related development is our work on visualization of the search space pruning process. We developed a radial, space-filling technique that allows the user to gain a better understanding of how the pruning constraints remove candidates from the search space [7].

Our major *contributions* include: (1) autotuning toolchain for specifying, building and testing user-defined kernels for accelerators; (2) use of familiar Python syntax for search space specification rather than dedicated Domain Specific Language (DSL); (3) DAG-based pruning of the search space; and (4) performance analysis of various language backends for our code generator.

IV. Related Work

The list of prominent autotuning software projects includes packages such as Automatically Tuned Linear Algebra Software (ATLAS) [8], and its predecessor, Portable High Performance ANSI C (PHiPAC) [9] that targeted superscalar processors with dense linear algebra kernels. Sparse matrix computations were the main focus of Optimized Sparse Kernel Interface (OSKI) [10], while FFT and similar transforms were optimized by Fastest Fourier Transform in the West (FFTW) [11] and Spiral [12]. Spiral also recently addressed matrix-matrix multiply [13]. None of these projects address autotuning for accelerators, and they mostly embed the expert knowledge of tuning inside the code rather than expose it in the form of stencils as BEAST does. DSLs also exist for the sole purpose of autotuning parallel scientific codes [14], [15]. A much more complete survey of recent advances in autotuning is available elsewhere [16].

In our autotuning work and HPC code design, we follow two particular examples of successful open source solutions for very efficient matrix-matrix multiplication. One was done by Volkov et al. [17] and the other was done by Nath et al. [18], [19]. These efforts showed how it was possible to discover the unknown parameters of the GPU hardware and to autotune the kernels of interest accordingly. Sadly, the era of autotuning based on open source software and using openly available information has ended with the introduction of highly optimized codes inside NVIDIA's cuBLAS library that use assembly instructions and binary codes not available to a regular user [20, Section 5].

Among the science kernels that have been successfully standardized, only in the case of dense linear algebra have autotuning techniques been used to achieve reasonable performance on new architectures. The performance of the MAGMA library on GPU-accelerated systems is probably one of the very few examples of leveraging standards to accelerate legacy algorithms with moderate recoding/porting effort. But even in the area of dense linear algebra, the autotuned library offload model is breaking down. A number of cases can be identified that arise in the context of hybrid environments but are not envisioned by the existing standards and not supported by any libraries, e.g., new matrix layouts, such as the tile layout [21], higher precision than supported by hardware, such as the quadruple precision [22], [23], non-IEEE arithmetic, such as interval arithmetic [24, ch. 9], etc. Consequently, classic library autotuning approaches (e.g. ATLAS, FFTW) are not addressing hybrid architectures, and even if they were, the vast majority of science kernels would be out of their scope as being too specific to include in a general purpose software library.

```
r = range( N )
fibonacci = Iterator( [ 1, 1, 2, 3, 5, 8, 13 ] )
```

Fig. 1. Various forms of iterator definitions in the BEAST language.

```
@iterator
def inner():
    return range( outer )
@iterator
def outer():
    if archiecture == Fermi: return range( 32 )
    elif archiecture == Kepler: return range( 192 )
    elif archiecture == Maxwell: return range( 256 )
ex_outer = range( 100 )
ex_inner = range( ex_outer )
```

Fig. 2. Deferred iterators that show how dependent iterators are handled in the BEAST language and their expression-based counterparts.

A technique called light modular staging [25], [26] recently was used [27] to port Discrete FFT from the Spiral framework [12] to the Scala's LMS system [28]. Our approach is similar in principle in that we use code generation and embed autotuning DSL (Domain Specific Language) inside the Python code. Note that we have been doing this before in the context of the HPC Challenge benchmark [29].

V. Iterators

The following parameter iterators exist in the BEAST language:

- Expression iterators
- Deferred iterators
- Closure iterators (generator-based)

```
@iterator
def primes():
    yield 1
    yield 2
    n = 3
    old_primes = list()
    while n <= MAX:
        for i in old_primes:
            if n % i == 0:
                break
        else:
            yield n
            old_primes.append(n)
        n += 2
```

Fig. 3. Closure iterator that generates prime numbers smaller than or equal to MAX with MAX>= 3.

Expression iterators are defined through Python expressions, most notably the range() builtin as shown in Figure 1. The figure shows that the builtin function was overloaded and accepts not only integers but also other iterators, which is the basic tool for nesting of iterators and making them depend upon each other. We cover the intricacies of iterators' dependence analysis and use in Section X-A. Additional forms of syntax for expression-based definition of iterators are shown in Figure 1. The expression syntax extends beyond defining new iterators and also covers the use of iterator values in intermediary expressions, which is described in Section VIII. Finally, expressions involving iterators are the primary way of defining constraints as described in Section VI.

Deferred iterators may be considered an extension of *expression iterators* that allow the developer to use a much broader set of Python's constructs in order to achieve more advanced semantics. This primarily includes operators that cannot be overloaded in a generic way such as the boolean operators. Also permitted are the if-elif-else statements that cannot be achieved through Python's ternary operator. This extended syntax possibilities are shown in Figures 2. Another advantage of *deferred iterators* is that the order of definitions of iterators is relaxed. This avoids the requirement that *expression iterators* need to be defined in the order that puts the independent iterators first, the iterators that depend on those second, and so on. Figure 2 shows two deferred iterators: outer and inner. The former does not depend on any external variables and will become the outer loop in the generated code. The latter depends on the former and needs to become the inner loop in the generated code. However, the order of definitions of these iterators in the code can be arbitrary. This is not the case for the two expression iterators in the figure: ex_outer and ex_inner; they have to be defined in the order shown in the figure or otherwise will cause either NameError or UnboundLocalError exception because the iterator variable is used before definition.

Closure iterators are based on Python's generators and allow the user to define the most complex iterators as required by the search space. Figure 3 shows an example of a *closure iterator* that iterates over prime numbers smaller than or equal to MAX. The new values are generated with the Python yield statement. The return statement or reaching the end of the function terminates the iteration just as is the case in any standard Python code. The *closure iterators* may be thought of as inheriting the functionality available in *expression iterators* and *deferred iterators* with addition of ability to hold on to the internal state between executions of the yield statement. In the figure, this is represented by the old_primes list of previously generated primes. One example of when such a prime number generator would be useful is autotuning an FFT implementation for hard-to-optimize problem sizes [30].

