Applying the Roofline Model

Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros, Daniele G. Spampinato, Markus Püschel
Department of Computer Science
ETH Zurich, Switzerland
{ofenbeck, caparrov, danieles, pueschel}@inf.ethz.ch

***Abstract*—The recently introduced roofline model plots the performance of executed code against its operational intensity (operations count divided by memory traffic). It also includes two platform-specific performance ceilings: the processor's peak performance and a ceiling derived from the memory bandwidth, which is relevant for code with low operational intensity. The model thus makes more precise the notions of memory- and compute-bound and, despite its simplicity, can provide an insightful visualization of bottlenecks. As such it can be valuable to guide manual code optimization as well as in education. Unfortunately, to date the model has been used almost exclusively with back-of-the-envelope calculations and not with measured data. In this paper we show how to produce roofline plots with measured data on recent generations of Intel platforms. We show how to accurately measure the necessary quantities for a given program using performance counters, including threaded and vectorized code, and for warm and cold cache scenarios. We explain the measurement approach, its validation, and discuss limitations. Finally, we show, to this extent for the first time, a set of roofline plots with measured data for common numerical functions on a variety of platforms and discuss their possible uses.**

I. Introduction

Software performance is determined by the complex interaction of source code, the compiler used, and the architecture and microarchitecture on which the code is run. Because of this, performance is hard to estimate, understand, and optimize. This is particularly true for numerical or mathematical functions that often are the bottleneck in applications from scientific computing, machine learning, multimedia processing, and other domains. Common practice in performance optimization is to try various optimization techniques together with repeated runtime measurements until a desired performance is achieved. The result is then reported in a graph that plots the achieved performance (say, in floating point operations per second or flop/s) against a range of input sizes. This plot reveals the efficiency achieved (percentage of peak performance) and shows the trend as the size increases, but nothing more. On the other hand, tools like Intel VTune [3] provide large amounts of data for a given code extracted from performance counters [13], [20], such as various types of cache and TLB misses and others. Extracting meaning from this data is daunting for most. As aid in the understanding and optimization of performance it hence seems important to have more ways of visualizing performance that have the simplicity of a performance plot but that provide additional insights. The recently introduced roofline model [37] tries to provide exactly that. It plots performance against operational intensity, which is the quotient of flop count and the memory traffic in bytes. Such a plot reveals new information: the memory bandwidth becomes an additional upper bound for computations with low intensity, the plot clearly distinguishes memory and compute bound computations (terms that are often used casually), and it shows the behavior of intensity across sizes, just to name a few. The downside is that the necessary data is considerably harder to obtain in a reliable and meaningful way than in an ordinary performance plot. Indeed, the original paper [37] shows only four programs with a fixed input size, and the many papers that have used the model to date have done so through back-of-the-envelope calculations rather than measurements.

Contribution. The first contribution of this paper is a strategy on how to produce roofline plots with measured data on recent generations of Intel platforms. We show that by a suitable measurement strategy and the use of the right set of performance counters, meaningful and reliable results can be achieved. This includes roofline plots for single- and multi-threaded code and for cold and warm data cache scenarios.

The second contribution is a detailed performance analysis of common numerical kernels such as BLAS functions and the fast Fourier transform (FFT) with the roofline model. We show for the first time roofline plots (which are quite different from the usual performance plots) for these in various scenarios including single/multithreaded and cold/warm data cache. The plots clearly reveal the different operational intensities between kernels, the effect of common optimizations, but also some surprising behavior not visible in ordinary performance plots. We show and discuss to which extent a roofline plot can be used to study optimizations.

Organization. We first provide background on operational intensity and the roofline model. Then we introduce our measurement methodology and explain how we validated our approach. Finally, we show a set of experiments on various platforms. After a discussion of related work we conclude.

II. Background

In this section we provide the necessary background on the roofline model. First, we explain the concept of operational intensity and then discuss the actual roofline model (see [37] for the original paper). We also briefly discuss performance counters, which are necessary to turn the model into roofline plots. Throughout the paper, properties of executed programs (runtime, performance, etc.) are written using upper case letters, whereas platform properties (peak performance, cache size, etc.) are written using lower case Greek letters.

BLAS function		$W(n)$	$Q_r(n)$	$Q_w(n)$	$Q(n) = Q_{r+w}(n)$	$I_r(n)$	$I(n) = I_{r+w}(n)$
daxpy	$\mathbf{y} \leftarrow \alpha\mathbf{x} + \mathbf{y}$	$= 2n$	$\geq 16n$	$\geq 8n$	$\geq 24n$	$\leq \frac{1}{8}$	$\leq \frac{1}{12}$
dgemv	$\mathbf{y} \leftarrow \alpha A\mathbf{x} + \beta\mathbf{y}$	$= 2n^2 + 2n$	$\geq 8n^2 + 16n$	$\geq 8n$	$\geq 8n^2 + 24n$	$\leq \frac{n+1}{4n+8} \approx \frac{1}{4}$	$\leq \frac{n+1}{4n+12} \approx \frac{1}{4}$
dgemm	$C \leftarrow \alpha AB + \beta C$	$= 2n^3 + 2n^2$	$\geq 24n^2$	$\geq 8n^2$	$\geq 32n^2$	$\leq \frac{n+1}{12}$	$\leq \frac{n+1}{16}$

TABLE I
OPERATIONAL INTENSITY ANALYSIS FOR SOME BLAS1–3 ROUTINES.

A. Operational Intensity

Work. As commonly done, we denote with W the *work*, i.e., the number of operations performed by a given program. Although W may refer to any type of operation, such as comparisons or integer arithmetic, the focus of this paper is numerical code and thus W will count the number of *floating point* additions and multiplications. This is meant in a mathematical sense, i.e., for example one AVX addition of vectors of four doubles will count as four. In our case studies, the work will depend only on the sizes of the inputs, for example when adding two vectors of length n. In this case we write $W = W(n)$. The work is a property of the chosen algorithm and does not depend on the platform.

When run on a platform with some input, the program will achieve a runtime of T and thus a performance of $P = W/T$.

Memory traffic. We denote with Q the number of bytes of memory traffic incurred by executing a given numerical program. Again we write $Q = Q(n)$ to express the dependency on the input size n. In contrast to W, Q heavily depends on the properties of the platform such as the details of the cache hierarchy. Because of this, Q can be estimated only asymptotically in most cases (e.g., [15]), and exact values have to be determined by measurements. We will distinguish reads and writes as Q_r and Q_w: $Q = Q_r + Q_w$. The read data traffic $Q_r(n)$ for sizes for which the data fits within the cache is equal to the number of compulsory misses, i.e., reading the input data. For example, for dgemm (third row in Table I) the three matrices, each of size n^2, have to be read. Therefore, the compulsory read data traffic in bytes is $24n^2$. For larger sizes, there will also be reads associated to data that has been evicted from the cache, but this number cannot be determined analytically. Hence, $Q_r(n) \geq 24n^2$. In dgemm, only the output matrix C has to be written (plus all the evicted data). Therefore, the write data traffic in bytes is $Q_w(n) \geq 8n^2$.

