CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2015)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3516

Experiences in autotuning matrix multiplication for energy minimization on GPUs

Hartwig Anzt[1,*,†], Blake Haugen[1], Jakub Kurzak[1], Piotr Luszczek[1] and Jack Dongarra[1,2,3]

[1]*Department of Electrical Engineering and Computer Science (EECS), University of Tennessee, Knoxville, TN 37996-2250 USA*
[2]*Oak Ridge National Laboratory, USA*
[3]*University of Manchester, UK*

SUMMARY

In this paper, we report extensive results and analysis of autotuning the computationally intensive graphics processing units kernel for dense matrix–matrix multiplication in double precision. In contrast to traditional autotuning and/or optimization for runtime performance only, we also take the energy efficiency into account. For kernels achieving equal performance, we show significant differences in their energy balance. We also identify the memory throughput as the most influential metric that trades off performance and energy efficiency. As a result, the performance optimal case ends up not being the most efficient kernel in overall resource use. Copyright © 2015 John Wiley & Sons, Ltd.

Received 30 September 2014; Revised 24 March 2015; Accepted 24 March 2015

KEY WORDS: automatic software tuning; hardware accelerators; matrix multiplication; power; energy

1. INTRODUCTION

For the longest time, runtime performance was the sole metric of interest in high performance computing (HPC). The popularity of this metric is reflected in the longevity of the High Performance LINPACK benchmark (HPL) [1], which serves as the basis for the TOP500 list [2] that ranks the supercomputing installations. In particular, the LINPACK number is the performance achieved when solving a large dense system of linear equations via the LU factorization, a compute-bound operation that is known to deliver a floating-point operations per second (FLOP/s) rate close to that of the dense matrix–matrix product (DGEMM), and therefore the theoretical peak performance of the underlying platform. A consequence of supercomputers growing not only in floating-point performance but also in size and core count is a significant increase in power consumption. Current HPC facilities used for scientific applications typically require multiple megawatts (MW) of power for operation. Power usage of 1 MW a year costs roughly $1 *million* USD, which means that energy has become a significant portion of the cost to acquire, maintain, and operate a large-scale supercomputer. As a result, the energy needs of HPC systems have become a constraint for both hardware and software design[‡].

Hardware manufacturers responded with the development of low-power processors and the integration of manycore accelerators like graphics processing units (GPUs), which traditionally achieve high performance per Watt ratios [3]. For the software stack, dynamic voltage and frequency scaling [4] allows for trading lower power consumption for an extended runtime. This technique particularly

*Correspondence to: Hartwig Anzt, Innovative Computing Lab (ICL), University of Tennessee, Knoxville, USA.
[†]E-mail: hanzt@icl.utk.edu
[‡]US Department of Energy: Top Ten Exascale Research Challenges, http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf.

pays off for memory-bound operations where the computing cores idle while waiting for data. To this end, significant research has focused on improving the energy balance of sparse linear algebra routines that are traditionally limited by the memory bandwidth [3]. At the other end of the spectrum, dense linear algebra operations are traditionally expected to provide the best resource efficiency at the fastest execution (race-to-idle).

In our autotuning work and HPC code design, we follow two particular examples of succesful open-source solutions for very efficient matrix–matrix multiplication. One was done by Volkov *et al*. [5] and the other was done by Nath *et al*. [6, 7]. These efforts showed how it was possible to discover the unknown parameters of the GPU hardware and 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 [8, Section 5].

For manycore accelerators, running at a fixed voltage-frequency setting, tuning for energy efficiency is mostly considered to be equivalent to performance tuning. In this paper, we will show that tuning for performance is not equivalent to tuning for energy, even in the case of dense linear algebra applications. The investigation will focus on NVIDIA GPUs executing the dense matrix-matrix product in double precision (DGEMM), often considered the most basic dense linear algebra routine. We generate a large set of DGEMM kernels using the Bench-testing Environment for Automated Software Tuning (BEAST) [9] autotuner and analyze their performance and energy characteristics. The data reveal that several kernels may perform at the same level but produce a wide variety of energy efficiency results. We explore kernel performance, energy efficiency, and relate to several GPU metrics to identify dependencies. Also, we demonstrate that energy-efficient kernels are not always the fastest kernels as one may assume.

The rest of the paper is structured as follows: in the remainder of this section, we provide a brief review about existing work in the fields of energy analysis, resource-efficient hardware, and autotuning. We then describe in Section 2 the BEAST autotuning framework and the search space and filters that we apply to generate the set of DGEMM kernels we based this paper on. We also introduce the hardware platform with its energy measurement capabilities, and the analysis methodology we follow to guarantee a high accuracy and reproducibility of the results. The main experimental contribution of the paper is Section 3, where we report the experimental results, identify relevant GPU metrics, and analyze the data for dependencies and tradeoffs. We conclude in Section 4 with a short summary of the findings and list future research directions.

1.1. Energy analysis on high performance computing systems

Monitoring power and energy consumption of an application executed on a specific computing platform can be a complicated and tedious process. This problem is exacerbated by the size and complexity of modern HPC systems. The simplest, but very coarse-grained, method is to monitor power between the power outlet and the system and to subtract the idle power to get the application's energy usage. However, this method does not account for alternating current/direct current (AC/DC) loss, increasing fan-speed when heating up, and may therefore provide only a very rough estimate.

A fine-grained external measurement requires monitoring of the current and the voltage in the power lines for distinct components via a third-party system [10, 11]. The beauty of this non-intrusive approach is that the performance of the application subject to energy profiling is not affected by the measurement process if the data are collected in a remote system. Using high-resolution power meters allows for fine-grained resource analysis, which makes this method particularly interesting for moderate-sized clusters [11] or experimental processors [3]. The infrastructure required for the measurement, however, makes the integration into large-scale HPC clusters less attractive.

The increasing focus on resource efficiency has motivated hardware manufacturers to start embedding energy counters in the processors themselves [12, 13]. Profiling tools like performance application programming interface [14] provide user interfaces to these metrics, and allow developers to correlate the power and energy consumption to other application-specific metrics [15]. Similarly, when Cray introduced the XC-30 line, they included the Power Management Data Base

DOI: 10.1002/cpe

(PMDB) [16] tool that makes power and energy information readily available. In particular, it collects power measurements from a comprehensive set of hardware sensors located throughout the system and allows developers to access the power consumption, energy, current, and voltage of the racks, blades, and nodes of an XC-30 system through simple database queries. The Piz Daint GPU-accelerated supercomputer [17] used for the experiments in this paper is equipped with this technology, and a comprehensive study has revealed a very good match between the reported values and external measurement techniques [16].

1.2. Execution parameter tuning for energy efficiency

Significant research has been conducted in order to improve the energy efficiency of scientific algorithms and applications. On CPU-based clusters, a popular non-intrusive method is to optimize the hardware execution parameters like the amount of required computing resources (number of nodes, number of cores, and communication strategy). Also, modifying the task list scheduling can improve the energy balance [18, 19]. Utilizing voltage and frequency scaling exposed by the hardware provides another method to reduce the energy consumption [20, 21]. On manycore accelerators, where modification of the execution parameters is restricted, improving the energy efficiency typically requires a reformulation of the underlying algorithm or a redesign of the implementation [22]. For sparse linear algebra operations, the implementation redesign often aims to reduce the communication and memory transfers. Dense linear algebra operations, typically compute-bound, are traditionally expected to achieve the highest efficiency when reducing the overall runtime.

1.3. Autotuning

Autotuning kernel code for efficient GPU execution has been investigated on different levels of abstraction: from elementary basic linear algebra (BLAS) kernels [23, 24] to high-level languages [25, 26] and directive-based GPU programming [27].

Sparse linear algebra operations have also been targeted: Monakov *et al*. present an autotuning strategy for the sparse matrix vector product based on a problem-adapted matrix storage format [28] and Choi *et al.* [29] propose to use a model-driven framework for autotuning the sparse matrix vector product. Mametjanov *et al.* [30] address the challenge of autotuning stencil operations.

2. AUTOTUNING SETUP

This section will introduce the BEAST autotuning framework and the GEMM kernel configuration. The hardware and data collection methodology is also presented.

2.1. Bench-testing environment for automated software tuning framework

Figure 1 shows the overall design of the BEAST framework, which is currently only partially realized, with the initial software release anticipated to happen at the beginning of 2015. Current implementation includes the hardware probing component, a prototype generation and pruning engine, and an ad hoc infrastructure for compiling and benchmarking (not distributed yet). Results of the initial work on the visualization component have been published [31]. So far, only NVIDIA devices supporting the CUDA framework have been targeted [32].