With the iterators described so far, it is possible to express virtually any iteration behavior and in that sense we consider the BEAST language to be functionally complete. The remaining issue is the object-oriented interface promoted by the

```
dim = range( WARP_SIZE, MAX_THREADS+1, WARP_SIZE )
blk_m = range( dim_m, MAX_M+1, dim_m )
```

Fig. 4. Global lexical scope in the BEAST language.

```
@iterator
def blk_n_a(blk_m, blk_k):
  x = blk_k
  if trans_a != 0:
    x = blk_m
  return range(x, 0, -1)
```

Fig. 5. Local lexical scope in the BEAST language.

Python's standard library and the wider Python community. Such interfaces are generally considered more *Pythonic* and, consequently, we reserve the possibility to develop them in the future.

VI. Iterator Constraints

Iterator constraints prune the search space define by the iterator(s); sometime by as much as 99% [14]. Constraints' code executes during the iteration and evaluates (or is cast) to a boolean value. In other words, the True/False value indicates whether a particular tuple of iterator values should/shouldn't be considered in the tuning process. The constraints allow the user to express conditions known to yield good results from the performance engineering standpoint. Because these conditions might involve an interplay between not just one but multiple iterators, and because the conditions could involve non-trivial expressions/statements, sometimes they might be the only means of defining a non-affine search space for the autotuning.

In a similar fashion to the iterators, the iterator constraints come in two types as expression and deferred constraints. The former constraints allow for very casual definition of simple conditions that prune the search space and often drastically reduce the execution time of the autotuner. Due to the fact that the syntax is limited to Python expressions, certain semantics are not available and require the latter type of constraints. The deferred constraints require a Python function definition with the appropriate annotation, which obviously requires more typing and could be prone to errors much more so than simple definitions of expression constraints. But the deferred constraints offer additional functionality that is harder or even impossible to achieve otherwise. Additionally, the deferred constraints can be specified in any order. In particular, the use of a deferred constraint can precede its definition.

VII. Iterator Types and Their Lexical Scopes

A. Lexical Scopes

The language for defining autotuning parameter space is meant for convenience, and, as such, it needs to provide flexibility of programming patterns and expressiveness to

```
@iterator
def fibonacci():
  k = n = 1
  while n <= MAX:
    yield n
    n, k = n+k, n
```

Fig. 6. Closure lexical scope in the BEAST language.

deal with more complicated iteration spaces that call for better structuring of code. Consequently, the BEAST language supports three main iterator types and corresponding lexical scopes:

- global scope,
- local scope, and
- closure scope

The *global scope* allows the user to conveniently define parameter iterators that can be used throughout all the available scopes. In essence, they are Python global variables as shown in Figure 4. In addition to the standard Python semantics, the the BEAST language offers overloaded standard functions and operators that allow for streamlined creation and manipulation of iterators – see Section VIII. The *global scope* is useful for expressing code that is free of side-effects. If, however, extended semantics are needed (e.g., for stateful iterators) or the code needs better structure and organization for clarity, then the other two scopes should be used.

Local scopes are mainly used to control the visibility of names. Creating a local scope hides variables from the *global scope* and allows for better code organization. This is done with Python's function and annotation syntax as shown in Figure 5. While the local scope may serve as an organizational tool to provide a structured definition of iterators with clear syntax, it lacks the ability to attach a state to the iteration process. This kind of functionality requires the third kind of lexical scope that uses closures.

Closure scopes are used for iterators with stateful behavior that is not possible otherwise with side-effect free constructs. They also enable control of visibility because they create a local name space just as was the case for *local scopes*. From the Python syntax standpoint, *closure scopes* are simply generators with the BEAST-defined annotation. The presence of the `yield` keyword in the function code marks the closure scope and the explicit `return` keyword marks the end of iteration. The lack of `return` will create a closure that will stop iteration when the end of the function is reached. In such a situation, Python implicitly executes `return None` statement. Figure 6 shows a sample iterator based on a *closure scope* – the code generates Fibonacci numbers up to and including `MAX`.

VIII. Operating on Iterators and Their Constraints

In Figures 2 and 4 we showed the basic operation (aside from initial construction) on iterators: casting of the iterator to an integer. This is accomplished through overloaded functions from Python's standard library. This feature is more general as the iterator variables can be used in arbitrary expressions because their Python implementation overloads the operator method such as `__add__`. In addition, we added as a matter of convenience the ability to overload some operators that do not allow for overloading such as the ternary operator. The standard operators overloaded for the iterators include arithmetic, binary, logical, and relational iterators. The relational operators prove especially useful for defining constraints as we discuss below.

When overloading Python's builtins and standard classes, we faced a choice of implementing various type-dispatch variants [31, p.125-141][32]. We opted for simplicity and similarity to the semantics of existing languages that the users would be familiar with. Therefore, we use single dispatch throughout our implementation.

An iterator algebra is a functionality that is unique to iterators, and it enables more structured definition of iteration spaces. The user has the flexibility to define the iterators in a way that corresponds closely to the search space, hardware features and software tools, rather than being limited by the syntax (loop nesting, etc.) of the generated code. Consequently, the set-algebra operations, union, intersection, etc., can be used to combine the iterators for expressive search space definition.

A. Operations on Iterator Constraints

Just as the iterators themselves, the constraints are instances of standard Python classes and the comments made above about overloading apply here as well. In particular, the logical operators are well suited for creating complex expressions that correspond to the desired limitations imposed on the search space by the user. The complexity of these logical expressions has an interesting consequence on the performance of the generated code and may limit the options of the optimizing stage to reduce the number of iterations and the running time of the autotuner. In that context, the short-circuiting property of the logical operators becomes an important optimization tool.

IX. The Model Autotuning Problem: GEMM

A. The GEMM Kernel

Figure 7 shows the tiling of the GEMM kernel. Each thread block computes a part of the C matrix in registers, by streaming thin stripes of the A and B matrices through the shared memory.