Typically, we assume a cold data cache unless stated otherwise. The optimal Q is closely related to the I/O complexity [15], but may include also traffic not incurred by data, such as loads due to the page table or the cache coherency protocol.

Operational intensity. The *operational intensity*, I, relates W and Q:

$$I = \frac{W}{Q}. \tag{1}$$

That is, I determines the number of floating point operations per byte of memory traffic. We also define $I_r = W/Q_r \geq I$, and $I_w = W/Q_w \geq I$.

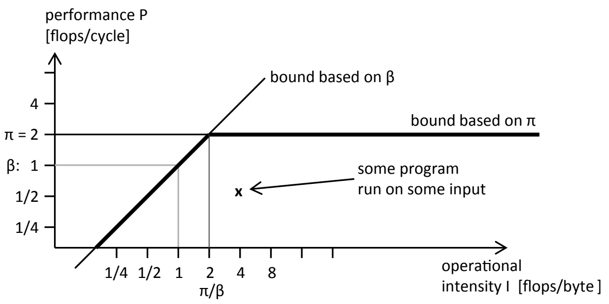


Fig. 1. Roofline plot for $\pi = 2$ and $\beta = 1$.

Operational intensity: Examples. In simple cases, I can be estimated by analysis. As example, consider the daxpy routine in BLAS [1], which implements

$$\mathbf{y} \leftarrow \alpha\mathbf{x} + \mathbf{y}$$

where $\mathbf{x}$ and $\mathbf{y}$ are vectors of doubles of length n and α is a scalar. The work is $W(n) = 2n$. Since there is no reuse in the computation, the data traffic is determined by the compulsory misses and the write-back of the result, i.e., $Q_r(n) \geq 16n$, $Q_w(n) \geq 8n$, and thus $Q(n) \geq 24n$. This analysis yields a lower bound since it excludes, for example, memory traffic due to loading the code and the cache coherency protocol in case of threading. A similar analysis can be done for other mathematical kernels as shown in Table I for representative BLAS 1–3 functions. Note that only dgemm is not bound by $O(1)$. However, the shown bound is too loose once $3n^2$ exceeds the cache size. The achievable optimum (using suitable blocking) is $\Theta(\sqrt{\gamma})$, where γ is the cache size [15]; [14] provides a non-asymptotic bound. Analysis of a simple triple loop implementation yields $I(n) = O(1)$ due to bad locality.

B. The Roofline Model

The roofline model [37] visually relates performance P and operational intensity I of a given program to the platform's peak performance and memory bandwidth. We call the visualization a *roofline plot*; an example is shown in Fig. 1. On the x-axis is the operational intensity, on the y-axis performance. Both are in log scale. We use cycles instead of seconds to abstract from the CPU frequency. A given numerical program run on a given input has a specific operational intensity I and a specific performance P and thus becomes one point in the plot. P obeys two upper bounds: the peak performance π of the processor (in Fig. 1: $\pi = 2$ flops/cycle) and the bound obtained from the memory bandwidth β (in Fig. 1: β = 1 byte/cycle) for computations with low intensity:

$$P \leq \min(\pi, I\beta).$$

The intersection of the two bounds is at $I = \pi/\beta$. Computations with this intensity are exactly those that are balanced in the sense of Kung [17].[1] Computations with $I \leq \pi/\beta$ are memory bound; those with $I \geq \pi/\beta$ are compute bound. The work in [8] uses the term "balanced" to denote compute-bound computations.

The model can be extended to include more bounds [37]. For example, the peak performance changes if one allows the use of vector instructions or the use of multiple cores. Similarly, the effective memory bandwidth changes (is reduced) if a program has no spatial locality. Depending on the bounds used, the notion of memory- and compute-bound changes. Finally, one can build an analogous model and plot by considering only read traffic by using I_r and the peak read bandwidth β_r.

C. Hardware Performance Counters

Most modern microachitectures include hardware support to monitor microprocessor activity. Intel platforms, for example, contain a Performance Monitoring Unit (PMU) that consists of a set of a machine specific registers (MSRs), that can be programmed to count microprocessor events, such as instructions retired, elapsed core clock ticks, or L2/L3 cache hits and misses. The Intel Performance Counter Monitor (PCM) [13] is a set of C++ routines that provide access to these counters. The main advantage of PCM compared to other such libraries is its support for both core events and uncore events, i.e., those measured in the memory controller. This enables the measurement of bytes read from and written to memory controller, which, as discussed later, is helpful in measuring the memory traffic Q.

III. Methodology

To construct roofline plots (e.g., Fig. 1), we need to measure three code-specific quantities: W and Q to compute I, and T to also compute P. In this section we first describe our general measuring strategy, and then explain how each quantity W, Q, and T is obtained using performance counters. We explain validation and discuss caveats. Finally, we briefly explain how π and β are obtained.

A. Measuring strategy

At a high level of abstraction, our measuring strategy has the form:

```
nr_of_runs = get_nr_of_runs(target_code, data)
for (nr_of_repeats){
    start_measure()
    for (nr_of_runs) {
      target_code(data)
    }
    stop_measure()
}
```

[1]More precisely, Kung assumes also full utilization, i.e., that the peak performance π is reached and thus the time for computation and the time for memory traffic are the same.

The approach uses two loops. The outer loop repeats the measurement for statistical purposes to obtain median and quartile information. We choose `nr_of_repeats` = 20 in all experiments. The final result is the median. In the plots we also show, for each point, the quartiles (25^{th} and 75^{th} percentiles) along both axes (P and I) through (vertical and horizontal) lines of appropriate length.

The inner loop reduces the error induced by the measuring overhead [18], [34]. The number of runs `nr_of_runs` is determined by an auxiliary routine that ensures that the total measurement time is larger than a certain threshold; we determined that for our machines a threshold of a 10^8 cycles is sufficient.

The sketched strategy produces warm cache measurements.

Cold cache measurement. Cold cache measurements expose additional information since they include all compulsory cache misses. A naive approach would flush the cache before each computation (`target_code(data)` in the code above). The flush routines could be:

- Using the clflush instruction if the data addresses are available.
- Flushing up to LLC by reading a large buffer.
- Flushing the TLB by accessing data across many pages.
- Flushing the instruction cache by running code as large as the cache.

