ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

HPC Lab for CSE

FS 2024
Dr. R. Käppeli,
Dr. A. Schell, M. R. Herde,
C. Humbel, A. Mazzoleni,
N. A. Siegenheim

Project 1b

Due date: 11 March 2024, 11:59pm (midnight)

The first part of this project (a) will introduce you to using the Euler cluster and collecting some performance characteristics. The second part of the project (b) will be about the optimization of general matrix multiplication. Both project parts will be serial. Parallel programming will be the subject of the forthcoming projects.

3 Auto-vectorization [10 points]

Read the Chapter "Automatic Vectorization" of the *Intel oneAPI DPC++/C++ Compiler Developer Guide and Reference* [5] and answer briefly the following questions:

1. Why is it important for data structures to be aligned?
2. What are some obstacles that can prevent automatic vectorization by the compiler?
3. Is there a way to help the compiler to vectorize and how?
4. Which loop optimizations are performed by the compiler to vectorize and pipeline loops?
5. What can be done if automatic vectorization by the compiler fails or is sub-optimal?

If you wish, you can play around with (a slightly outdated version of) the Intel compiler by loading the following module:

```
[user@eu-login-39 ~]$ module load intel
[user@eu-login-39 ~]$ icx --version
Intel(R) oneAPI DPC++/C++ Compiler 2022.0.0 (2022.0.0.20211123)
Target: x86_64-unknown-linux-gnu
Thread model: posix
InstalledDir:
↪  /cluster/apps/nss/intel/oneapi/2022.1.2/compiler/2022.0.2/linux/bin-llvm
```

4 Matrix multiplication optimization[1] [30 points]

4.1 Motivation

Linear algebra is a cornerstone operation in computational science and holds a particularly pivotal role in HPC. In particular, we are going to focus on matrix multiplication. This mathematical operation underpins a vast array of scientific and engineering disciplines, facilitating the modeling of complex systems, simulations of physical phenomena, and the processing of large datasets. In HPC environments, the efficiency and scalability of matrix multiplication directly influence the performance of algorithms (used in computational sciences, engineering, machine learning, and beyond). This functionality is offered by BLAS (Basic Linear Algebra Subprograms), a specification encompassing a suite of low-level routines designed for executing common linear algebra operations efficiently.

Each manufacturer typically supplies a BLAS library (e.g., Intel's MKL library, AMD's AOCL, ...), carefully hand-optimized for its machines. It includes a large number of routines, not just matrix multiplication, although matrix multiplication is one of the most important, because it is used as a benchmark

[1] This task is based on a previous HPC Lab for CSE by Prof. Olaf Schenk from the Institute of Computing at the Faculty of Informatics at Universita della Svizzera Italiana (USI), which was itself originally based on a tutorial from Prof. Katherine A. Yelick from the Computer Science Department at the University of Berkeley (http://www.cs.berkeley.edu/~yelick/).

to compare the speed of computers (e.g., LINPACK Benchmark[2] used in the Top 500[3] list of the fastest supercomputers). In addition to manufacturer supplied libraries, there are also a number of open source projects. For instance, there are the ATLAS[4], GotoBLAS[5], OpenBLAS[6], and BLIS[7] (the 2023 recipient of the James H. Wilkinson Prize for Numerical Software). The list is not meant to be exhaustive.

The performance difference between naive and library implementations can be drastic. Figure 1 shows the performance in GFlops/s of n-by-n matrix multiplication (on a single core of an AMD EPYC 7763 CPU) as a function of matrix size n. It compares the highly optimized Intel MKL and OpenBLAS libraries with a naive implementation. As apparent from the figure, the naive implementation is sub-optimal (to say the least). As it turns out, matrix multiplication, commonly referred to as `DGEMM` (Double precision GEneral Matrix Multiplication), has a growing operational intensity (with the matrix size n). Therefore it can be (asymptotically) pushed into the compute bound regime (e.g., roofline model). However, achieving this requires specialized, hardware-dependent optimization techniques. The goal of this project task is to familiarize you with some basic techniques and offer pointers to literature for more advanced techniques.

However, achieving this requires very specialized, hardware dependent, optimization techniques. It is the goal of this project task to familiarize you with some basic techniques and provide you some pointers to the literature for more advanced techniques in the next paragraph.