- **dim_m** $\times$ **dim_n** defines the shape of the thread grid for computing C.
- **blk_m** $\times$ **blk_n** defines the area of C that the thread block is responsible for.
- **blk_m** $\times$ **blk_k** defines the size of the stripe of A in shared memory.
- **blk_k** $\times$ **blk_n** defines the size of the stripe of B in shared memory.
- **dim_m_a** $\times$ **dim_n_a** defines the shape of the thread grid for reading A from device memory to shared memory.
- **dim_m_b** $\times$ **dim_n_b** defines the shape of the thread grid for reading B from device memory to shared memory.

The implementation is also parameterized to handle all cases of transpositions (either A or B transposed, or none, or both) and all four standard LAPACK precisions (single-real, single-complex, double-real, double-complex). We used this basic structure extensively in the past for our autotuning efforts, including tuning for the Fermi architecture [1], [2], the Kepler architecture [3], and tuning for energy [4].

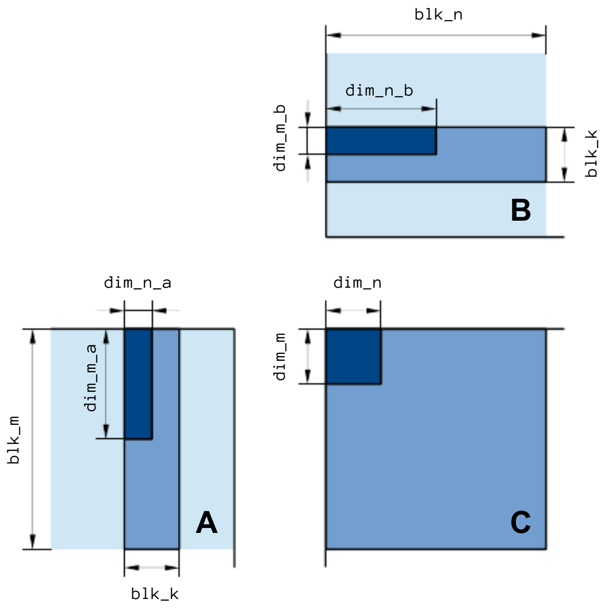

Fig. 7. Tiling of the GEMM kernel: $C \leftarrow \alpha A \times B + \beta C$.

B. The Device Parameters

An important part of the search space and pruning process is the device information. Some of the device information can be queried by using the `cudaGetDeviceProperties` function. Figure 8 shows the device parameters that can be retrieved that way. The values are for Tesla K40c.

Some of the device information cannot be queried, but is available in NVIDIA documentation and tied to the compute capability of the device. These parameters are stored in a table and retrieved using the major number and the minor number of the compute capability, as shown in Figure 9

C. The Autotuning Settings

The autotuning process is carried out separately for each precision and each case of transposition. Therefore, the precision and the input transpose configuration are part of the definition of the search space. Here, we are using the common case of double precision real arithmetic, with both A and B not transposed. Figure 10 shows the settings.

D. The Search Space

The search space is defined by 15 iterators shown in Figure 11. This is a large number of dimensions and demonstrates the hardship of defining the space as a set of nested loops. The

```
max_threads_per_block = 1024
max_threads_dim_x = 1024
max_threads_dim_y = 1024
max_shared_mem_per_block = 49152
warp_size = 32
max_regs_per_block = 65536
max_threads_per_multi_processor = 2048
cudamajor = 3
cudaminor = 5
max_registers_per_multi_processor = 65536
max_shmem_per_multi_processor = 49152
float_size = 4
```

Fig. 8. Device information coming form a device query that is specific to Tesla K40c.

```
MaxBlocksPerMultiProcessor = [
   [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
   [ 8,  8,  8,  8, -1, -1, -1, -1, -1, -1],
   [ 8,  8,  8,  8,  8,  8,  8,  8,  8,  8],
   [16, -1, -1, -1, -1, 16, -1, -1, -1, -1] ]
MaxWarpsPerMultiProcessor = [
   [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
   [24, 24, 32, 32, -1, -1, -1, -1, -1, -1],
   [48, 48, 48, 48, 48, 48, 48, 48, 48, 48],
   [64, -1, -1, -1, -1, 64, -1, -1, -1, -1] ]
MaxRegistersPerThread = [
   [ -1,  -1,  -1,  -1,  -1, - 1,  -1,  -1,  -1,  -1],
   [128, 128, 128, 128,  -1,  -1,  -1,  -1,  -1,  -1],
   [ 63,  63,  63,  63,  63,  63,  63,  63,  63,  63],
   [ 63,  -1,  -1,  -1,  -1, 255,  -1,  -1,  -1,  -1] ]
max_blocks_per_multi_processor =
     MaxBlocksPerMultiProcessor[cudamajor][cudaminor]
max_warps_per_multi_processor  =
     MaxWarpsPerMultiProcessor[cudamajor][cudaminor]
max_registers_per_thread       =
     MaxRegistersPerThread[cudamajor][cudaminor]
```

Fig. 9. Device information coming from a compute capability lookup.

```
precision = "double" ; trans_a = 0
arithmetic = "real"  ; trans_b = 0
```

Fig. 10. Global settings.

search space for the GEMM kernel is defined by the following iterators:

- **dim_m** is the vertical dimension of the thread grid for computing C.
- **dim_n** is the horizontal dimension of the thread grid for computing C.
- **blk_m** is the vertical size of the block's tile of C.
- **blk_n** is the horizontal size of the block's tile of C.
- **blk_k** is the width of a stripe of A and the height of a stripe of B.
- **dim_vec** defines the size of the vector type used in the implementation. Our implementation permits the use of standard types (`float`, `double`, `cuFloatComplex`, `cuDoubleComplex`), built-in vector types (`double2`, `float4`), and a custom type (`cuFloatComplex2`).
- **vec_mul** defines if actual matrix multiplication is performed using vector types, i.e., if A and B in shared memory are accessed using vector types. While A and B can be read from device memory to shared memory using vector operations, they may be read from shared memory to registers using non-vector operations. This will happen, e.g., in the case of **dim_vec**= 4 and **vec_mul**= 0.
- **dim_m_a** is the vertical dimension of the thread grid for reading A.
- **dim_n_a** is the horizontal dimension of the thread grid for reading A.
- **dim_m_b** is the vertical dimension of the thread grid for reading B.
- **dim_n_b** is the horizontal dimension of the thread grid for reading B.
- **tex_a** defines if texture reads are used for reading A.
- **tex_b** defines if texture reads are used for reading B.
- **shmem_l1** defines the shared memory versus L1 cache preference, as set by `cudaFuncSetCacheConfig`.
- **shmem_banks** defines the 4-byte versus 8-byte shared memory bank size, as set by

```
dim_m = range(1, max_threads_dim_x+1)
dim_n = range(1, max_threads_dim_y+1)
@iterator
def blk_m(dim_m):
  return range(dim_m, max_threads_dim_x+1, dim_m)
@iterator
def blk_n(dim_n):
  return range(dim_n, max_threads_dim_y+1, dim_n)
blk_k = range(1, min(max_threads_dim_x, max_threads_dim_y)+1)
@iterator
def dim_vec(arithmetic, precision):
  if arithmetic == "double":
     if precision == "real":
        return range (1, 3)
     else:
        return 1
  else:
     if precision == "real":
        return range(1, 5, 3)
     else:
        return range(1, 3)
@iterator
def vec_mul(dim_vec):
  if dim_vec == 1:
     return 0
  else:
     return range(0, 2)
@iterator
def dim_m_a(blk_m, blk_k):
  if trans_a == 0:
     return range(1, blk_m/dim_vec+1)
  else:
     return range(1, blk_k/dim_vec+1)
@iterator
def dim_n_a(blk_m, blk_k):
  if trans_a == 0:
     return range(1, blk_k+1)
  else:
     return range(1, blk_m+1)
@iterator
def dim_m_b(blk_k, blk_n):
  if trans_b == 0:
     return range(1, blk_k/dim_vec+1)
  else:
     return range(1, blk_n/dim_vec+1)
@iterator
def dim_n_b(blk_k, blk_n):
  if trans_b == 0:
     return range(1, blk_n+1)
  else:
     return range(1, blk_k+1)
tex_a = range(0, 2)
tex_b = range(0, 2)
shmem_l1 = range(0, 2)
shmem_banks = range(0, 2)
```

Fig. 11. Iterators defining the search space for the GEMM implementation.

`cudaDeviceGetSharedMemConfig`.

E. The Pruning Constraints

We can distinguish three classes of pruning constraints: hard, soft, and correctness – all described in the sub-sections that follow. They rely on a set of derived variables shown in Figure 12. These derived variables are:

- **threads_per_block** is the number of threads in a block.
- **thr_m** is the vertical dimension of the local array used by a single thread to store C (intended for registers).
- **thr_n** is the horizontal dimension of the local array used by a single thread to store C (intended for registers).
- **regs_per_thread** is the number of 32-bit registers required by a single thread to store C.
- **regs_per_block** is the number of 32-bit registers required by the block to store C.

```
threads_per_block = dim_m * dim_n
thr_m = blk_m / dim_m
thr_n = blk_n / dim_n
regs_per_thread = thr_m * thr_n
if precision  == "double":
     regs_per_thread = regs_per_thread * 2
if arithmetic == "complex":
     regs_per_thread = regs_per_thread * 2
regs_per_block = regs_per_thread * threads_per_block
shmem_per_block = blk_k * (blk_m + blk_n) * float_size
if precision  == "double":
     shmem_per_block = shmem_per_block * 2
if arithmetic == "complex":
     shmem_per_block = shmem_per_block * 2
max_blocks_by_regs  = max_registers_per_multi_processor / regs_per_block
max_blocks_by_regs  = \
     min(max_blocks_by_regs, max_blocks_per_multi_processor)
max_threads_by_regs = max_blocks_by_regs * threads_per_block
max_blocks_by_shmem  =
     max_shmem_per_multi_processor / shmem_per_block
max_blocks_by_shmem  =
     min(max_blocks_by_shmem, max_blocks_per_multi_processor)
max_threads_by_shmem = max_blocks_by_shmem * threads_per_block
loads_per_thread = (thr_m + thr_n) * blk_k / dim_vec
loads_per_block  = loads_per_thread * threads_per_block
if arithmetic == "complex":
     loads_per_block = loads_per_block * 2
fmas_per_thread = thr_m * thr_n * blk_k
fmas_per_block  = fmas_per_thread * threads_per_block
if arithmetic == "complex":
     fmas_per_block = fmas_per_block * 4
```

Fig. 12. Derived variables.

- **shmem_per_block** is the size of shared memory in bytes, required by the block to store a stripe of A and of B.
- **max_blocks_by_regs** is the maximum number of blocks that can be placed in a single multiprocessor, taking into account the number of registers required by a single block.
- **max_blocks_by_shmem** is the maximum number of blocks that can be placed in a single multiprocessor, taking into account the amount of shared memory required by a single block.
- **loads_per_block** is the number of load instructions from shared memory to registers, executed by each block, in order to process one stripe of A and B.
- **fmas_per_block** is the number of fused multiply add (FMA) instructions, executed by each block, in order to process one stripe of A and B.

Hard constraints are closely tied to hardware parameters. The objective of the *hard constraints* is to eliminate kernels that would fail to compile due to exceeding hardware limits, or that would compile, but fail to launch. At the same time, some of the hard constraints are only a guideline and may eliminate kernels that would successfully run or permit kernels that would fail. The four hard constraints shown in Figure 13 are generally applicable to any kernel.

The hard constraints used here are as follows:

- **over_max_threads** prevents exceeding the maximum number of threads per block. This is an exact limit.
- **over_max_regs_per_thread** prevents exceeding the maximum number of registers per thread. This only means the theoretical demand for registers, not the actual register usage, since the actual usage is up to the compiler.
- **over_max_regs_per_block** prevents exceeding the maximum number of registers per block. As with the previous one, this limit is also only theoretical.

```
@condition
def over_max_threads(threads_per_block):
  return threads_per_block > max_threads_per_block
@condition
def over_max_regs_per_thread(regs_per_thread):
  return regs_per_thread > max_registers_per_thread
@condition
def over_max_regs_per_block(regs_per_block):
  return regs_per_block > max_regs_per_block
@condition
def over_max_shmem(shmem_per_block):
  return shmem_per_block > max_shared_mem_per_block
```

Fig. 13. Hard constraints that are applicable to any kernel.