Unfortunately, for small sizes the expensive flush invalidates the measurement. For the cold cache experiments within this paper, we use a different approach, similar to the one described in [34], that takes the form:

```
for (nr_of_repeats){
    start_measure()
    for (nr_of_runs) {
      target_code(replica_of_data[run])
    }
    stop_measure()
}
```

Note that we are not using any flushing. Instead, we change the working data set in every iteration, thus making sure that it is not cache resident when the target code executes. To ensure cold cache between repetitions, we enforce a lower limit on the number of runs chosen, such that (γ is the LLC size)

$$\text{sizeof(data)} \times \text{runs} \geq \gamma \times \text{associativity}.$$

Further, the data (arrays of doubles in our experiments) for each iteration is allocated separately (each aligned to cache line boundaries) to eliminate the effect of prefetching from one iteration to the next. We limit the size of data replication to avoid excessive memory consumption and unwanted side effects that may come with it. This limit is implemented as

```
...
      target_code(replica_of_data[run%(limit)])
...
```

where `limit` is chosen similar to the lower bound for the runs:

$$\text{sizeof(data)} \times \text{limit} \geq \gamma \times \text{associativity}.$$

Note that while this strategy enables measurements with cold input/output data, it cannot control other cached information, such as the target code and data allocated by the target code.

Cold cache and write-back cache. When dealing with a write-back cache, data up to the size of the LLC is potentially not transferred back to memory before the measurement ends. Also, we might measure write traffic not due to our code, but evicted in order to provide space for our data. We resolve this by executing the target code before starting our measurements such that it fills the LLC with data. Once the measurement starts, the initial evictions compensate the data that are not written back to memory at the end of the measurements.

B. Measuring Work W

Counters for floating point operations. Table II lists the counters used for measuring floating point operations. These counters are only incremented when either arithmetic or comparison instructions are issued. Memory instructions and shuffles are ignored by the counters. For vector instructions, the measured values have to be multiplied by the corresponding vector length. Thus, the total number of double floating point operations on a Sandy Bridge platform is

$$W = \text{Scalar\_double} + \text{SSE\_double} \times 2 + \text{AVX\_double} \times 4,$$

and for the Westmere platform

$$W = \text{Scalar} + \text{SSE} \times 2.$$

Caveats. For the Sandy Bridge based microarchitectures, six counters per core need to be programmed, if both single and double precision floating point operations should be measured simultaneously. Each physical core has eight programmable performance counters, which are shared among the hyperthreads that each core can host (two in the referred platform). On the Sandy Bridge microarchitecture, hence, disabling hyperthreading allows the access of all eight counters from a single hardware thread, and all floating point operations can be measured at a time. For the Westmere platform, mixed single and double precision measurements are not possible if packed operations are used, since the same counter accumulates single and double precision operations.

C. Measuring Runtime T

Counters for timing. While the regular time stamp counter of the x86 platform is commonly known, Intel has introduced more counters to measure cycles on the CPU. With the counters listed in Table II, it is possible to measure the cycle count per CPU. The time stamp counter and the unhalted reference cycles counter measure reference cycles of the socket, while the unhalted core cycles counter measures CPU cycles. For measuring performance, only the total run time is required, thus the regular time stamp counter (rdtsc) is still the right choice and it is the one used in our experiments. Nonetheless one has to be aware of the effects of frequency scaling, and why the three counters can differ.

Event	**Event Mask Mnemonic**
Flops	
Sandy / Ivy Bridge	
Scalar single	FP_COMP_OPS_EXE.SSE_FP_SCALAR_SINGLE
SSE single	FP_COMP_OPS_EXE.SSE_PACKED_SINGLE
AVX single	SIMD_FP_256.PACKED_SINGLE
Scalar double	FP_COMP_OPS_EXE.SSE_FP_SCALAR_DOUBLE
SSE double	FP_COMP_OPS_EXE.SSE_PACKED_DOUBLE
AVX double	SIMD_FP_256.PACKED_DOUBLE
Westmere	
Scalar	FP_COMP_OPS_EXE.SSE_FP_SCALAR
SSE	FP_COMP_OPS_EXE.SSE_FP_PACKED
Memory ops	
Sandy / Ivy Bridge	
Cache lines reads	UNC_CBO_CACHE_LOOKUP.I
	UNC_CBO_CACHE_LOOKUP.ANY_REQUEST_FILTER
Cache lines writes	UNC_ARB_TRK_REQUEST.EVICTIONS
Westmere-EP	
Cache lines reads	UNC_QMC_NORMAL_READS.ANY
Cache lines writes	UNC_QMC_WRITES.FULL.ANY
Sandy Bridge-EP	
Cache lines reads	UNC_IMC_NORMAL_READS.ANY
Cache lines writes	UNC_IMC_WRITES.FULL.ANY
Timers	
Core Cycles	UnHalted Core Cycles
Reference Cycles	UnHalted Reference Cycles
Time stamp counter	IA32_TIME_STAMP_COUNTER

TABLE II
COUNTERS. EVENT MASK MNEMONIC AS USED IN [11] AND [12].

Caveats. All three counters measure cycles, but the latter two can differ from the time stamp counter for the following reasons:

- The time stamp counter is a system-wide counter. Therefore, it also measures any effects of the operating system. The unhalted reference cycles counter and unhalted core cycles counter, on the other hand, are only incremented when the selected hardware thread is running.
- In the parallel scenario, the reference cycles will report how many cycles are spent per CPU, but reconstructing their sum back into wallclock time is not trivial due to partially overlapping regions.
- The unhalted core cycles counter is incremented in each core cycle. These cycles, however, might tick slower than the reference cycles in case of speed stepping, or tick faster in case of turbo boost. This would imply a shift of the peak performance bound with respect to the reference one, and hence some points might appear above the roofline.

We want to emphasize that enabling turbo boost (by default enabled on all modern Intel processors) will make any kind of performance plot subject to non-deterministic fluctuations, as the boost is controlled based on the temperature of the processor. Even between repeats of the same problem, the operational frequency of the CPU might differ. All our measurements are done with turbo boost and frequency scaling disabled in the BIOS.

D. Measuring Memory Traffic Q

Counters for memory traffic. This measurement is the most challenging for creating roofline plots, since the set of counters differ between different microarchitectures. The Intel PCM software has the highest coverage of memory traffic counters for Intel microarchitectures, but we still had to extend it to measure traffic on the desktop versions of the Sandy Bridge and Ivy Bridge microarchitecture. We list all events used across the different platforms in Table II.

Caveats. As a first attempt to measure memory traffic, the counter for LLC misses seems appropriate. However, it is not suitable for roofline plots for several reasons. First, the considered caches are all write-allocate, meaning that an LLC write miss may cause having twice the amount the traffic. Further, other events may cause data transfers like the prefetcher, page table loads, and streaming (non-temporal) memory operations. Thus, the most appropriate approach is to measure the raw traffic on the memory controller, which accounts for all the aforementioned traffic. Unfortunately, accessing the memory-related counters listed in Table II is not supported in most libraries because it is different for each microarchitecture of Intel. As mentioned in the background section, this was one of the main reasons why we chose and extended the Intel PCM software to access those counters. We measure read and write traffic separately, and add the results to obtain Q.