The anatomy of a HPC matrix multiplication is described in the article [2]. The BLIS framework is described by Zee et al. [10] and Low et al. [6] propose an analytical performance model to determine certain parameters. The latter article gives a nice overview of the anatomy of a fast `DGEMM`. The recent article by Alaejos et al. [1] provides templates to write micro-kernels using vector intrinsics. Last but not least, we highly recommend the very comprehensive online course "LAFF-On Programming for High Performance" [9]. On a less scientific level, you might also find the newspaper article [7] interesting.

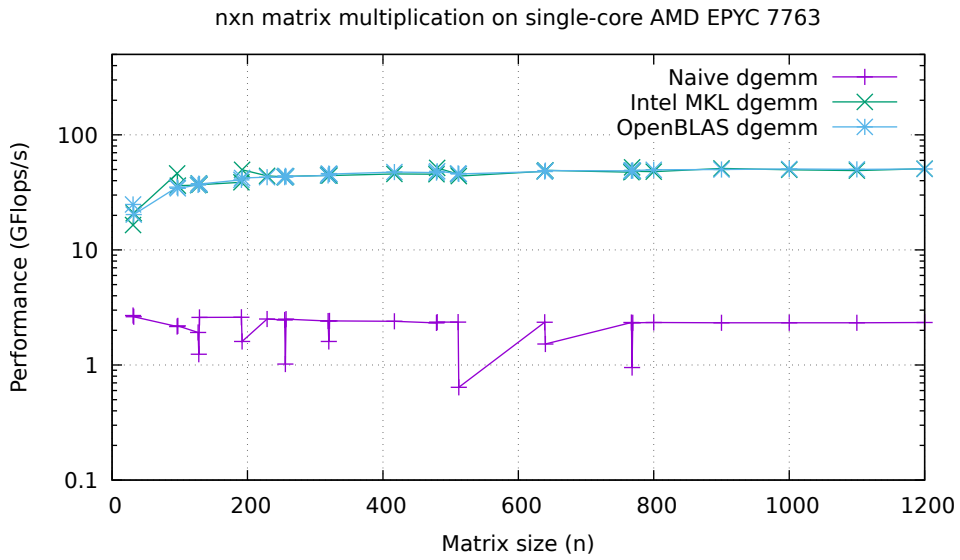

Figure 1: Serial performance of matrix multiplication on the AMD EPYC 7763.

4.2 Importance of data access optimization

Among the various factors that can limit performance in HPC, data access stands out as the most critical. This is due to the inherent imbalance in modern CPUs, where there is a significant discrepancy between their theoretical peak performance and the available memory bandwidth. Therefore, any attempt at optimization should initially focus on minimizing data traffic, or, if that proves to be impractical, strive to make data transfer as efficient as possible. If you haven't already, we strongly recommend to read Chapter 3 of the book [3].

[2] https://www.top500.org/project/linpack/
[3] https://www.top500.org/
[4] https://math-atlas.sourceforge.net/
[5] https://en.wikipedia.org/wiki/GotoBLAS
[6] https://www.openblas.net/
[7] https://github.com/flame/blis

4.2.1 Modeling memory hierarchies

Real memory hierarchies are very complicated, so modeling them carefully enough to predict the performance of an algorithm is hard. To make progress, we will use the following simplified model:

(i) There are only **two levels** in the memory hierarchy, **fast** (e.g., registers, caches) and **slow** (e.g., main memory).

(ii) All the input data starts in the slow level, and the output must eventually be written back to the slow level.

(iii) The size of the fast memory is M, which is large but much smaller than the slow memory. In particular, the input of large problems will not fit in the fast memory simultaneously.

(iv) The programmer explicitly controls which data moves between the two memory levels and when. **Remark**: It might seem incorrect to suggest that programmers can control data movement between main memory and cache, as this process is typically automated by the hardware. However, we know how the hardware works: it moves data to the cache precisely when the user first tries to load it into a register to perform arithmetic, and puts it back in main memory when the cache is too full to get the next requested word. This means that we can write programs as if we controlled the cache explicitly by doing arithmetic operations in different orders. Thus, two programs that do the same arithmetic in different orders may run at very different speeds, because one reduces data movement between the main memory and cache.

(v) Arithmetic (and logic) can only be done on data residing in the fast memory and data movement and computation cannot overlap. Each arithmetic operation takes time t_a. Moving a word of data from one memory to the other takes time $t_m \gg t_a$. Hence, if a program performs m memory moves and a arithmetic operations, the total time it takes is $T = a \cdot t_a + m \cdot t_m$, where it may be that $m \cdot t_m \gg a \cdot t_a$.