- **over_max_shmem_per_block** prevents exceeding the size of shared memory per block. This is an exact limit.

Soft constraints shown in Figure 14 are meant to eliminate kernels that are correct, but guaranteed to perform poorly. Similarly to the hard constraints, the *soft constraints* are also fairly generic and, in principle, applicable to any kernel.

```
min_threads_per_multi_processor = 256
min_fmas_per_load = 2
@condition
def low_occupancy_regs(max_threads_by_regs):
  return max_threads_by_regs < min_threads_per_multi_processor
@condition
def low_occupancy_shmem(max_threads_by_shmem):
  return max_threads_by_shmem < min_threads_per_multi_processor
@condition
def low_fmas(loads_per_block, fmas_per_block):
  return fmas_per_block / loads_per_block < min_fmas_per_load
@condition
def partial_warps(threads_per_block):
  return threads_per_block % warp_size != 0
```

Fig. 14. Soft constraints, applicable to any kernel.

Here, we first define two variables:

- **min_threads_per_multiprocessor** defines the lowest desired level of occupancy.
- **min_fmas_per_load** defines the lowest desired number of FMA instructions per each load instruction from shared memory to registers.

Then we form the following constraints:

- **low_occupancy_regs** rejects kernels with maximum possible occupancy lower than desired, due to lack of registers.
- **low_occupancy_shmem** rejects kernels with maximum possible occupancy lower than desired, due to lack of shared memory.
- **low_fmas** rejects kernels with less than desired number of FMA instructions per every load instruction from shared memory to registers.
- **partial_warps** rejects kernels that use a number of threads that is not divisible by the warp size.

Finally, the *correctness constraints* reject kernels that violate assumptions inherent in kernel's algorithmic formulation, such as divisibility of sizes. The particular set of such constraints for GEMM kernels is shown in Figure 15. Violating these constraints produces numerically incorrect results. Clearly, this set of constraints is kernel-specific.

- **cant_reshape_a1** rejects cases where reading A (from device memory to shared memory) would require a different number of threads than computing C.

```
@condition
def cant_reshape_a1(dim_m_a, dim_n_a, threads_per_block):
  return (dim_m_a * dim_n_a != threads_per_block)
@condition
def cant_reshape_b1(dim_m_b, dim_n_b, threads_per_block):
  return (dim_m_b * dim_n_b != threads_per_block)
@condition
def cant_reshape_a2(blk_m, blk_k, dim_m_a, dim_n_a):
  return (trans_a == 0 and \
        ((blk_m % (dim_m_a*dim_vec) != 0 ) or (blk_k % dim_n_a != 0))) \
        or \
        (trans_a != 0 and \ ((blk_k % (dim_m_a*dim_vec) != 0 ) or \
          (blk_m % dim_n_a != 0)))
@condition
def cant_reshape_b2(blk_k, blk_n, dim_m_b, dim_n_b):
  return (trans_b == 0 and \
        ((blk_k % (dim_m_b*dim_vec) != 0) or (blk_n % dim_n_b != 0))) \
        or \
        (trans_b != 0 and ((blk_n % (dim_m_b*dim_vec) != 0) or \
          (blk_k % dim_n_b != 0)))
```

Fig. 15. Correctness constraints.

- **cant_reshape_b1** rejects cases where reading B would require a different number of threads than computing C.
- **cant_reshape_a2** rejects cases where the dimensions of a stripe of A (in shared memory) are not evenly divisible by the dimensions of the thread grid.
- **cant_reshape_b2** rejects cases where the dimensions of a stripe of B are not evenly divisible by the dimensions of the thread grid.

X. Theoretical Framework for Code Generation

A. Dependency DAG Example

The BEAST language can used to express a set of iterators that define the search space, and a set of constraints that prune the parameter space. Examples of iterators are the range of block sizes to test, and the range of thread block dimensions to examine. These ranges may be inter-related, in that, for code correctness, the value of one range depends on the value of other ranges. For instance, in tuning the matrix multiplication (GEMM) GPU kernel, the thread-block dimension must evenly divide the block size (NB). The dependencies among iterators and constraints can be represented by a directed acyclic graph (DAG). An example DAG is given in Figure 16. Loops defining the iterators may be reordered as long as the order (level sets) in the DAG is respected. These concepts are formalized below.

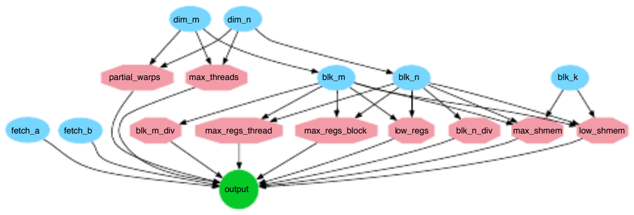

Fig. 16. Graph of dependencies between iterators (blue circles) and constraints (red octagons).

B. Graph Model of Iterators and Conditions

The iterators/constraints and their dependencies may be modeled by a DAG G with vertices V and edges $E : G = (V, E)$. The set of vertices consists of all the iterators/constraints defined by the user: $V = I \cup C$. The dependencies are the edges

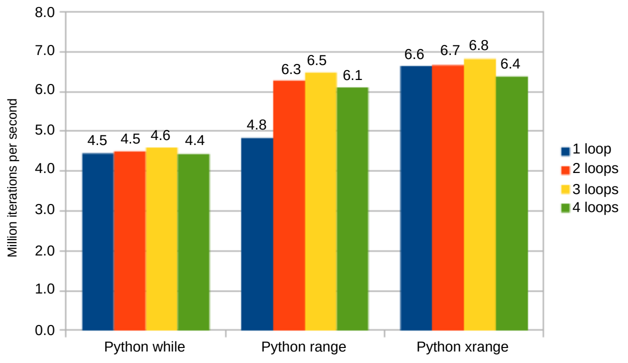

Fig. 17. Performance of simple loops from BEAST autotuner in Python for 10^8 total iterations.