E. Other Issues

Dead code elimination. If the compiler can determine that the target code within the measuring strategy does not affect the program results, it will optimize it away. We mainly observed this when `target_code` was compiled together with the surrounding measurement code. To avoid it, we compile the measurement code and the target code separately. If this is not possible, we just make use of the processed data after the measurement, e.g. by printing it out.

Initialization. If the input data is not initialized after allocation, the compiler potentially optimizes for this case. The resulting program will then have less reads compared to the same program with initialized data.

Alignment. In general, one should be aware that alignment has a significant impact on the measurements. We used 64 byte (cache line) alignment for this work to overcome this shortcoming.

Asynchronous calls. Asynchronous calls might lead to a callback during the actual measurement and potentially disturb the results. As these effects are very hard to debug, we try to avoid them whenever possible.

Hardware prefetcher. In our experiments, we could see significant differences in the amount of memory transferred depending on whether the prefetcher was deactivated (via BIOS) or activated. All experiments are reported with active prefetcher as this is the common use-case.

Mnemonic name	**XSB**	**XW**	**CSB**
CPU Model	Xeon E5-2660	Xeon X5680	Core i7-3930K
Microarch.	Sandy Bridge EP	Westmere EP	Sandy Bridge E
ISA	AVX	SSE 4.2	AVX
Cores	8	6	6
Sockets	2	2	1
Frequency [GHz]	2.2	3.3	3.2
π per core [Flops/cycle]	8	4	8
β one/all cores [Bytes/cycle]	6.7/14.1	6.7/13.9	6.2/10.4
Operating System	RHEL Server 6	RHEL Server 6	Ubuntu 12.10 Windows 7

TABLE III
PROPERTIES OF THE PLATFORMS USED FOR THE EXPERIMENTS.

F. Validation

To validate our measurement of W and Q, we consider cold cache measurements of the three BLAS functions in Table I, choosing sizes for which the input data fits into the LLC. In this case, the bounds on Q and I from the table should be a close approximation of the expected result, and the value of W is known precisely. For daxpy, we choose the size ranges $n = 10^7 i, 1 \leq i \leq 6$, and for dgemv and dgemm, $n = 100i, 1 \leq i \leq 6$. To validate our experiments, we use the single-threaded code from Intel's MKL, which is highly optimized (and certainly vectorized). For the validation we also experimented with straightforward loop-based implementation, but the effects of the compiler and the used flags impact the results tremendously (to the point that it would warrant a study by itself) and therefore this data is omitted.

We run the experiments on the three computers shown in Table III. Table IV collects the results, reporting the median and the maximum ratio between the measured and the expected results. We can see that the floating point operation count is very precise for all kernels. For dgemv on the CSB platform, we measure a big relative error for Q_w, which will be explained in the experimental section.

G. Measuring π and β

The peak performance π can be estimated using a suitable microbenchmarks, for example, a vector reduction with sufficiently many accumulators. However, for our plots we use the information obtained from the manual to ensure integer results. We distinguish between single and multi-core peak performance.

Measured memory bandwidth β highly depends on the memory access pattern of the microbenchmark used to measure, and how the different prefetchers interact with the access

		W		Q_r		Q_w		Q_{r+w}	
Platform	Kernel	Med.	Max.	Med.	Max.	Med.	Max.	Med.	Max.
XSB	daxpy	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	dgemv	1.15	1.19	1.00	1.00	1.34	1.49	1.01	1.01
	dgemm	1.00	1.00	1.01	1.03	1.04	1.09	1.02	1.05
XW	daxpy	1.02	1.02	1.00	1.00	1.00	1.00	1.00	1.03
	dgemv	1.05	1.00	1.01	1.04	1.06	1.00	1.01	1.00
	dgemm	1.00	1.00	1.01	1.05	1.02	1.10	1.01	1.06
CSB	daxpy	1.00	1.00	1.04	1.06	1.02	1.03	1.03	1.05
	dgemv	1.12	1.18	1.01	1.02	1.97	2.96	1.01	1.02
	dgemm	1.01	1.01	1.23	1.39	1.16	1.40	1.21	1.40

TABLE IV
VALIDATION OF W AND Q BY COMPARING THE MEASURED VALUES (MEDIAN AND MAXIMUM) WITH THE CORRESPONDING THEORETICAL BOUNDS IN TABLE I. PLATFORM NAMES REFER TO TABLE III.

.

stream. We measure and report a best-case memory bandwidth by executing the optimized Intel MKL copy routine to transfer 1 GB of contiguously allocated data. Additionally, we measure the read-only and write-only bandwidth. Note that β is not a simple composition of the two. To obtain them, we use intrinsic instructions to only load or only store a 1 GB buffer. Single core bandwidth is usually lower than the bandwidth achieved when multiple cores are used. This can be measured by using the parallel version of the MKL routine, and by extending the read-only and write-only routines with OpenMP annotations. We also did experiments with the STREAM benchmark but in the end used the MKL routines since they achieved a higher bandwidth. A bandwidth bound for random access (meaning no spatial locality) can easily be added to the roofline plot for applications with such access patterns.

IV. BENCHMARKS

In this section we show a number of experiments that illustrate possible uses of roofline measurements. This includes comparing the operational intensity of common numerical kernels, assessing the optimality of memory-bound computations, studying successively optimized versions of the same kernel, and studying the effect of warm and cold data caches, as well as of threading. We first specify the experimental setup; then we discuss the various experiments.

Experimental setup. We ran all experiments on the three recent Intel-based platforms shown in Table III, running two different operating systems. We omitted the results obtained on an Ivy Bridge platform running OS X, as they are qualitatively similar to the CBS platform and give no new insights. We consider the following numerical kernels:

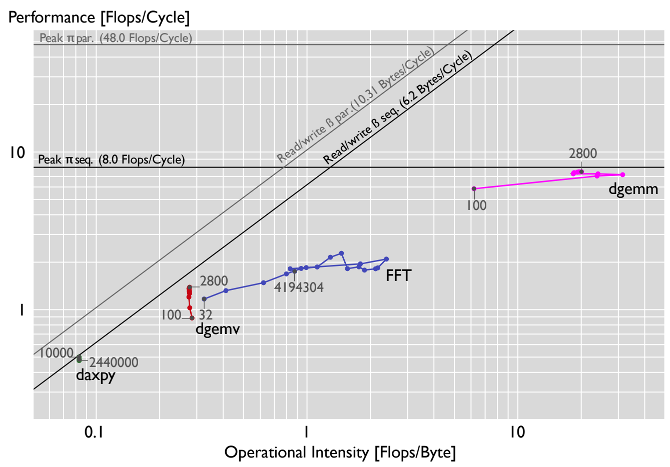
- BLAS functions from MKL v.11 [2], ATLAS v.3.10[35], and the corresponding hand-coded implementations.
- FFT libraries from Numerical Recipes (NR) [24], MKL, FFTW [9], and Spiral-generated code from the website of [26], [25].