The current implementation of the generation and pruning component provides a declarative notation for defining the search space and a set of *filters* for pruning it. The notation is currently based on the Lua language [33]. The user defines the search space through a set of iterators, which can be as simple as (*start, stop, step*) triplets or as complicated as closures. Closure iterators contain arbitrary Lua, and return the next value along with an end-of-sequence flag. While the simple iterators address the most common cases, the closure-based iterators allow for handling more complex cases, such as the canonical example of generating the Fibonacci series and returning all possible prime number factorizations of a number. Filters are boolean functions, either accepting or rejecting a set of parameters, and can be defined using values of the iterators, device-dependent constants, user-defined constants, and arbitrarily complex functions thereof.

DOI: 10.1002/cpe

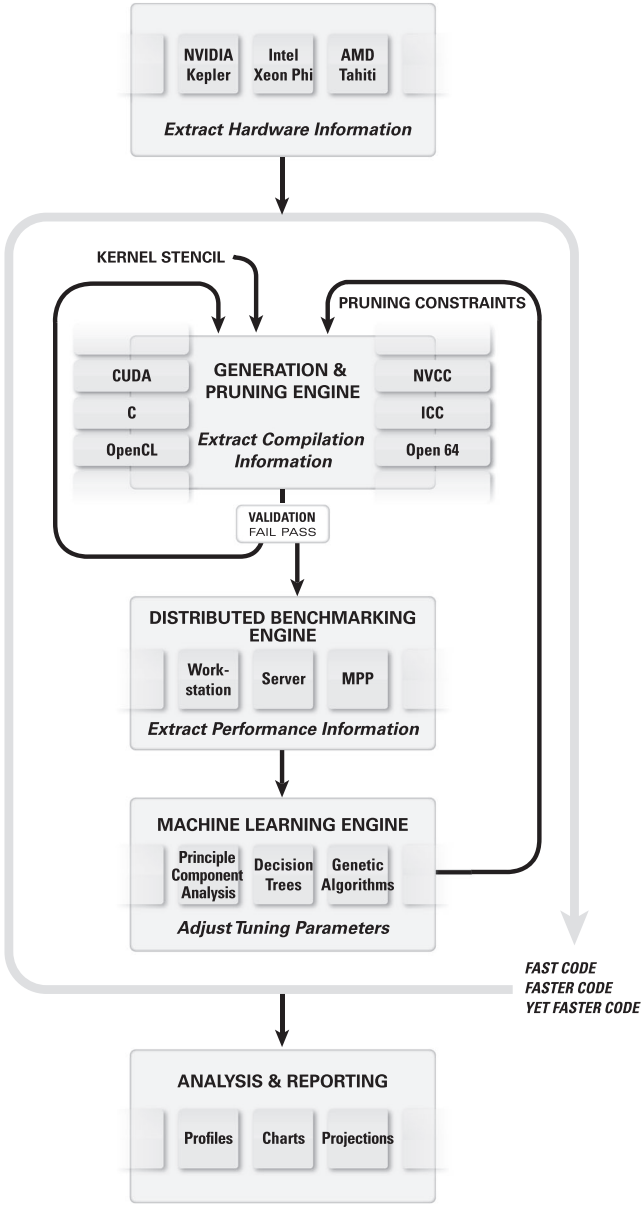

Figure 1. Overall design of the bench-testing environment for automated software tuning framework.

Ongoing efforts in BEAST focus on improving the power of expression and the processing speed of the generation and pruning engine, implementing a distributed engine for benchmarking and profiling, developing a machine learning toolset for analysis of the results, and improving the overall workflow of the entire toolchain.

The BEAST framework acts on the high-level language Compute Unified Device Architecture (CUDA). This implies, that kernels generated may achieve the best performance that can possibly be attained using the CUDA language; however, a better implementation of the operation may be possible when going deeper in the abstraction level. In particular, implementations based on an assembly code might achieve higher performance and energy efficiency ratios.

2.2. GEMM stencil

The tunable GEMM implementation, called the *stencil*, is completely parametrized using the pyexpander preprocessor [34]. Figure 2 shows the five dimensions of tiling in the GEMM stencil. Each

DOI: 10.1002/cpe

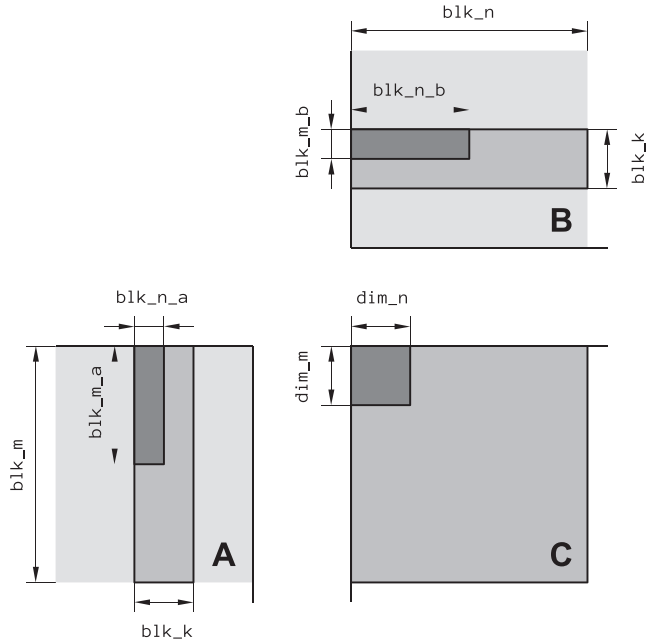

Figure 2. Tiling in the parametrized GEMM stencil.

Table I. Example configuration of the parameters for the GEMM kernel.

	dimension of computed tile	partitioning into stripes	arrangement of reads
A		blk_m = 128 blk_k = 32	blk_m_a = 64 blk_n_a = 64
B		blk_k = 32 blk_n = 128	blk_m_b = 16 blk_n_b = 64
C	dim_m = 128 dim_n = 128		

thread block consists of $dim\_m \times dim\_n$ threads and computes a $blk\_m \times blk\_n$ tile of the matrix C. In doing so, each thread block passes through a block of rows of matrix A of height blk_m and a block of columns of matrix B of width blk_n. In one iteration, the thread block loads a $blk\_m \times blk\_k$ stripe of A and $blk\_k \times blk\_n$ stripe of B. The stripes are first loaded to the shared memory and subsequently loaded to registers by individual threads and applied towards the final product. When reading the stripes, the threads are arranged in the $blk\_m\_a \times blk\_n\_a$ shape for reading A and in the $blk\_m\_b \times blk\_n\_b$ shape for reading B.

For an example kernel using the configuration, see Table I. Each thread block is $dim\_m \times dim\_n = 32 \times 32$ in size, and the total number of threads per block is 1024. Each thread block computes a tile of C of size $blk\_m \times blk\_n = 128 \times 128$. For this, each thread block loads a stripe of A and a stripe of B to shared memory at a time. The stripe of A is of size $blk\_m \times blk\_k = 128 \times 32$, and the stripe of B is of size $blk\_k \times blk\_n = 32 \times 128$. When reading a stripe of A, threads are shaped in a $blk\_m\_a \times blk\_n\_a = 64 \times 16$ rectangle. When reading a stripe of B, threads are shaped in a $blk\_m\_b \times blk\_n\_b = 16 \times 64$ rectangle.

The stencil is parametrized to support both real arithmetic and complex arithmetic in both single precision and double precision, and all cases of transposition of A and B. In this paper, we only consider the real arithmetic, double precision case (DGEMM), where A and B are not transposed. As further explained in Section 2.2.1, the stencil allows for accessing A and B with or without the use of texture reads. In this paper, we always use texture reads, which seems to generally provide higher performance. Also, as further explained in Section 2.2.2, the stencil provides implementations of the kernel with or without the use of Single Instruction Multiple Data (SIMD) vector intrinsics. Here, we

DOI: 10.1002/cpe

simultaneously tune both implementations. And finally, the stencil provides two implementations for reading A and B, one reading them in sequence (first A, then B), the other one reading A and B simultaneously in a single loop, with appropriate cleanup code if their dimensions do not match. Here, we always use the interleaved reading, which seems to always provide higher performance.