$E = V \times V$ in the graph. For any two vertices $v, w \in V$, there is an edge $e = (v, w) \in E$ if and only if iterator/constraint v is used to express the value of the iterator/constraint w. Note that the transitive closure of G is not a strict superset of G. For example, an empty set of edges occurs when iterators/constraints are not expressed in terms of each other but only in terms of hardware and algorithmic parameters. The dependence relation between the iterators/constraints induces a weak order within the set of vertices that can be used to correctly and efficiently generate loop nests for the exploration of the autotuning search space. Formally, $v \succ w$ (v succeeds w) if $(v, w) \in E$ (v and w are connected) or there exists a path between v and w: $\exists_{u \in V} \mid v \succ u \wedge u \succ w$. The level sets of vertices are defined as $\mathcal{L} = \{L_i \mid \forall_{v,w \in L_i} v \nsucc w \wedge v \nprec w\}$ and can be generated with a greedy traversal of the graph G. The level sets $\mathcal{L}$ of iterators/constraints represent subset of vertices that are unordered with respect to each other and are used to create independent sets of loop nests. Within each level, the loops may be interchanged according to external rules or requirements, for example, to facilitate loop fusion or introduce parallelization (through multithreading or multiprocessing) that can be very beneficial at the outermost loop nests, close to level 0: L_0.

XI. Performance Comparison

A. Hardware and Software Used in Tests

We ran all of our performance tests on Intel Xeon Sandy Bridge E5-2650 v3 2.3 GHz. In order to provide the performance base for tested environments, we present only sequential runs as not all tested environments allow for multithreading to the same extent (see further discussion below). All C and Fortran codes were compiled with the GNU `gcc/gfortran` compiler suite version 4.4.7-16 and flags: `-O2 -march=native -mtune=native`. For Java tests, we used the latest Oracle Java version 1.8.0 update 60 with HotSpot JIT Server. We used Python version 2.7.10 and ran the tested codes with maximum optimization flag: `-OO`. Also, Lua 5.1.4 was used.

B. Python Performance for Search Space Pruning

The majority of tools and utilities for the BEAST project are written in Python, which has been of growing importance in the HPC field over the past years [33]. Hence, we start with the Python language and present in Figure 17 performance of various depths of loop nests (between 1 and 4) across syntactic variants of the implementation. The quantity of merit for the figure is *iterations per second* and we are interested in maximizing this value. The number of total iterations is 10^8 and they are either performed by a single loop, two loops of length $\lceil \sqrt{10^8} \rceil$ each, three ($\lceil \sqrt[3]{10^8} \rceil$), or four (length $\lceil \sqrt[4]{10^8} \rceil$) loop nests. The innermost body of the loop performs integer arithmetic on local variables – there are no memory accesses through mutable containers such as lists or dictionaries. It is clear from the figure that the syntax of the loop matters and the while construct is about 30% slower than the range and xrange syntax. The explanation is that Python's access to variables is through associative array lookup (there is one array per lexical scope) and this, combined with standard handling of loop variables (increments and comparisons) causes the while variant to execute the slowest. When using the range builtin function, the loop overhead is hidden because it is handled inside the interpreter that is written in C and this results in a performance increase. There is still a visible slowdown for a single loop nest with the range construct – this has to do with instantiating in memory a list of 10^8 integers that define the iteration space. This overhead disappears with xrange, which was designed to remove this exact memory overhead and the figure clearly shows that it outperforms the other two solutions. Despite the optimizations, the rate of execution for loop iterations is still too low considering the fact that modern processors can execute in excess of 1000 MIPS. As a point of reference, we generated the autotuning code for the BEAST matrix-matrix multiply kernel and it took 66948 seconds (over 18.5 hours), which is unacceptable for productive autotuning.

As a possible solution, multi-threading can be considered. However, we cannot effectively use multi-threading, on par with other languages. This is due to Python's Global Interpreter Lock (GIL) that prevents simultaneous execution of threads inside Python's Virtual Machine. Also, multi-processing in Python is available as a builtin module but involves kernel-level calls and use of Unix shared memory. It also performs data copies that are not required in a multi-threading scenario. We simply assume that with sufficient effort, any system would benefit equally from multi-threading but the BEAST language system just makes that feature transparent to the user. Finally, it would be hard to breach the performance gap of two orders of magnitude that exists for autotuning search between Python and compiled languages as is shown below.

C. Lua Performance for Search Space Pruning

Our earlier work with BEAST autotuning [4] relied on the Lua language for specification of the search space. Consequently, we show in Figure 18 performance of Lua code that is equivalent to Python tests reported in Figure 17. We see a significant difference in performance between various syntactic variants: using `while` is about 10% slower than the `repeat-until` variant which is about 30% slower than the `for` variant. We also see a significant improvement, 5-fold in

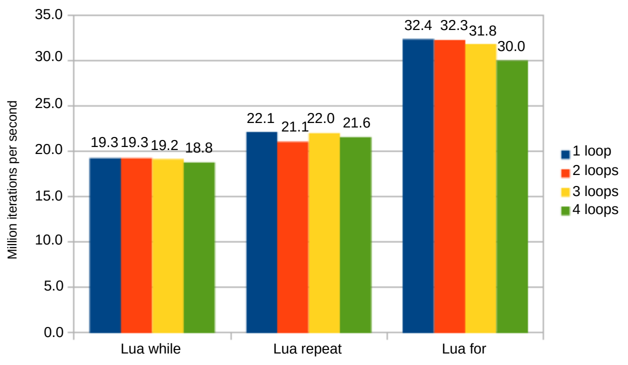

Fig. 18. Performance of simple loops from BEAST autotuner in Lua for 10^8 total iterations.