MKL is provided as binary code, the ATLAS and FFTW libraries are compiled using gcc v.4.4 (the default), and hand-coded kernels as well as Spiral code are compiled using the Intel icc v.13 compiler with flags "-O3 -xHost". The data reported in the plots is always the median of several repetitions, and for each point, we also report the 25^{th} and 75^{th} percentile, represented as vertical or horizontal bars along the corresponding axis. Note that due to the logarithmic scale of the plot, only significant variations will be visible in the plot.

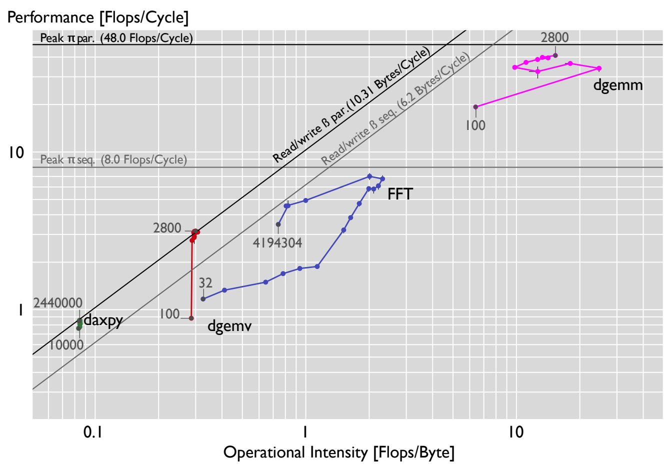
Different kernels: Single versus multithreaded. In the first experiment we run a set of kernels that are known to have different operational intensities: the BLAS 1–3 functions from Table I (on square matrices) and the FFT. Fig. 2(a) shows the performance of the set of kernels when executed sequentially on CSB using Windows. The set of input sizes n is chosen to use different levels of the memory hierarchy. For daxpy we use $n = 10000 + 30000i^2$, ($0 \leq i < 10$), for dgemv and dgemm $n = 100 + 300i$ ($0 \leq i < 10$), and for FFT, $n = 2^k$ ($5 \leq k < 23$). Daxpy and dgemv are memory-bound as expected, reaching about 95% and 90% of the peak $P = \beta I$. Dgemm is compute-bound for all sizes reaching also about 95% of the peak π. Note that daxpy hits very precisely the $I = 0.083$ from Table I, and dgemv is close to the expected $I = 0.25$. The operational intensity of FFT is between dgemv and dgemm, which is expected since the optimal $I(n)$ is known to be $\Theta(\log(\gamma))$ [15] versus $\Theta(\sqrt{\gamma})$ for matrix multiplication (using $2n^3$ operations). Note that the FFT is plotted with measured flops and not with pseudo-flops (which assume an op count of $5n\log_2(n)$, an overestimation), as commonly done.

Fig. 2(b) shows the same plot, but measured with the multithreaded versions of the kernels. We measure an increase in the memory bandwidth from 6.2 to 10.31 Bytes/cycle, which allows for a 69% improvement in the performance of daxpy and dgemv. Due to a shift of the ridge point from 1.3 to 4.6 flops/byte, the FFT kernel becomes clearly memory-bound. The effect of multithreading can be seen for sizes starting from $n = 2^{11}$. In both the parallel and the sequential case, after $n = 2^{17}$, the operational intensity starts decreasing. Using six cores, the dgemm kernel exhibits up to 5.4x speedup over its sequential version and slightly lower I, likely due to a parallelization overhead.

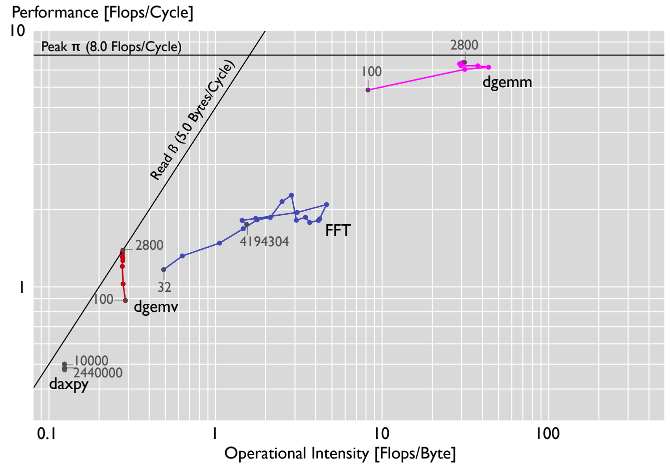
Different kernels: Read- and write-only roofline plots. If a computation is memory-bound (e.g., daxpy and dgemv in Figs. 2(a) and (b)), one question is whether the read or the write bandwidth is the bottleneck. This can be answered through modified roofline plots that use I_r and I_w instead of I as shown in Figs. 2(c) and (d) for a cold cache. For both daxpy and FFT, $Q_r(n), Q_w(n) = \Omega(n)$, while for dgemm, $Q_r(n), Q_w(n) = \Omega(n^2)$. For dgemv, however, $Q_r(n) = \Omega(n^2)$ while $Q_w(n) = \Omega(n)$. Given that $W(n) = \Omega(n^2)$, dgemv appears memory-bound in the read-only plot and compute-bound in the write-only plot. The read-only bottleneck also explains the gap between the performance of dgemv and the roofline in Fig. 2(a), as β_r becomes a limiting



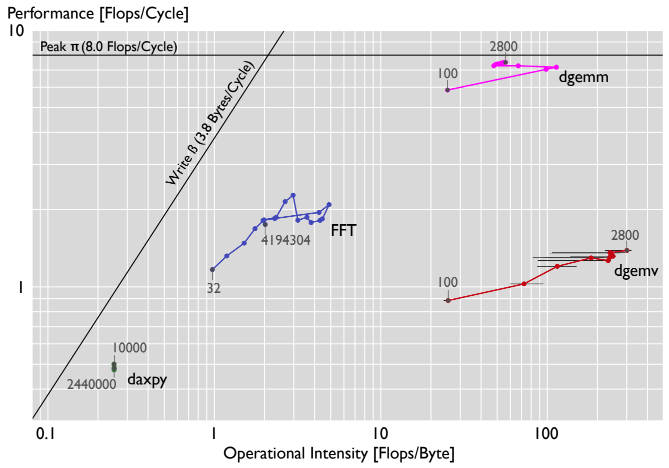
(a) Sequential code.