The current implementation is a simplification of previous versions [32] and lacks any attempts at multibuffering. The kernel follows a simple cycle:

- read A and B from DRAM to shared memory,
- synchronize threads,
- compute the product, accessing A and B in shared memory,
- synchronize threads,
- advance pointers.

In the process of looping, each thread accumulates the product of $A \times B$. When the looping completes, C is updated by a read-modify-write operation.

2.2.1. Texture reads. Compute Unified Device Architecture supports a subset of the texturing hardware that the GPU uses for graphics to access texture and surface memory. Reading data from texture or surface memory instead of global memory can have several performance benefits (see Section 5.3.2. or the *CUDA Programming Guide* [35] for more details). The A and B matrices can be read using either regular memory reads or texture fetches. Also, they can be defined as either 1D or 2D textures. We have observed that the use of texture fetches is generally beneficial to performance. At the same time, there is no performance difference between using 1D and 2D textures. In this paper, we always read A and B using 1D texture fetches.

There are two different APIs to access texture and surface memory: the texture reference API, which is supported on all devices, and the texture object API, which is only supported on devices of compute capability 3.x. The use of the object API is more convenient from the software development perspective; therefore, we are currently using the object API. The choice of texture API has no effect on performance.

Textures also offer an elegant mechanism for eliminating the *cleanup code*, that is, the code required for dealing with the boundary regions of matrices not divisible by the tiling sizes. The matrix dimensions are rounded up, and texture clipping is used to nullify accesses to elements outside of the A and B matrices. Matrix boundaries are only checked for the C matrix when applying the update at the end of the kernel execution.

Devices of compute capability 3.5 offer a bit simpler alternative to explicit use of textures. A global memory read performed using the `__ldg()` function (see section B.10 of the *CUDA Programming Guide* [35]) is always cached in the read-only data cache. When applicable, the compiler will also compile any regular global memory read to `__ldg()`. A requirement for data to be cached in the read-only cache is that it must be read-only for the entire lifetime of the kernel. In order to make it easier for the compiler to detect that this condition is satisfied, pointers used for loading such data should be marked with both the `const` and `__restrict__` qualifiers. Performace-wise, both techniques are equivalent, and right now, we are explicitly using textures for backward compatibility with older devices.

2.2.2. Single instruction, multiple data vectorization. Although the main form of parallelism in CUDA is the Single Instruction Multiple Thread (SIMT) model, CUDA also offers SIMD vector types, such as `float4` and `double2`, which can improve reads from both DRAM and shared memory (see Section B.3.1 of the *CUDA Programming Manual*). For instance, the use of the `float4` type, instead of `float`, replaces four `ld.global.nc.f32` instruction in PTX with a single `ld.global.nc.v4.f32` instruction (four `LDG.E` instructions with a single `LDG.E.128` instruction in assembly). Similarly, when accessing shared memory, four `ld.shared.f32` instructions in PTX are replaced with a single `ld.shared.v4.f32` instruction (four `LDS` instructions with a single `LDS.128` instruction in assembly).

Although SIMD instructions are available to access data, there are no SIMD arithmetic instructions. Arithmetic relies on the SIMT model. Therefore, when arrays are declared using floating point

DOI: 10.1002/cpe

```
1: dim_m = [ 16 : max_threads_dim_0 : 16 ],
    dim_m = [ 1  : max_threads_dim_0 : 1  ],
2: dim_n = [ 16 : max_threads_dim_1 : 16 ],
    dim_n = [ 1  : max_threads_dim_1 : 1  ],

3: blk_m = [ dim_m : INF : dim_m ],
4: blk_n = [ dim_n : INF : dim_n ],
5: blk_k = [ 1 : INF : 1 ],

6: blk_m_a = [ blk_m : 1 : -1 ],
7: blk_n_a = [ blk_k : 1 : -1 ],
8: blk_m_b = [ blk_k : 1 : -1 ],
9: blk_n_b = [ bkl_n : 1 : -1 ],

10: simd = [ 0 : 1 ],

11: read_a = {'texture', 'no_texture'},
12: read_b = {'texture', 'no_texture'},
13: read_ab = {'interleaved', 'non_interleaved'},

14: shared_mem_bank = {'four_byte', 'eight_byte'},
15: cache_config = {'prefer_L1', 'prefer_shared'}
```

Figure 3. Search space iterators.

vector types, such as `float4` or `double2`, individual components have to be accessed using the *x, y, (z, w)* fields. The SIMD vectorization complicates coding, as dimensions have to be divided or multiplied by the vector size. Furthermore, the divisibility of dimensions introduces a set of new constraints in the process of pruning the search space, which heavily narrows down the number of valid parameter configurations. Nevertheless, we see the SIMD vectorization as an important addition, and we tune both vectorized and non-vectorized GEMM kernels.

2.3. GEMM search space

Like most software autotuners, BEAST relies on the idea of sweeping a search space that is large to start with, but pruned to a managable size by a set of conditions, eliminating cases that would result in invalid kernels or kernels performing poorly. In the two subsections to follow, we describe the set of iterators defining the search space (2.3.1) for the GEMM stencil and a set of filters used for pruning that space (2.3.2).

2.3.1. Iterators. Figure 3 shows the set of iterators defining the search space for the GEMM stencil, using the current notation in the BEAST project. An exhaustive search would sweep through all possible values of all iterators. Here, we introduce some arbitrary constraints to set up a small-scale experiment. The changes are indicated by the grayed out text and further commented on in the rest of this section.

- The `dim_m` and `dim_n` iterators (line 1 and 2) define the shape of the thread block. In an exhaustive search, the increment would be 1. For the purpose of this experiment, the increment is 16. They go up to the hardware limit for the thread block size, which on the Kepler is 1024.
- The `blk_m`, `blk_n`, and `blk_k` iterators (lines 3 to 5) define the basic tiling factors. `blk_m` and `blk_n` define the shape of the tile of C that the thread block is responsible for and iterate in steps of `dim_m` and `dim_n`, respectively. `blk_k` defines the width of the stripe of A and the height of the stripe of B in shared memory and iterates with the step of 1. Theoretically, the upper limit is infinity. In practice, an arbitrary large number is set, for example, 1024, and never reached because of the actions of the filters, described in Section 2.3.2.
- The `blk_m_a`, `blk_n_a`, `blk_m`, `blk_m` iterators (lines 6 to 9) define the arrangements of the threads in the block for reading the stripes of A and B.
- The `simd` iterator (line 10) defines if the SIMD vectorization is used or not, as discussed in Section 2.2.2.

DOI: 10.1002/cpe

- The `read_a` and `read_b` iterators define if texture reads are used for reading A and B. For the purpose of this experiment, we always use texture reads for both A and B.
- The `read_ab` filter defines if A and B are read in an interleaved fashion (inside one loop nest) or not (first A, then B). Here, we always use interleaved reading.
- The `shared_mem_bank` iterator defines if the width of shared memory banks is 4 bytes or 8 bytes. Because we are targeting double precision only, we always use 8-byte wide banks.
- The `cache_config` iterator defines if the 64 KB cache is split into 32 KB of L1 and 32 KB shared memory, 16 KB of L1 and 48 KB of shared memory, or the other way around. Because the GEMM kernel relies on heavy use of shared memory, here, we use the default 16/48 L1/shared memory split.

In the general case, the search space is 15-dimensional. In this particular case, the search space is 10-dimensional. Even with 10 dimensions, defining the search space using imperative code becomes difficult and error-prone, and even more so with 15 dimensions. This motivates our work on a declarative input format, like the one in Figure 3. This comes with its own set of challenges, the main concerns being the power of expression and the speed of evaluation.