With this model, let us get a **lower bound** on the speed of any algorithm for a problem that has m_i inputs and m_o outputs, and does a arithmetic operations. The only difference between algorithms that we permit is the order in which the arithmetic operations are executed. According to our model, the run-time taken will be at least

$$T = a \cdot t_a + (m_i + m_o) \cdot t_m,$$

no matter which clever order we do the arithmetic in.

For example, suppose the problem is to take two input arrays of n numbers each, $a[i]$ and $b[i]$ for $i = 1, .., n$, and produce two output arrays s and p, where $s[i] = a[i] + b[i]$ and $p[i] = a[i] \cdot b[i]$. Then the number of arithmetic operations is $a = 2n$, and the number of memory moves is $m_i = 2n$ and $m_o = 2n$. Thus, a lower bound on the run-time for any algorithm for this problem is $T = 2nt_a + 4nt_m$. Let us look at two different algorithms for this problem, and analyze which will be faster. The two algorithms are listed in Table 1. According to our model, we assume that a and b are initially in the slow memory, and $M \ll n$, so a and b cannot fit in the fast memory, and we have indicated when loads from and stores to slow memory occur. The point is that Algorithm 1 loads a and b twice, whereas Algorithm 2 only loads

Algorithm 1	Algorithm 2

`for i = 1, n`
`  Load a[i], b[i] into fast memory`
`  s[i] = a[i] + b[i]`
`  Store s[i] into slow memory`
`end for`
 	
`for i = 1, n`
`  Load a[i], b[i] into fast memory`
`  p[i] = a[i] * b[i]`
`  Store p[i] into slow memory`
`end for`
	`for i = 1, n`
	`  Load a[i], b[i] into fast memory`
	`  s[i] = a[i] + b[i]`
	`  Store s[i] into slow memory`
	`  p[i] = a[i] * b[i]`
	`  Store p[i] into slow memory`
	`end for`

Table 1: Two algorithms to compute the sum and product of the arrays.

them once (remember, there is not enough room in the fast memory to keep all the $a[i]$ and $b[i]$ for a second pass to compute $p[i]$). As per our simple model, Algorithm 1 takes run-time $T_1 = 2nt_a + 6nt_m$, whereas Algorithm 2 takes only $T_2 = 2nt_a + 4nt_m$, the minimum possible. Thus, for $t_a \ll t_m$, Algorithm

1 takes 50% longer to run than Algorithm 2. In the case of matrix multiplication, we will observe an even more dramatic difference.

4.2.2 Minimizing slow memory access in matrix multiplication

We will only consider different ways to implement n-by-n matrix multiplication $C = C + A \cdot B$ using $2n^3$ operations (i.e., we will not consider Strassen's method [8] and variants thereof). Thus, the only difference between algorithms will depend on the number of loads and stores to the slow memory. We also assume that the slow memory is large enough to contain our three n-by-n matrices A, B and C, but the fast memory is too small for this. Otherwise, if the fast memory were large enough to contain A, B, and C simultaneously, then our algorithm would simply be:

```
1  Load A, B and C from slow into fast memory
2
3  method dgemm_fastmem(A, B, C)
4
5    Compute C = C + A · B entirely in fast memory
6
7  end method dgemm_fastmem
8
9  Store the result C back to slow memory
```

The number of slow memory accesses for this algorithm is $4n^2$ ($m_i = 3n^2$ loads of A, B, and C into fast memory, and $m_o = n^2$ stores of C to slow memory), yielding a run-time of $T = 2n^3 t_a + 4n^2 t_m$. Clearly, no algorithm doing $2n^3$ arithmetic operations can be faster. At the extreme where the fast memory is very small ($M = 1$), then there will be at least 1 memory reference per operand for each arithmetic operation involving entries of A and B, for a run-time of at least $T = 2n^3 t_a + (2n^3 + 2n^2) t_m$.

So, when the size of fast memory M satisfies $1 \ll M \ll 3n^2$, what is the fastest algorithm?

This is the practical question for large matrices, since real caches have thousands of entries. As we just saw, the worst case run-time is $2n^3 t_a + (2n^3 + 2n^2) t_m$, and the best we can hope for is $2n^3 t_a + 4n^2 t_m$, which would be almost $n/2$ times faster for $t_m \gg t_a$.