(b) Parallel code.



(c) Read-only bandwidth.



(d) Write-only bandwidth.

Fig. 2. Roofline plot for the Intel MKL BLAS 1–3 and FFT kernels on CSB, with cold cache.

factor: $P = P_r = I_r \beta_r$. Moreover, most of the time is spent in computing W flops and transferring Q_r bytes. This results in a larger error in the measurement of Q_w and, consequently, in the computation of I_w.

Different kernels: Cold versus warm cache. Warm cache conditions remove the operational intensity limits obtained from counting compulsory misses. For problem sizes that completely fit into cache, $I = \infty$. In reality, Fig. 3 shows that for small sizes, I becomes very large and a high measurement error for I is introduced (horizontal lines). This is due to the presence of memory traffic measured by the uncore (per socket) memory counters. As the sizes increase, the code starts to behave as in the cold cache scenario above.

Study of code optimizations: Dgemm. Fig. 4 shows the roofline plot for the dgemm kernel at different levels of optimizations. In addition to ATLAS and MKL we consider a hand-coded standard triple loop and a six-fold loop obtained through blocking with block size 50.

We first discuss the triple loop. For matrix sizes that fit into cache, $I(n) \approx n/16$, and indeed we measure $I(100) = 6.1$

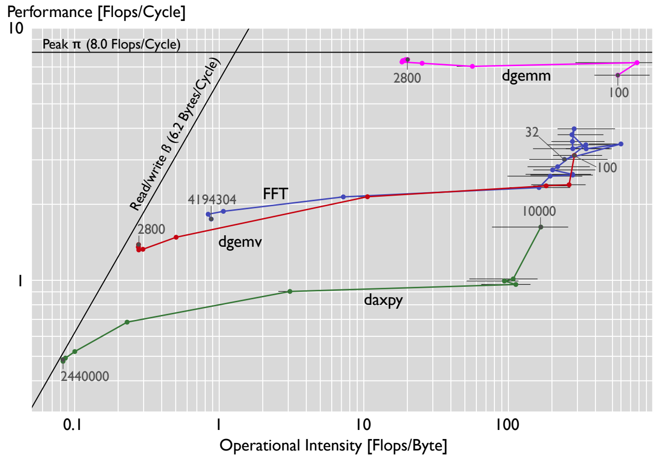


Fig. 3. Roofline plot for the Intel MKL BLAS 1–3 and FFT kernels on CSB: sequential code, warm data cache.

flops/byte. When the data do not fit in cache anymore (in our experiment for $n \geq 1000$), the value of I starts decreasing due

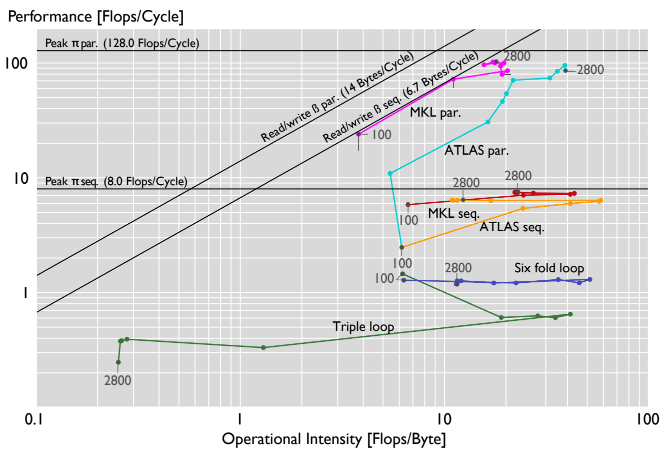


Fig. 4. Roofline plot for matrix-matrix multiply kernels on CSB with cold data cache, sequential and parallel versions.

to larger values of Q, which slowly becomes cubic in n, as we need to reload repeatedly all elements of the second matrix [38]. For large matrices, the operational intensity can then be estimated as $I(n) = \frac{2n^3}{8n^3} = 0.25$ flops/byte as confirmed in the experiment.

The six-fold loop implements an L1-blocked version of the kernel using squared blocks of size $N_B^2 = 50^2$ doubles. By improving locality, the operational intensity of large problems becomes $I = O(N_B)$, as $Q(n) \geq \frac{8n^3}{N_B}$. The MKL and ATLAS kernels provide in addition improved ILP, SIMD vectorization, and multithread parallelism. ATLAS improves locality by autotuning the block size when installing the library. For the multithreaded ATLAS, the first performance point lays below $\pi = 8$ flops/cycle (peak performance for sequential code with AVX instructions). This is most likely due to the library's choice of using one thread for smaller problem sizes.

This example demonstrates to what extent roofline plots can help with performance optimization. Fig. 4 shows that the nave triple loop implementation for large sizes becomes memory bound; thus, a first optimization should aim at increasing the operational intensity of the kernel. Blocking the computation (six-fold loop) achieves this. Further optimizations (as done in MKL) are not explained by the roofline plot.

Study of code optimizations: FFT. We compare various FFT kernels in Fig. 5. As in the previous plots, we use sizes $n = 2^k$ $(5 \leq k < 23)$, except for the available Spiral-generated kernels, which only support sizes up to 2^{12}. Additionally to the regular warm and cold cache roofline plots, we show plots that use the pseudo-flop count of $5n\log_2(n)$ [32] instead of the measured ones. We note that both types of plots have inherent problems: in the plot based on measured flops, the performance is not proportional to runtime since W differs between codes. This is most explicit in the NR code as explained below. Using pseudo-flops resolves that problem but overestimates the actual W and thus distorts the relation to the bounds in the plot.