2.3.2. Filters. Filters play a vital role in pruning the search space to a manageable size. Our search space is already of manageable size because of the use of a non-unit step for the `dim_m` and `dim_n` iterators and eliminating the last 5 iterators (lines 11–15 in Figure 3). The application of filters is still important for two reasons: First, the GEMM stencil is not completely generalized, that is, some configurations result in invalid code. Filters preserve correctness by, for example, enforcing divisibility of certain dimensions. Second, there are configurations guaranteed to result in inferior performance. Filters eliminate configurations that stand no chance of producing fast code. For this experiment, we used the following set of filters eliminating invalid or low-performing cases:

- The number of threads in the block exceeds the hardware limit. The number of registers in the block equals $dim\_m \times dim\_n$. The hardware limit is a device property (1024 for the Kepler architecture).
- The number of registers per block exceeds the hardware limit. We are using a heuristic here. The assumption is that a tile of C has to fit entirely in registers, that is, the number of registers per block is computed as $blk\_m \times blk\_n$. The actual number of registers required by the kernel depends on the compiler and is usually larger.
- The number of registers per thread exceeds the hardware limit. The number of registers per thread is computed as the number of registers per block divided by the number of threads.
- The amount of shared memory per block exceeds the hardware limit. The amount of required shared memory equals the combined size of the stripe of A and the stripe of B, that is, $blk\_m \times blk\_k + blk\_k \times blk\_n$.
- The number of threads in the block is not divisible by the warp size, that is, $dim\_m \times dim\_n \; mod \; 32 \neq 0$.
- The computational intensity is too low. The computational intensity is computed as the size of the stripes of A and B in shared memory in bytes over the number of floating point operations, specifically the *Fused Multiply-Add* (FMA) operations. Here, we are enforcing no less than four FMA operations per one byte of shared memory.
- The occupancy, defined as the ratio between the number of threads (grouped in thread blocks) that are scheduled in parallel and the hardware-imposed thread limit (threads per streaming multiprocessor), is too low (limited by registers). The occupancy is computed as the maximum number of threads that can be launched in one multiprocessor, given the register usage. Here, we require no less than 512 threads per multiprocessor.
- The occupancy is too low (limited by shared memory). The occupancy is computed as the maximum number of threads that can be launched in one multiprocessor, given the shared memory usage. Here, we also require no less than 512 threads per multiprocessor.
- If vector types are not used, we enforce blk_m_a divides blk_m; blk_n_a divides blk_k; blk_m_b divides blk_k; and blk_n_b divides blk_n. If vector types are used, then we enforce $blk\_m\_a \times sizeof(double2)$ divides blk_m, and so on.

DOI: 10.1002/cpe

The filters are simple heuristics, meant to eliminate kernels that are invalid or guaranteed to perform poorly. Specifically, the formula used for register usage usually underestimates the actual number, that is, the compiler assigns many more registers per thread. Because the size of the thread block is not known until launch time, also unknown is the actual number of registers required by the thread block. Therefore, a number of kernels will fail to launch if exceeding the register file size. Right now, we catch failures to launch and discard the failed kernels. In the future, we plan to retrieve the actual register usage from the compilation step and skip benchmarking of faulty kernels.

2.4. Hardware and energy measurement capabilities

The Piz Daint Supercomputer at the Swiss National Computing Center in Lugano was listed in November 2014 as the sixth fastest supercomputer on the TOP500 [36], while its energy efficiency ranked number nine in the Green500 [37]. A single node is equipped with an 8-core 64-bit Intel Sandy Bridge CPU (Intel Xeon E5-2670), an NVIDIA Tesla K20X with 6 GB GDDR5 memory, and 32 GB of host memory. The nodes are connected by the 'Aries' specialized interconnect from Cray, with a dragonfly network topology [17]. Piz Daint has 5272 compute nodes, corresponding to 42,176 CPU cores in total – with the possibility to use up to 16 virtual cores per node when hyperthreading is enabled, that is, 84,352 virtual cores in total – and 5,272 GPUs. The peak performance is 7.8 Petaflops [17]. PMDB enables the user to monitor power and energy usage for the host node and separately for the accelerator at a frequency of 10 Hz [16].

2.5. Profiling methodology

In a preprocessing step, we use the cascaded filtering process of the BEAST autotuner to identify 770 kernel configurations valid for a DGEMM kernel on Kepler K20 GPUs. The target system size of 4096 for the dense matrix–matrix multiplication in double precision ensures that we are in the asymptotic range of performance and energy efficiency; see Figure 4. In the profiling process, we run every kernel for at least 30 s, to ensure a sufficient number of power measurements of the PMDB power monitoring tool [17]. To avoid temperature-related variations, every executable prefixes a heatup phase using the respective target kernel. An additional kernel launch is used to measure runtime performance, and one more to analyze the kernel metrics using NVIDIA's nvprof profiler [38]. Also, we include a cuBLAS DGEMM kernel to identify possible hardware slowdown. In order to examine the reproducibility of the data, we launched some kernels 10 times and reported the average, minimum, and maximum energy consumption, as well as the mean deviations. The analysis revealed performance differences of less than 1%, and the energy differences within a range of 3% when running on the same node of the Piz Daint cluster. We therefore consider the data based on a single sweep as valid. At this point, we want to stress that although we observed only negligible differences when comparing the results from different nodes, a particular hardware device may have slightly higher or lower performance and energy characteristics. This, however, does not affect the relevance of the experimental results, as the ratios are expected to remain constant, and we executed all tests on the same node. The target metric we use to quantify the energy efficiency is GFLOPs/W, which serves as the basis for the Green500 list [37], and provides insight about how many operations can be executed using a given energy budget:

$$\frac{\text{performance}}{\text{power}} = \frac{\text{work per time unit}}{\text{energy per time unit}} = \frac{\text{work}}{\text{energy}}. \tag{1}$$

The advantage of using this metric over the plain consumed energy is that it is independent of the size of the target problem, assuming asymptotic performance of the GEMM kernel.

Recently, metrics accounting for performance and energy efficiency have become popular [39]. This approach is generally designed as a tradeoff between energy usage e and runtime t using a weight α in the objective function to minimize

$$M(t, e) = \alpha \cdot t + (1 - \alpha) \cdot e. \tag{2}$$

DOI: 10.1002/cpe

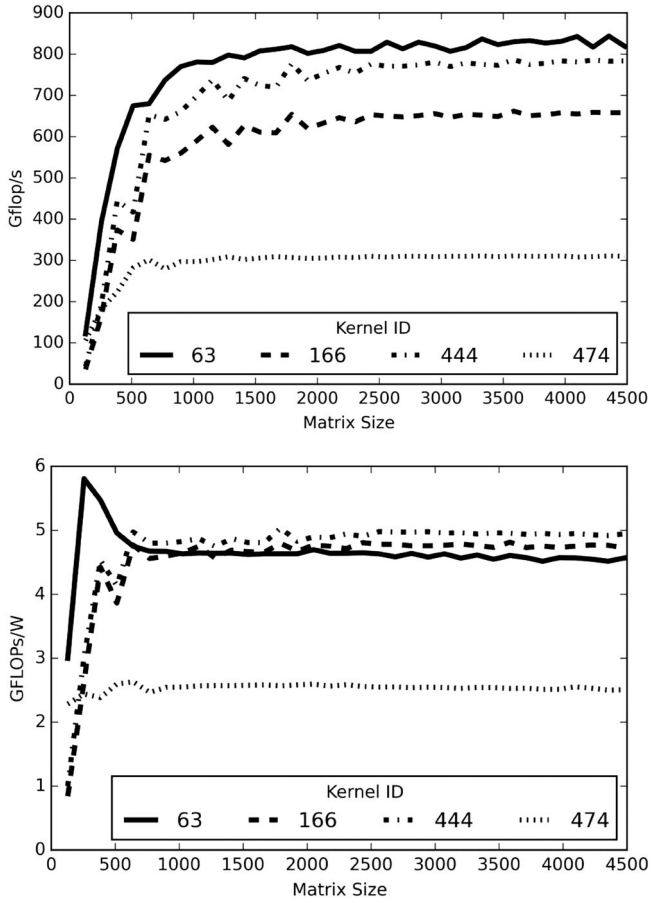

Figure 4. Scaling of performance (top) and energy efficiency of selected kernel configurations with respect to system size.

3. EXPERIMENTAL RESULTS

In the following sections, we report the experimental results and identify metrics that are meaningful when quantifying the kernel's performance and resource efficiency. For these metrics, we present a correlation analysis to detect dependencies and tradeoffs.

We analyze the results under the assumption that the autotuning sweep contains kernels that reach a local limit of attainable performance and energy efficiency while the other metrics vary. This allows us to enclose the datapoints in a convex hull (Figure 5) and to assume that any other kernel configuration obtained by varying the tuning parameters we used in the BEAST DGEMM framework will be included in this area. The kernel configurations determining the corner points of the convex hull are listed in Table II.