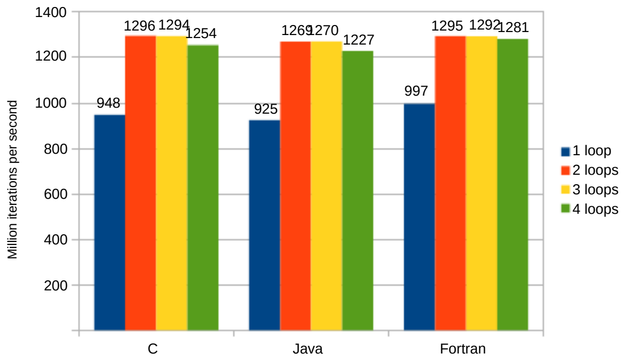

Fig. 19. Performance of simple loops from BEAST autotuner in C, Java, and Fortran for $2^{31} - 1$ total iterations.

fact, over the Python-based iteration. This is still quite a large performance gap from the native performance of the hardware.

D. *Compiled Languages' Speed for Search Space Pruning*

Clearly, the prior experiments indicate that there is a substantial overhead in using scripting or weakly typed languages for synthetic loop nest benchmarks. Additionally, GEMM – a practical case of autotuning – cannot be handled by these languages in a reasonable time frame. This is one of the reasons why we developed the the BEAST language for specifying autotuners and added a code generation to automatically produce fast code to enumerate and prune the search space. We now turn to compiled and strongly typed languages: C, Java, and Fortran as the backends for the BEAST code generator. We start with the same experiment as was performed for Python and Lua. Figure 19 shows the performance for C, Java, and Fortran. We had to increase the total iteration count from 10^8 to the largest signed 32-bit integer, $2^{31} - 1$, to amortize the loop setup and tear-down overheads. Also, functions were made static where possible to increase the compiler's potential for optimization. Java turns out to be the slowest and Fortran the fastest, albeit by a negligibly small margin. Also, the single loop nest turned out to be the worst performing variant. Analysis of the generated assembly code reveals that for more than one loop nest, the compiler generates a better instruction mix that gets higher execution rate per cycle and uses the registers more efficiently. As the ultimate test of the improvement over the Python-based iteration, we ran the BEAST autotuning space sweep with the generated C for BEAST matrix-multiply kernel and it finished in 264 seconds. This is over a 250-fold speedup which allows much more productive autotuning [4].

TABLE I
PERFORMANCE LEVELS ACHIEVED WITH THE BEAST AUTOTUNER.

Kernel name and type	Improvement
GEMM [4]	80% of peak
Batched factorizations (small size) [5]	up to 1000%
Batched factorizations (medium size) [34], [35], [36]	up to 300%

E. *Application Use Cases*

The the BEAST language was also used to develop a number of highly tuned implementations of basic numerical kernels. Table I summarizes the performance improvements that we achieved so far. The GEMM [4] study aimed at not only high performance but at the same time optimal energy consumption. The ability of the BEAST framework to explore the parameter space allowed us to draw conclusions about trade-offs necessary to optimize two objective functions at once. The batch factorizations of a large number of very small matrices is essential in some machine learning tasks and the BEAST implementation was able to deliver superior performance level against any other code currently available [5]. Finally, large counts of mid-sized matrices are common throughout various disciplines of science. For a large range of sizes, our autotuned kernels delivered performance that was up to 3 times faster than any competing implementation [34], [35], [36]. The code for these kernels is either already available inside the MAGMA library or will be released pending final testing as part of, again, MAGMA or the BEAST software release.

XII. Conclusions and Software Release

We showed how the BEAST language makes it easy to describe a complex, multidimensional search space and apply pruning constraints. As our experiments showed, this greatly increases the speed of evaluating the search space. Our notation is declarative rather than prescriptive and we showed how it allows us to apply a variety of aggressive optimizations and generate standard C code for fast and multithreaded evaluation. Unlike other approaches that use kernel code annotations or DSLs, our notation is Python-based and thus instantly familiar to the users: benchmarking engineers or scientists interested in autotuning. We presented autotuning of an important kernel for NVIDIA GPUs with 15 search dimensions and the speed of evaluation is orders of magnitude faster compared to imperative generators in various scripting and compiled languages. Modern accelerators, especially Xeon Phi, offer opportunity to further improve the evaluation speed by utilizing their man-core design and we intend to target them in the future. Also, the plan is to incorporate statistical search methods to address the multidimensional search space growth.

The software described here, the the BEAST language and the accompanying framework, will be released as part of the BEAST software release in 2016. It will be available on the project website[1] and distributed under a permissive, three-clause BSD license.

References

[1] J. Kurzak, S. Tomov, and J. Dongarra, "Autotuning GEMMs for Fermi," Electrical Engineering and Computer Science Department, Tech. Rep. UT-CS-11-671, April 18 2011, also available as LAPACK Working Note 245.

[2] ——, "Autotuning GEMM kernels for the Fermi GPU," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 11, pp. 2045–2057, November 2012.

[3] J. Kurzak, P. Luszczek, S. Tomov, and J. Dongarra, "Preliminary results of autotuning GEMM kernels for the NVIDIA Kepler architecture - GeForce GTX 680." Innovative Computing Laboratory, Tech. Rep. LAWN267, 2012.

[4] H. Anzt, B. Haugen, J. Kurzak, P. Luszczek, and J. Dongarra, "Experiences in autotuning matrix multiplication for energy minimization on GPUs," *Concurrency and Computation: Practice and Experience*, vol. Accepted for publication, 2015.

[5] J. Kurzak, H. Anzt, M. Gates, and J. Dongarra, "Implementation and tuning of batched Cholesky factorization and solve for NVIDIA GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. (submitted), 2015.

[6] M. Gates, H. Anzt, J. Kurzak, and J. Dongarra, "Accelerating collaborative filtering using concepts from high performance computing," in *Big Data (Big Data), 2015 IEEE International Conference on*. IEEE, 2015.

[7] B. Haugen and J. Kurzak, "Search space pruning constraints visualization," in *VISSOFT'14: 2nd IEEE Working Conference on Software Visualization*, Victoria, BC, Canada, 2014, http://vissoft.iro.umontreal.ca.

[8] R. C. Whaley, A. Petitet, and J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parellel Comput. Syst. Appl.*, vol. 27, no. 1-2, pp. 3–35, 2001, DOI: 10.1016/S0167-8191(00)00087-9.