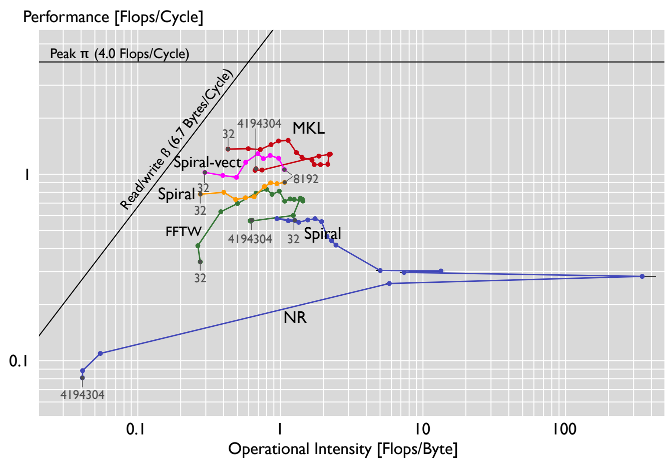
Fig. 5(a) shows the measured results using cold cache. The order of the kernels, in terms of performance, is roughly as expected, with MKL and one version of Spiral using the SSE instruction set, and the rest of the implementations being unvectorized. The only surprise is the performance of the straightforward NR code. To explain this, we use Fig. 5(c), which uses pseudo flops. We see that the NR curve moves to the bottom left, while the rest of the curves move slightly to the top right. The reason is that the NR code calculates the twiddle constants at runtime for all sizes, resulting in an increased flop count and thus increased performance and operational intensity. The rest of the kernels move in the opposite direction, as the used pseudo-flops are overestimated. When looking at warm cache measurements in Fig. 5(b), the performance gap becomes bigger as the kernels are now mainly compute-bound. The NR kernel is clearly separated from the rest due to poor locality, and when looking at the pseudo-flop results in Fig. 5(d), we can see that the additional computation has a more dominant effect under this bound.

V. Related Work

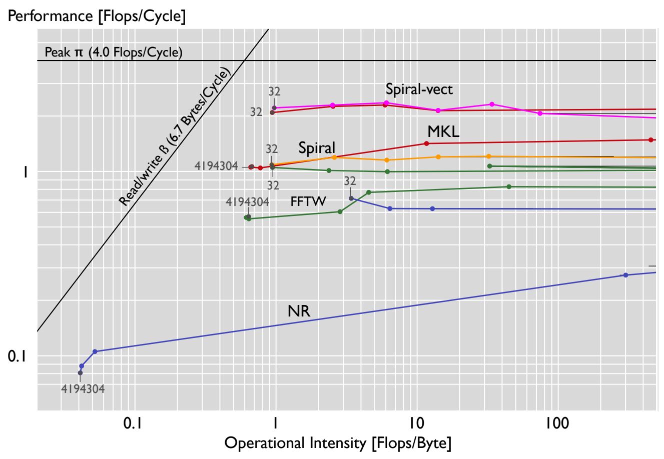
The paper introducing the roofline model [37] shows roofline plots for four platforms and four numerical functions, each with a fixed input size. Besides that we are not aware of any prior work that shows roofline plots using measured data. However, there are some lines of work that are closely related to this paper as discussed next.

Accessing hardware performance counters. There are various interfaces for accessing performance counters. PAPI [20] is one of the most popular because it is available across different microarchitectures, supports various operating systems, and provides both routines for low-level access to hardware performance counters and high-level routines that facilitate its usage. Other well-known tools for performance counters monitoring are perf_events [28] (the standard interface used in Linux), OProfile [22], and the libraries libpfm and libperfctr, which are both based on the perfmon2 [5] and PerfCtr [23] extensions of the Linux kernel, respectively. All of these provide a low-level interface for accessing the performance counters but are not available in Windows or OS X. We use PCM as it is available across OSes and, as of now the only one, that provides access to uncore events. The limitation is the restriction to recent Intel platforms.

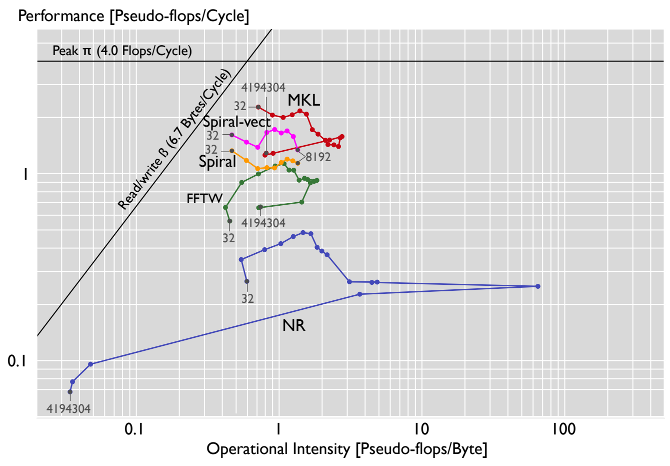
Performance analysis tools. There are a number of tools that combine *measurement* and *analysis* of program performance. VTune [3], for example, is an integrated tool that automatically profiles the execution of an application on an Intel platform using performance counters, and reports a detailed breakdown of execution cycles. Similarly, the HPC TOOLkit [6] performance tools, use statistical sampling of timers and hardware performance counters to collect accurate measurements of a program's behavior. GOoDA [10] is another recent tool that combines a hardware performance counter access infrastructure with a performance data analyzer and a web based visualizer. Our work falls into this category and could possibly be integrated into any of these tools.



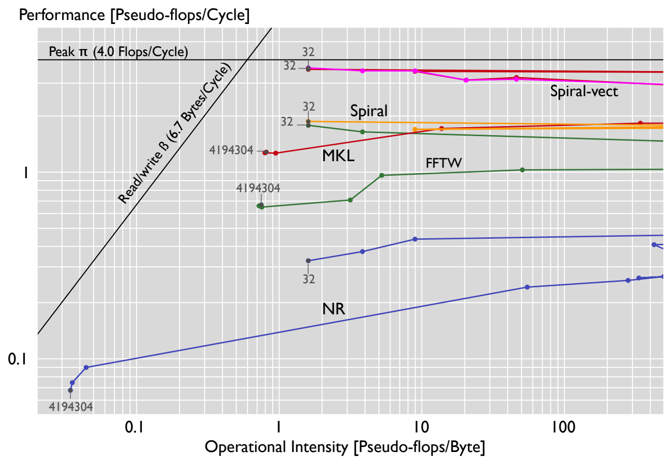
(a) Cold cache.



(b) Warm cache.



(c) Cold cache with pseudo-flops.



(d) Warm cache with pseudo-flops.

Fig. 5. Roofline plots for several implementations of the FFT.

Measuring computing systems performance. Performance measurements range from simple routines that use clock wall time to measure execution time, to more sophisticated mechanisms that implement different timing methodologies to achieve accurate and reliable results [34]. The most common sources of errors in the measurement process are the additional software instructions executed to access the counters (also known as the observer effect) and perturbations due to measurement context bias [21]. Inaccuracies in the measurements also depend on other factors such as the duration of the execution, the enabling or disabling of certain features such as frequency scaling. In turn, these factors vary depending on the library used for accessing the counters and the underlying microarchitecture [39]. A number of papers describe techniques to achieve accurate results using hardware performance counters [19].

Measuring performance also requires an understanding of the properties of the underlying hardware architecture. This is commonly done by carefully designing microbenchmarks that stress the target hardware features to be tested. The STREAM Benchmark [30] is widely used for measuring memory bandwidth. [29] proposes different experiments for measuring relevant parameters of the underlying memory subsystem, such as cache size and cache block size.