3.1. Performance analysis

In Table II, we list the parameter configurations along with the runtime performance and energy efficiency for the kernels that form the convex hull in Figure 5. The achieved performance ranges from 308 GFLOPs to 904 GFLOPs, with an average of 567 GFLOPs. Most kernels provide between 400 and 700 GFLOPs. The configuration providing the best performance achieves 63% of the theoretical peak [40]. For the execution of the DGEMM, it required 30.87 $Joules$, which results in an energy efficiency of 4.49 GFLOPs/W. Although outside the main focus of this paper, for completeness, we mention that the cuBLAS DGEMM achieves 1146 GFLOPs on this hardware and hence outperforms all kernels generated by this BEAST autotuning sweep. Also, the corresponding energy efficiency

DOI: 10.1002/cpe

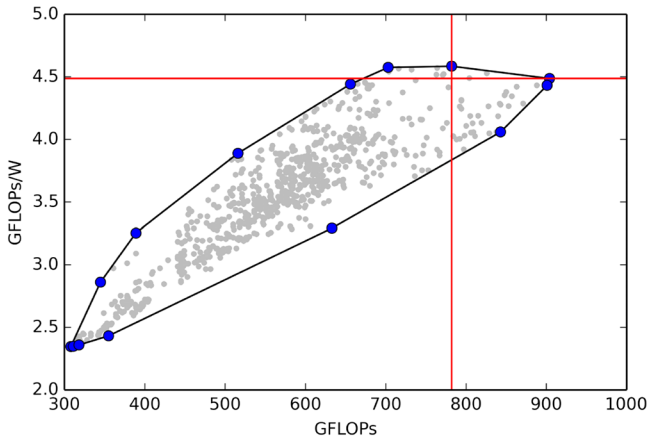

Figure 5. Energy efficiency (GFLOPs/Watt) versus performance (GFLOPs)

Table II. Parameter configurations of selected kernels defining the convex hull in Figure 5 along with the achieved runtime performance and energy efficiency.

	Parameter configuration									Perf.	Energy eff.
	dim_m	*dim_n*	*blk_m*	*blk_n*	*blk_k*	*m_a*	*n_a*	*m_b*	*n_b*	[GFLOPs]	[GFLOPs/W]
upper arc	16	16	64	96	16	64	4	16	16	904	4.4867
	32	16	128	128	16	32	16	8	64	782	4.5849
	16	16	128	128	8	64	4	8	32	703	4.5754
	16	16	128	128	8	32	8	4	64	656	4.4415
	16	16	64	256	8	32	8	4	64	516	3.8891
	16	16	128	128	2	128	2	2	128	389	3.2539
	16	16	64	128	4	64	4	2	128	345	2.8617
lower arc	48	16	48	96	16	48	16	8	96	308	2.3473
	64	16	64	64	16	64	16	16	64	311	2.3488
	16	48	96	48	16	48	16	16	48	318	2.3590
	64	16	64	64	48	64	16	16	64	355	2.4323
	32	16	64	64	24	64	8	8	64	633	3.2919
	16	16	64	80	16	32	8	16	16	843	4.0591
	16	16	80	80	16	16	16	16	16	901	4.4317

The displayed kernel order is obtained from traversing the convex hull counterclockwise, starting at the maximum energy efficiency.

of 5.98 GFLOPs/W is not achieved by any of the BEAST-generated kernels. The primary reason for this performance gap and efficiency loss is the fact that the autotuning framework, sophisticated as it is, relies on the CUDA language, which, in this particular instance, is unable to exploit the full performance potential of the Kepler GPU. On the other hand, the cuBLAS DGEMM includes proprietary assembly instruction mixes that are not easily generated by the compiler. This implies that, at the present time and likely in the future, these two performance levels cannot be matched by either a pure CUDA implementation or a specific mix of PTX instructions [8, Section 5]. Moreover, the close nature of cuBLAS does not give us any information with regard to the performance-energy trade-offs that the NVIDIA team made when shipping their current implementation. Our approach provides a limited view into the kinds of choices that face an HPC kernel coder.

3.2. Energy efficiency

In Figure 5, we visualize the energy efficiency of the DGEMM kernels with respect to their runtime performance. The most resource-efficient kernel requires 29.98 Joules for the execution at 782 GFLOP/s, which results in an energy efficiency of 4.58 GFLOPs/W; see Table II. This kernel

DOI: 10.1002/cpe

performs GEMM with 3% higher energy efficiency while executing 16% slower than the fastest kernel. Although these differences look marginal, they already contradict the common assumption that tuning for runtime performance is sufficient to identify the configuration with the highest energy efficiency. The theoretical energy efficiency of this GPU is 6.08 GFLOPs/W when computed as a ratio between the theoretical peak of 1430 GFLOPs and the maximal power draft of 235 W, The most energy efficient kernel achieves 75%, while the average kernel achieves (3.55 GFLOPs/W) 58% of the efficiency peak. We notice that the generated kernels are closer to the optimal energy efficiency than to the optimal performance in terms of the average and metric-specific winner.

3.3. Performance and energy spread

In Figure 5, a convex hull is used to define an enclosure of the attainable performance-energy-efficiency ratios. Any vertical line provides information about the energy efficiency range of kernels providing the same runtime performance. Similarly, any horizontal line identifies kernel configurations achieving different GFLOP rates for a fixed energy efficiency. The length of the section of line that intersects the convex hull provides the spread for performance or energy efficiency when fixing the other parameter.

As previously identified, the performance winner achieves 904 GFLOPs at a resource efficiency of 4.49 GFLOPs/W. Kernels located on the horizontal line need the same amount of energy to compute the DGEMM, but require up to 25% longer runtimes. The GFLOP rate of the most resource-friendly configuration is achieved by other kernels located on the vertical line that require up to 1.2 times the energy. This implies that ignoring the resource usage when developing GPU kernels close to the attainable performance limit can result in up to 20% energy overhead.

3.4. Multi-objective optimization

The horizontal and vertical lines partition the convex hull into four segments. Kernels located in the area right of the vertical line achieve higher performance than the most resource friendly configuration – at the cost of lower energy efficiency. For our BEAST sweep, 36 kernels (4.6%) are located in this area. Similarly, the 12 kernels located in the area above the horizontal line achieve a higher energy efficiency than the performance winner – at the cost of extended runtime.

The upper, right quadrant represents the kernels that are more resource-friendly than the performance winner and achieve a performance higher than the most energy-friendly kernel.

When trading off energy efficiency and performance, the arc of this segment is of particular interest, as kernels located on it optimize the metric given in function (2) for a particular value of α.

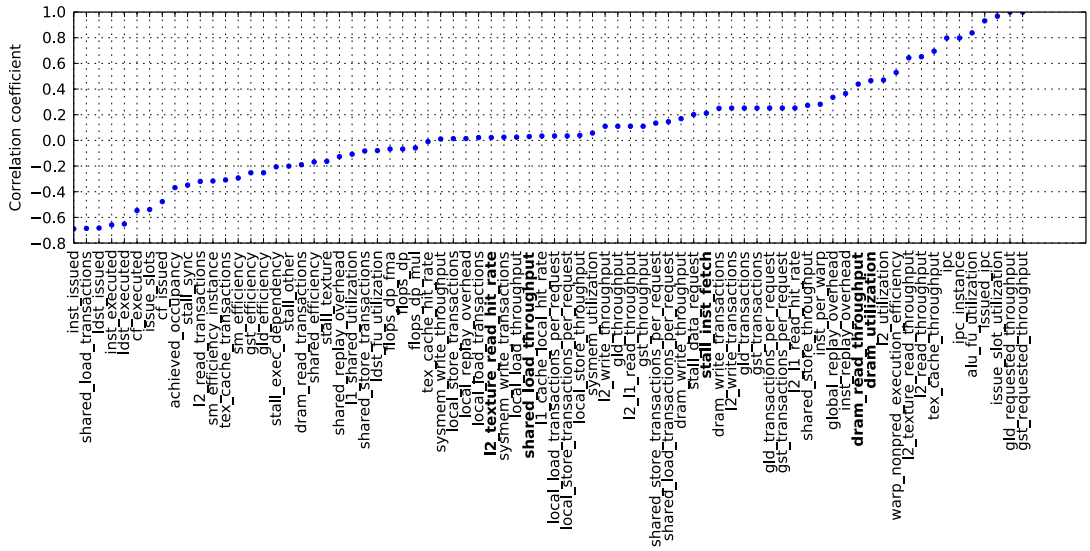

Figure 6. Correlation coefficients of the collected performance metrics and the GFLOPs value.