[9] J. Bilmes, K. Asanović, J. W. Demmel, D. Lam, and C.-W. Chin, "Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology," LAPACK Working Note, Tech. Rep. 111, August 8 1996, university of Tennessee Computer Science Technical Report UT-CS-96-326. [Online]. Available: http://www.netlib.org/lapack/lawnspdf/lawn111.pdf

[10] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels." *SciDAC, J. Physics: Conf. Ser.*, vol. 16, pp. 521–530, 2005, dOI: http://dx.doi.org/10.1088/1742-6596/16/1/071.

[11] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, 2005, special issue on "Program Generation, Optimization, and Adaptation".

[12] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005, DOI: 10.1109/JPROC.2004.840306.

[13] R. Veras and F. Franchetti, "Capturing the expert: Generating fast matrix-multiply kernels with spiral," in *High Performance Computing for Computational Science – VECPAR 2014, The Ninth International Workshop on Automatic Performance Tuning (iWAPT)*, M. Daydé, O. Marques, and K. Nakajima, Eds., 2014, pp. 236–244.

[14] C. A. Schaefer, V. Pankratius, and W. F. Tichy, "Atune-IL: An instrumentation language for auto-tuning parallel applications," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, S. B. . Heidelberg, Ed., vol. LNCS, no. 5704/2009, Aug. 2009, publikation, pp. 9–20. [Online]. Available: /Tichy/uploads/publikationen/216/original.pdf

[15] S. Kamil, "Productive high performance parallel programming with auto-tuned domain-specific embedded languages," Ph.D. dissertation, University of California, Berkeley, 2012, tech Report EECS-2012-255.

[16] S. Benkner, F. Franchetti, M. Gerndt, and J. K. Hollingsworth, "Automatic application tuning for hpc architectures," *Dagstuhl Reports*, vol. 3, no. 9, pp. 214–244, January 2014, http://dx.doi.org/10.4230/DagRep.3.9.214; http://drops.dagstuhl.de/opus/volltexte/2014/4423.

[17] V. Volkov, "Better performance at lower occupancy," in *GPU Technology Conference*, 2010. [Online]. Available: http://www.cs.berkeley.edu/\~{}volkov/volkov10-GTC.pdf

[18] R. Nath, S. Tomov, and J. Dongarra, "An improved MAGMA GEMM for Fermi graphics processing units," *Int. J. High Perf. Comput. Applic.*, vol. 24, no. 4, pp. 511–515, 2010, DOI: 10.1177/1094342010385729.

[19] ——, "Accelerating GPU kernels for dense linear algebra," in *Proceedings of the 2010 International Meeting on High Performance Computing for Computational Science, VECPAR'10*. Berkeley, CA: Lecture Notes in Computer Science 6449, June 22-25 2010, pp. 83–92, DOI: 10.1007/978-3-642-19328-6_10.

[20] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun, "Fast implementation of DGEMM on Fermi GPU," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 35:1–35:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063431

[21] F. G. Gustavson, L. Karlsson, and B. Kågström, "Parallel and cache-efficient in-place matrix storage format conversion," Department of Computer Science, Umeå University, Tech. Rep. UMINF 10.05, 2010, http://www8.cs.umu.se/research/uminf/reports/2010/005/part1.pdf (submitted to ACM TOMS).

[22] D. Mukunoki and D. Takahashi, "Implementation and evaluation of triple precision BLAS subroutines on GPUs," in *IPDPSW '12: Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE Computer Society, May 2012.

[23] ——, "Performance comparison of double, triple and quadruple precision real and complex BLAS subroutines on GPUs," in *ATIP '12: Proceedings of the ATIP/A*CRC Workshop on Accelerator Technologies for High-Performance Computing: Does Asia Lead the Way?* A*STAR Computational Resource Centre, May 2012.

[24] W. W. Hwu, Ed., *GPU Computing Gems Jade Edition*, ser. Applications of GPU Computing Series. Morgan Kaufmann, 2011, ISBN: 0123859638.

[25] T. Rompf and M. Odersky, "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs," in *Conference on Generative programming and component engineering, GPCE*, 2010, pp. 127–136.

[26] ——, "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs," *Commun. ACM*, vol. 55, no. 6, pp. 121–130, 2012.

[27] G. Ofenbeck, T. Rompf, A. Stojanov, M. Odersky, and M. Püschel, "Building program generators for high-performance: Spiral on Scala," in *The Ninth International Workshop on Automatic Performance Tuning (iWAPT)*, 2014.

[28] T. Rompf, "Lightweight modular staging and embedded compilers: Abstraction without regret for high-level high-performance programming," Ph.D. dissertation, École Polytechnique Fédérale de Lausanne, 2012.

[29] P. Luszczek and J. Dongarra, "High performance development for high end computing with Python Language Wrapper (PLW)," *The International Journal of High Performance Computing Applications*, vol. 21, no. 2, 2007.

[30] C. M. Rader, "Discrete Fourier transforms when the number of data samples is prime," *Proceedings of IEEE, Proceedings Letters*, pp. 1107–1108, June 1968.

[31] O. Zendra, D. Colnet, and S. Collin, "Efficient dynamic dispatch without virtual function tables. The SmallEiffel compiler." 1997.

[32] J. Bachrach and G. Burke, "Partial dispatch: Optimizing dynamically-dispatched multimethod calls with compile-time types and runtime feedback," 1999, available at http://www.ai.mit.edu/~jrb/Projects/pd.pdf.

[33] A. Terrel, T. Oliphant, and A. A. K. Smith, "Python in HPC," 2013, SC13 Tutorial.

[34] T. Dong, A. Haidar, S. Tomov, and J. Dongarra, "A fast batched Cholesky factorization on a GPU," in *Proceedings of International Conference on Parallel Processing (ICPP-2014)*, Minneapolis, MN, September 2014.

[35] A. Haidar, P. Luszczek, S. Tomov, and J. Dongarra, "Towards batched linear solvers on accelerated hardware platforms," in *8th Workshop on General Purpose Processing Using GPUs (GPGPU 8) co-located with PPOPP 2015*. San Francisco, CA: ACM, February 2015.

[36] ——, "Batched matrix computations on hardware accelerators," *International Journal of High Performance Computing Applications*, 2015, special issue from EuroMPI/Asia 2015 Workshop, Bordeaux, France, September 2015.

[1]http://icl.eecs.utk.edu/beast/