Users of the roofline model. The roofline model is used in a number of scientific papers that apply it to understand performance and guide software optimizations of applications including gridparticle interpolation [27], software correlators used in radio astronomy [33], and stencil computations [31]. In most of the cases, the operational intensity and the upper and lower bounds of the memory traffic are manually derived.

The roofline model has also been used to compare the performance of applications in three of the largest supercomputers, an IBM Blue Gene/P, a Sun-Infiniband cluster, and a Cray XT4, each of them with different theoretical peak performance and memory bandwidth [7]. Finally, it has also been applied to other platforms such as GPUs [16], and recently it has been extended to include bounds on performance due to energy limitations [36]. In contrast to our work, all these uses of the roofline model are based on paper calculations rather than on measurements.

VI. Summary and Conclusion

The roofline model considered in the paper can help with identifying effects and bottlenecks due to memory traffic. For example in the paper we describe the case of dgemv. Using read- and write-only roofline plots, the reason of the performance bottleneck appears visually, and can be intuitively explained by the (read) bandwidth limitation. Furthermore back-of-the-envelope calculations become next to impossible for more complex algorithms, such as a FFT or dgemm, since not only compulsory cache misses occur. In these cases, a roofline plot can only be created with measurements.

The first goal of the paper was to show that the roofline model can be used with measurements rather than back-of-the-envelope calculations. What at first glance seemed like a straightforward task, was surprisingly difficult due to several pitfalls arising from low level system details. Examples of such pitfalls include interactions with the operating system, default enabled features, like turbo boost and the hardware prefetcher, and different compiler optimizations.

The second goal was to show that with measurements, roofline plots can be a valuable tool in performance analysis alongside ordinary performance plots. The main reason is that the two key resource constraints, peak performance and memory bandwidth, are both available as upper bounds, and the notions of memory and compute bound are made precise. We focused on floating point computations but the model and the entire paper can easily be instantiated for integer computations. The roofline plots can help with performance optimization and are certainly of educational value. One of the authors has been using (measured) roofline plots regularly in his class. The source code accompanying this paper is available at [4].

Acknowledgments

The authors would like to thank Zoltán Majó for many helpful discussions.

References

[1] BLAS (Basic Linear Algebra Subprograms). http://www.netlib.org/blas/.

[2] Intel®Math Kernel Library (Intel®MKL) 11.0. http://software.intel.com/en-us/intel-mkl.

[3] Intel®VTune$^{\mathrm{TM}}$Amplifier XE 2013.

[4] Roofline code. http://www.spiral.net/software/roofline.html.

[5] S. Eranian. The perfmon2 project. http://perfmon2.sourceforge.net/.

[6] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLkit: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22:685–701, 2010.

[7] A. Bhatel, L. Wesolowski, E. Bohm, E. Solomonik, and L. V. Kal. Understanding Application Performance via Micro-benchmarks on Three Large Supercomputers: Intrepid, Ranger and Jaguar. *International Journal of High Performance Computing Applications*, 24(4):411–427, 2010.

[8] K. Czechowski, C. Battaglino, C. McClanahan, A. Chandramowlishwaran, and R. Vuduc. Balance principles for algorithm-architecture co-design. In *USENIX Conference on Hot Topic in Parallelism*, HotPar'11, pages 9–9, 2011.

[9] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2), 2005. www.fftw.org.

[10] GOoDA - PMU Event Analysis Package. http://code.google.com/p/gooda/.

[11] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. 2013.

[12] Intel Corporation. *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon processor 5500 series*. 2013.

[13] Intel®Performance Counter Monitor. http://software.intel.com/en-us/articles/intel-performance-counter-monitor/.

[14] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput.*, 64:1017–1026, 2004.

[15] H. Jia-Wei and H. T. Kung. I/O Complexity: The red-blue pebble game. In *Symposium on Theory of Computing (STOC)*, pages 326–333, 1981.

[16] K.-H. Kim, K. Kim, and Q.-H. Park. Performance analysis and optimization of three-dimensional FDTD on GPU using roofline model. *Computer Physics Communications*, 182(6):1201 – 1207, 2011.

[17] H. T. Kung. Memory requirements for balanced computer architectures. In *International Symposium on Computer Architecture (ISCA)*, pages 49–54, 1986.

[18] D. J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. 2005.

[19] W. Mathur and J. Cook. Toward accurate performance evaluation using hardware counters. In *ITEA Modeling and Simulation Workshop*, 2003.

[20] P. J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A Portable Interface to Hardware Performance Counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[21] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. We have it easy, but do we have it right? In *NSF Next Generation Systems Workshop*, pages 1–5, 2008.

[22] OProfile - A system Profiler for Linux. http://oprofile.sourceforge.net.

[23] PerfCtr. http://aspectr.sourceforge.net/perfctr/.

[24] W. H. Press, B. P. Flannery, T. S. A., and V. W. T. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1992.

[25] M. Püschel, F. Franchetti, and Y. Voronenko. *Encyclopedia of Parallel Computing*, chapter Spiral. Springer, 2011.

[26] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. 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, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005. www.spiral.net.

[27] D. Rossinelli, C. Conti, and P. Koumoutsakos. Mesh-particle interpolations on graphics processing units and multicore central processing units. *Philos Transact A Math Phys Eng Sci*, 369:2164–75, 2011.

[28] S. Eranian, Overview of the perf_event API. http://cscads.rice.edu/workshops/summer09/slides/performance-tools/cscads09-eranian.pdf.

[29] A. J. Smith and R. H. Saavedra. Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes. *IEEE Trans. Comput.*, 44(10):1223–1235, 1995.

[30] STREAM Benchmark. http://www.cs.virginia.edu/stream/.

[31] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *IEEE International Symposium on Parallel&Distributed Processing (IPDPS)*, pages 1–12, 2009.

[32] C. Van Loan. *Computational Frameworks for the Fast Fourier Transform*. 1992.

[33] R. V. van Nieuwpoort and J. W. Romein. Using many-core hardware to correlate radio astronomy signals. In *International Conference on Supercomputing*, pages 440–449, 2009.

[34] R. C. Whaley and A. M. Castaldo. Achieving accurate and context-sensitive timing for code optimization. *Softw. Pract. Exper.*, 38(15):1621–1642, Dec. 2008.

[35] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27:3–35, 2001.

[36] J. Whan Choi and R. Vuduc. A roofline model of energy. Technical report, Georgia Institute of Technology, School of Computational Science and Engineering, 2012.

[37] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52:65–76, 2009.

[38] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance blas? *Proceedings of the IEEE*, 93(2):358–386, Feb. 2005.

[39] D. Zaparanuks, M. Jovic, and M. Hauswirth. Accuracy of performance counter measurements. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 23 –32, 2009.