DOI: 10.1002/cpe

3.5. *Correlation of metrics*

We now turn to the statistical analysis of the data resulting from the autotuning process. Using NVIDIA's nvprof profiler [35], we collected the values of the performance counters, which gave us a multidimensional view into the kernel characteristics of the individual DGEMM kernels, which we can now relate to the performance and energy efficiency of the GPU. Figures 6, 7, and 8 show the correlation coefficient between the collected counter readings and raw performance, energy efficiency, and the product of both of them (performance × energy efficiency) combined, respectively. The correlations are computed against statistically normalized values. All three figures sort the counters based on their correlation coefficient (negative correlation left, positive correlation right). To track the movement of labels between the figures, which indicates a change in how a counter reading affects the correlated metric, we mark the counters moving significantly between performance (Figure 6) and energy efficiency (Figure 7) in bold. These metrics are as follows: L2 texture read hit rate (l2_texture_read_hit_rate), shared memory load throughput (shared_load_throughput), stalls

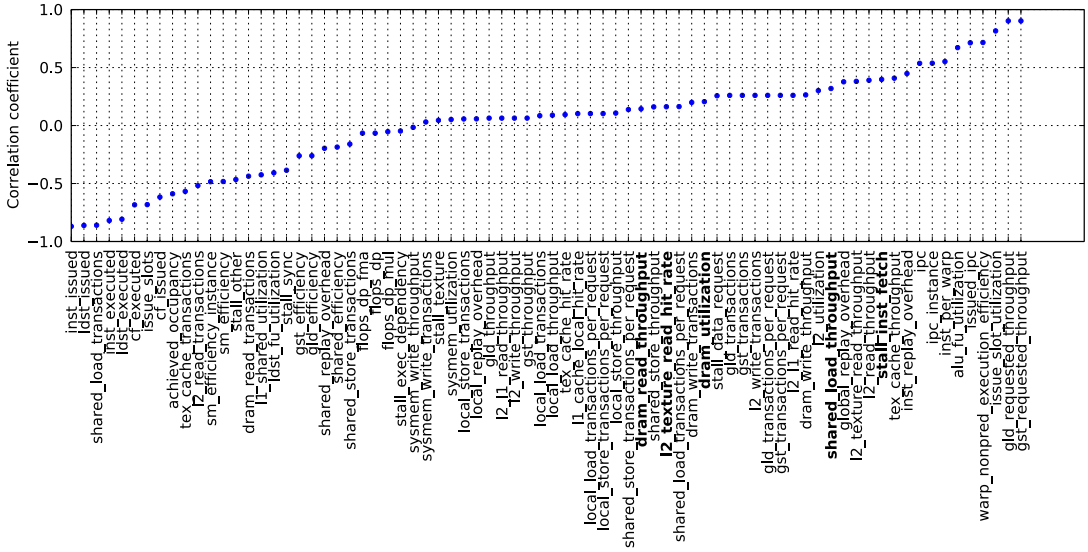

Figure 7. Correlation coefficients of the collected performance metrics and the GFLOPs/W value.

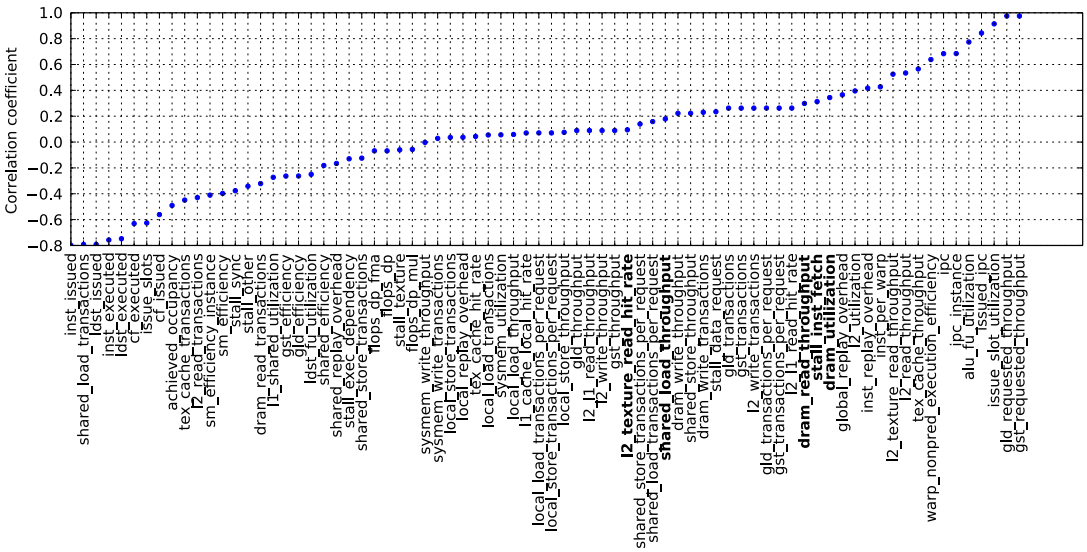

Figure 8. Correlation coefficients of the collected performance metrics and the GFLOPs value.

DOI: 10.1002/cpe

due to an unfetched instruction (stall_inst_fetch), DRAM read throughput (dram_read_throughput), and DRAM utilization (dram_utilization). The first three metrics move from left to right when switching from performance to energy efficiency. This may be expected, as larger values of any of these metrics contribute to short range data movement within the GPU and thus have a positive effect on the energy efficiency. The last two metrics moving from right to left relate to DRAM communication, which may be expected to incur high energy cost. It is worth noting the highly influential performance counters remain the same across Figures 6, 7 , and 8. The negatively correlated counters include shared_load_transactions, which counts the number of shared memory load transactions, and inst_issued, ldst_issued, inst_executed, and ldst_executed counters accounting for various types of instructions. As a general trend, we observe that more instructions decrease performance, energy efficiency and the combination of both. We have positive correlation for the counters for global load (gld_requested_throughput) and store (gst_requested_throughput), and, generally, IPC-related (Instructions Per Cycle) metrics (issued_ipc, ipc_instance, ipc). This, again, is an intuitively understood observation that high IPC levels indicate good performance, and from the energy efficiency standpoint, high IPC indicates lack of data movement stalls, which are detrimental to energy consumption.

3.6. Kernel metrics and energy/performance tradeoff

In Section 3.5, we have identified some metrics that are either strongly correlated for both performance and energy efficiency or show significant differences in their influence on performance and energy efficiency. We choose DRAM_read_throughput, L2_read_ throughput, texture_memory_read_throughput, executed_instructions_per_cycle, and occupancy for a deeper analysis. For each of those metrics, we compute the mean over the complete sweep of 770 kernels, and for each kernel, we determine whether its metric value is above or below the average. Figure 9 contains five rows and three columns. Each of the rows represents one of the metrics, while the columns show runtime performance versus metric, energy efficiency versus metric, and energy efficiency versus runtime performance. Each kernel is color-coded in red or green to indicate whether the metric being described is above or below the average of all kernels in the dataset.

- **Occupancy**
  High occupancy implies that few threads are unused during the execution of an application. The generated kernels separate in four discrete occupancy levels at 12.5%, 25%, 37.5%, and 50%; see Figure 9 first row, left. The natural intuition is that high occupancy is necessary for high runtime performance; however, the DGEMM kernels running at high occupancy generally achieve lower GFLOPs. This is consistent with observations that reducing occupancy can lead to performance benefits for specific algorithms [5]. The corresponding resource efficiency plot (center of first row in Figure 9) reveals that higher occupancy rates can also lead to lower energy efficiency for a fixed level of performance. If the data are broken into four subsets based on the four distinct values for occupancy and the mean for each set is calculated and plotted, an inverse correlation between energy efficiency and occupancy emerges. The final plot on the top row shows clustering of high occupancy kernels near the bottom, left corner of the convex hull. This demonstrates that lower occupancy can clearly lead to a more energy efficient kernel.
- **Instructions per cycle (IPC)**
  The left of the second row in Figure 9 reveals a linear trend: a high instruction per cycle rate is essential to achieve high performance. Varying the different kernel parameters, it is possible to generate configurations providing higher or lower GFLOP rates at a constant IPC rate, but for exceeding a certain performance limit, the IPC has to increase. The absolute values are interesting: the top performers execute close to two instructions per cycle, which indicates high resource utilization. Concerning energy efficiency, things are not as clear (center plot). Kernels with a low IPC rate can still provide very good energy efficiency. Also, the right in Figure 9 shows that for a certain performance rate, it is possible to improve energy efficiency without changing the IPC rate.

DOI: 10.1002/cpe

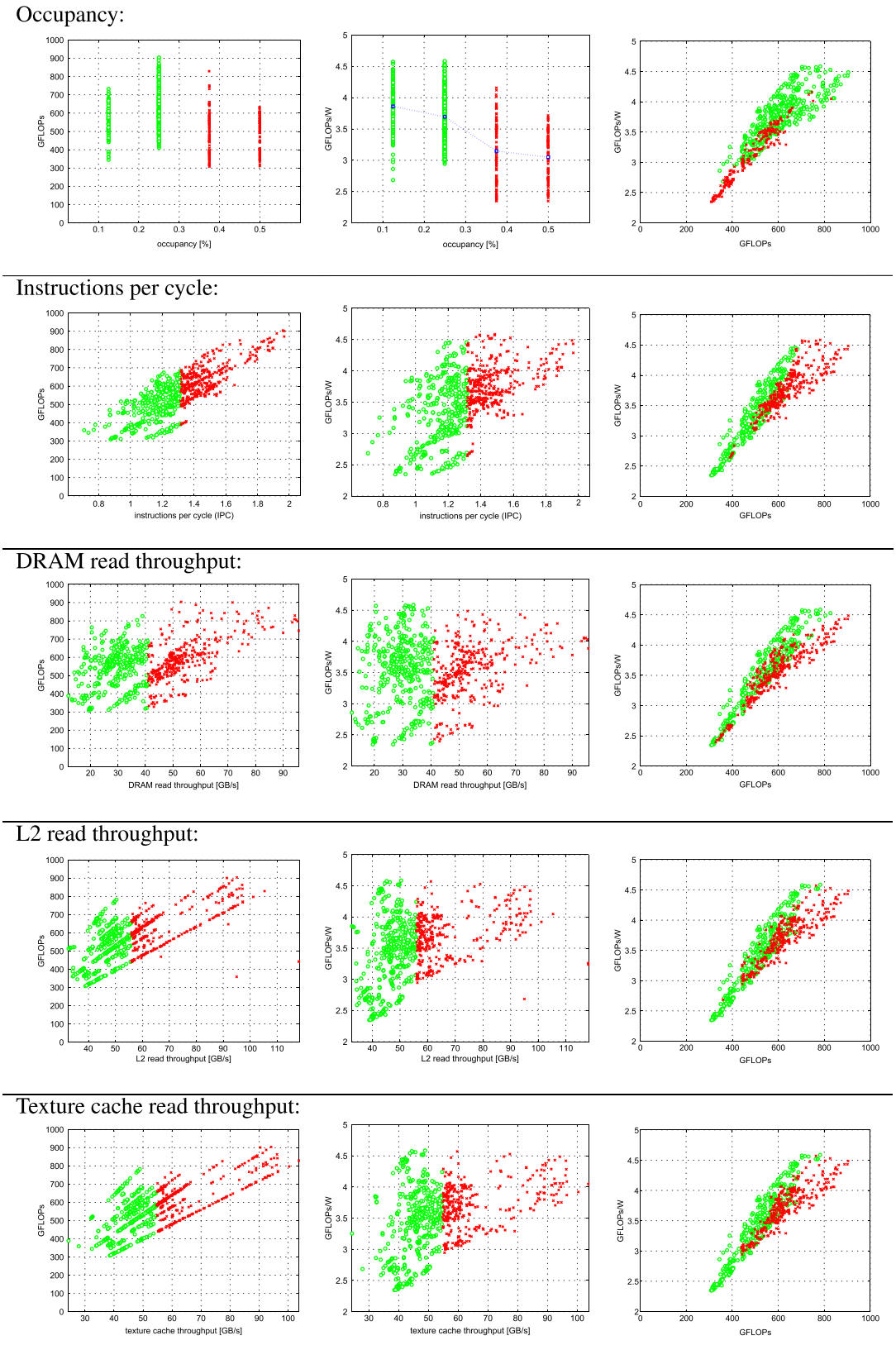

Figure 9. Classification of the generated kernels with respect to the values of different metrics above (red) or below (green) average along with the achieved runtime performance and energy efficiency.

DOI: 10.1002/cpe

- **DRAM read throughput**
  In the third row of Figure 9, we analyze the correlation between DRAM throughput, performance, and energy efficiency. The center of the third row in Figure 9 indicates that DRAM-stressing kernels provide lower energy efficiency. This confirms research results identifying the memory access as being among the most energy-consuming operations [41–43]. At the same time, the bandwidth rates for all kernels exceeding 820 GFLOPs are above the mean (see the right plot). This indicates that pressure on the memory reduces the energy efficiency, but is needed to achieve good performance. The right plot shows a clear separation of the red squares and the green circles along the efficiency/performance diagonal. Kernels putting less pressure on the memory bandwidth form the upper half of the convex hull, and all kernels below the diagonal have a DRAM throughput rate higher than the average.
- **L2 read throughput**
  A first observation when analyzing the L2 read throughput in the fourth line in Figure 9 is that the kernels arrange, in discrete bands, linearly correlating increasing L2 read throughput with increasing runtime performance (see left). Furthermore, only kernels with throughput rates above the average exceed the 800 GFLOPs limit. But, like for the DRAM metric, also accessing L2 cache increases the resource cost. As a result, kernels with high L2 read throughput achieve lower energy efficiency than other kernels providing the same performance (see the center plot). Another interesting observation is that in neglecting two artifacts, a lower bound for the energy efficiency relates linearly to the L2 read throughput. While increasing L2 read throughput generally results in low efficiency, it ensures that the efficiency does not drop below a certain lower bound. Like the DRAM throughput, the L2 throughput rate splits the convex hull relating energy efficiency by the diagonal, and almost all kernels located above this diagonal have an L2 throughput rate below average (see the fourth row of Figure 9, right).
- **Texture read throughput**
  Texture cache is a read-only option to bypass the L1 cache [35]. Concerning the impact on performance and energy, however, it shows similar trends like accessing data via the cache hierarchy: performance and throughput are correlated in a characteristic pattern of discrete stripes (left), and high throughput is necessary for high performance (left), but detrimental for energy efficiency (center). The right plot shows even sharper separation of the energy efficient kernels requiring less texture read throughput and the performance-tuned kernels pushing the throughput to the limit.

Embracing the different types of memory access (DRAM, L2 cache, texture cache) with the general term *memory throughput*, we conclude that when tuning the DGEMM kernel for energy efficiency, low throughput is preferred. At the same time, only high memory throughput allows for high performance. Moderate occupancy is beneficial for both runtime performance and energy efficiency. A high IPC rate is required for high performance, but is less important for energy efficiency.

4. SUMMARY AND FUTURE WORK

In this paper, we have summarized observations from autotuning of the dense matrix–matrix multiplication for energy efficiency on NVIDIA GPUs. For a set of 770 kernels, we have collected performance, energy, and GPU metrics. Investigating the trade-off between performance and energy efficiency, we have identified metrics that can be used to classify the kernels. In particular, we have shown that stressing the memory bandwidth is detrimental to the kernels' energy efficiency, but necessary to achieve high performance. As a result, the most resource-friendly configuration does not provide the highest runtime performance, contradicting the race-to-idle assumption of the fastest kernel being also the most energy-efficient. Future research will focus on more complex kernels that combine compute-bound with memory-bound computations. Also, we want to develop a model capable of predicting energy efficiency and performance of a kernel based on the respective parameter configuration and a calibrating sweep covering only a subset of the search space.

DOI: 10.1002/cpe

ACKNOWLEDGEMENTS

This work is supported by grant #SHF-1320603: 'Bench-testing Environment for Automated Software Tuning (BEAST)' from the National Science Foundation, the Russian Scientific Fund, Agreement N14-11-00190, and also in part by the NVIDIA, Intel and AMD corporations. We would like to thank the *Oak Ridge National Laboratory* (ORNL) for access to the Titan supercomputer, where development of the BEAST project takes place. We would also like to thank the *Swiss National Computing Centre* (CSCS) for granting access to their system and support in deploying the energy measurements.

REFERENCES

1. Dongarra JJ, Luszczek P, Petitet A. The Linpack benchmark: past, present, and future. Concurrency and computation: practice and experience. *Concurrency and Computation: Practice and Experience* 2003; **15**:2003.
2. Meuer HW, Strohmaier E, Dongarra JJ, Simon HD. Top500 supercomputer sites, 42nd edition, 2013. The report can be downloaded from (Available from: http://www.netlib.org/benchmark/top500.html) [accessed on November 2014].
3. Aliaga JI, Anzt H, Castillo M, Fernández JC, León G, Pérez J, Quintana-Ortí ES. Unveiling the performance-energy trade-off in iterative linear system solvers for multithreaded processors. *Concurrency and Computation: Practice and Experience* 2015; **27**:885–904. DOI: 10.1002/cpe.3341.
4. Genser A, Bachmann C, Steger C, Weiss R, Haid J. Power emulation based DVFS efficiency investigations for embedded systems. *2010 International Symposium on System on Chip (SOC)*, Tampere, Finland, 2010; 173–178.
5. Volkov V. Better performance at lower occupancy. *GPU Technology Conference*, Silicon Valley, California, 2015. Conference Presentation.
6. Nath R, Tomov S, Dongarra J. An improved MAGMA GEMM for Fermi graphics processing units. *International Journal of High Performance Computing Applications* 2010; **24**(4):511–515.
7. Nath R, Tomov S, Dongarra J. Accelerating GPU kernels for dense linear algebra. *Proceedings of the 2010 International Meeting on High Performance Computing for Computational Science, VECPAR'10*, Lecture Notes in Computer Science 6449, Berkeley, CA, 2010; 83–92.
8. Tan G, Li L, Triechle S, Phillips E, Bao Y, Sun N. Fast implementation of DGEMM on Fermi GPU. *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC'11*, ACM: New York, NY, USA, 2011; 35:1–35:11.
9. Anzt H, Dongarra J, Gates M, Haugen B, Kurzak J, Luszczek P, Marin G, Tomov S. Bench-testing Environment for Automated Software Tuning (BEAST), 2014. (Available from: http://icl.eecs.utk.edu/beast/) [accessed on November 2014].
10. Choi J, Dukhan M, Liu X, Vuduc R. Algorithmic time, energy, and power on candidate HPC compute building blocks. *IPDPS '14 Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, Phoenix, AZ, USA, May 2014; 447-457. (extitto appear).
11. Ge R, Feng X, Song S, Chang HC, Li D, Cameron KW. Powerpack: energy profiling and analysis of high-performance systems and applications. *IEEE Transactions on Parallel and Distributed Systems* 2010; **21**(5): 658–671.
12. David H, Gorbatov E, Hanebutte UR, Khanna R, Le C. RAPL: memory power estimation and capping. *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '10, ACM: New York, NY, USA, 2010; 189–194.
13. NVIDIA Corporation. *NVIDIA CUDA TOOLKIT V6.0*, 2013.
14. Performance application programming interface (PAPI), 2014. (Available from: http://icl.cs.utk.edu/papi/) [accessed on November 2014].
15. McCraw H, Terpstra D, Dongarra J, Davis K, Roy M. Beyond the CPU: hardware performance counter monitoring on blue gene/q. *Proceedings of the International Supercomputing Conference 2013*, LNCS 7905, Springer: Heidelberg, 2013; 213–225.
16. Fourestey G, Cumming B, Gilly L, Schulthess TC. First experiences with validating and using the Cray power management database tool, 2014.
17. Piz Daint computing resources, 2014.
18. Seo E, Jeong J, Park S, Lee J. Energy efficient scheduling of real-time tasks on multicore processors. *IEEE Transactions on Parallel and Distributed Systems* 2008; **19**(11):1540–1552.
19. Merkel A, Stoess J, Bellosa F. Resource-conscious scheduling for energy efficiency on multicore processors. *Fifth ACM SIGOPS EuroSys '10 Proceedings of the 5th European Conference on Computer Systems*, Paris, France, 2010; 153–166.
20. Alonso P, Dolz MF, Mayo R, Quintana-Orti ES. Improving power efficiency of dense linear algebra algorithms on multi-core processors via slack control. *2011 International Conference on High Performance Computing and Simulation (HPCS)*, Istanbul, Turkey, 2011; 463–470.
21. Alonso P, Dolz MF, Mayo R, Quintana-Ortí ES. Energy-efficient execution of dense linear algebra algorithms on multi-core processors. *Cluster Computing* 2013; **16**(3):497–509.
22. Anzt H, Quintana-Ortí ES. Improving the energy efficiency of sparse linear system solvers on multicore and manycore systems. *Philosophical Transactions of the Royal Society A – Mathematical, Physical and Engineering Sciences* Published 19 May 2014:2014 372 20130279. DOI: 10.1098/rsta.2013.0279.

23. Davidson A, Owens J. Toward techniques for auto-tuning gpu algorithms. *Applied Parallel and Scientific Computing* Jónasson K (ed.)., Lecture Notes in Computer Science, vol. 7134, Springer: Berlin Heidelberg, 2012; 110–119.
24. Matsumoto K, Nakasato N, Sedukhin SG. Performance tuning of matrix multiplication in OpenCL on different GPUS and CPUS. *High Performance Computing, Networking Storage and Analysis, SC Companion* 2012; **0**:396–405.
25. Grauer-Gray S, Xu L, Searles R, Ayalasomayajula S, Cavazos J. Auto-tuning a high-level language targeted to GPU codes. In *2012 Innovative Parallel Computing (InPar)*, San Jose, California, 2012; 1–10.
26. Tillmann M, Karcher T, Dachsbacher C, Tichy WF. Application-independent autotuning for gpus. In *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, vol. 25, Bader M, Bode A, Bungartz HJ, Gerndt M, Joubert GR, Peters F (eds)., Advances in Parallel Computing. IOS Press: Amsterdam, 2014; 626–635.
27. Magni A, Grewe D, Johnson N. Input-aware auto-tuning for directive-based GPU programming. *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, ACM: New York, NY, USA, 2013; 66–75.
28. Monakov A, Lokhmotov A, Avetisyan A. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *Proceedings of HiPEAC'10*. Springer-Verlag: Berlin, Heidelberg, 2010; 111–125.
29. Choi JW, Singh A, Vuduc RW. Model-driven autotuning of sparse matrix-vector multiply on GPUS. *SIGPLAN Notices* 2010; **45**(5):115–126.
30. Mametjanov A, Lowell D, Ma CC, Norris B. Autotuning stencil-based computations on GPUS, 2012.
31. Haugen B, Kurzak J. Search space pruning constraints visualization. *2014 Second IEEE Working Conference on Software Visualization (VISSOFT)*, September 29–30, 2014; pp. 30–39. DOI: 10.1109/VISSOFT.2014.15.
32. Kurzak J, Luszczek P, Tomov S, Dongarra J. Preliminary results of autotuning GEMM kernels for the NVIDIA Kepler architecture – GeForce GTX 680. *Technical Report 267, LAPACK Working Note*, Rio de Janeiro, 2012.
33. Ierusalimschy R. *Programming in Lua* (Third). Lua.Org, 2012.
34. Pfeiffer G. Pyexpander - a powerful macro processing language, 2014. (Available from: http://pyexpander.sourceforge.net/) [accessed on November 2014].
35. NVIDIA Corporation. *NVIDIA CUDA TOOLKIT V5.5*, 2013.
36. The top 500 list, 2014. (Available from: http://www.top500.org/) [accessed on November 2014].
37. The Green 500 list, 2012. (Available from: http://www.green500.org/) [accessed on November 2014].
38. NVIDIA Corp. *NVIDIA Profiler Users Guide V6.0*, 2014. DU-05982-001_v6.0.
39. Bekas C, Curioni A. A new energy aware performance metric. *Computer Science - Res. & Dev.* 2010; **25**:187–195.
40. NVIDIA Corp. TESLA K40 GPU ACTIVE ACCELERATOR, Board Specifications, 2013. BD-06949-001_v03.
41. Vogelsang T. Understanding the energy consumption of dynamic random access memories. *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO'43, IEEE Computer Society: Washington, DC, USA, 2010; 363–374.
42. Mittal S. A survey of architectural techniques for DRAM power management. *International Journal of High Performance Systems Architecture* 2012; **4**(2):110–119.
43. Kestor G, Gioiosa R, Kerbyson DJ, Hoisie A. Quantifying the energy cost of data movement in scientific applications. *2013 IEEE International Symposium on Workload Characterization (IISWC)*, Atlanta, Georgia, September 2013; 56–65.

DOI: 10.1002/cpe