We begin by analyzing the simplest matrix multiplication algorithm, which we repeat below, including descriptions of when data moves between the slow and the fast memories. Remember that A, B, and C all start in the slow memory, and that the result C must be finally stored in the slow memory.

```
1  Naive matrix multiplication C = C + A · B
2
3  method dgemm_naive(A, B, C)
4
5    for i = 1, n
6      for j = 1, n
7        Load c_{i,j} into fast memory
8        for k = 1, n
9          Load a_{i,k} into fast memory
10         Load b_{k,j} into fast memory
11         c_{i,j} = c_{i,j} + a_{i,k} * b_{k,j}
12       end for
13       Store c_{i,j} into slow memory
14     end for
15   end for
16
17 end method dgemm_naive
```

Let m_{naive} denote the number of **slow memory references** in this naive algorithm. Then

$$
\begin{aligned}
m_{\text{naive}} &= n^3 && \dots \text{ for loading each entry of A } n \text{ times} \\
&+ n^3 && \dots \text{ for loading each entry of B } n \text{ times} \\
&+ 2n^2 && \dots \text{ for loading and storing each entry of C once} \\
&= 2n^3 + 2n^2,
\end{aligned}
$$

or about as many slow memory references as arithmetic operations. Thus, the run-time is $T_{\text{naive}} = 2n^3 t_a + (2n^3 + 2n^2)t_m$, the worst possible.

Let us analyse an improved algorithm called **blocked matrix multiplication** (sometimes it is called **tiled** or **panelled** instead of **blocked**). The n-by-n matrix A is divided into smaller s-by-s sub-matrices or blocks A_{ij}, where s is a parameter called the block size that we will specify later. We assume s divides n for simplicity. Matrices B and C are similarly partitioned. Then, we can think of A as an n/s-by-n/s block matrix, where each entry $A_{i,j}$ is an s-by-s block. The inner loop $C_{i,j} = C_{i,j} + A_{i,k} \cdot B_{k,j}$ now runs for $k = 1$ to s, and represents an s-by-s matrix multiplication and addition. The algorithm becomes:

```
1  Blocked matrix multiplication C = C + A·B
2
3  method dgemm_blocked(A, B, C)
4
5    for i = 1, n/s
6      for j = 1, n/s
7        Load C_{i,j} block into fast memory
8        for k = 1, n/s
9          Load A_{i,k} block into fast memory
10         Load B_{k,j} block into fast memory
11         dgemm_fastmem(A_{i,k}, B_{k,j}, C_{i,j})
12       end for
13       Store C_{i,j} into slow memory
14     end for
15   end for
16
17 en method dgemm_blocked
```

The inner loop `dgemm_fastmem`$(A_{i,k}, B_{k,j}, C_{i,j})$ has all its data residing in the fast memory, and so causes no slow memory traffic at all. Redoing the count of slow memory references yields

$$\begin{aligned} m_{\text{blocked}} = m_{\text{blocked}}(s) &= (n/s)^3 \cdot s^2 && \dots \text{ for loading each block } A_{i,k} \ (n/s)^3 \text{ times} \\ &+ (n/s)^3 \cdot s^2 && \dots \text{ for loading each block } B_{k,j} \ (n/s)^3 \text{ times} \\ &+ 2(n/s)^2 \cdot s^2 && \dots \text{ for loading and storing each block } C_{i,j} \text{ once} \\ &= 2n^3/s + 2n^2. \end{aligned}$$

Comparing $m_{\text{naive}} = 2n^3 + 2n^2$ and $m_{\text{blocked}} = 2n^3/s + 2n^2$, it is obvious that we want to pick s as large as possible to make $m_{\text{blocked}}(s)$ as small as possible. But how big can we pick s? The largest possible value is obviously $s = n$, which corresponds to loading all of A, B and C into the fast memory, which we cannot do. So, s depends on the size M of the fast memory, and the constraint it must satisfy is that the three s-by-s blocks $A_{i,k}, B_{k,j}$, and $C_{i,j}$ must simultaneously fit in the fast memory, which implies $3s^2 \leq M \implies s \leq \sqrt{M/3}$. Therefore, the largest value $s = \sqrt{M/3}$ yields

$$m_{\text{blocked}}(\sqrt{M/3}) = \sqrt{12}\frac{n^3}{\sqrt{M}} + 2n^2.$$

In other words, for large matrices (large n) we decrease the number of slow memory references, the most expensive operation, by a factor $\mathcal{O}(M)$. This is attractive, because it says that cache (fast memory) helps, and the larger the cache the better.

In summary, the running time for this algorithm is

$$T_{\text{blocked}} = 2n^3 \cdot t_a + (\sqrt{12}\frac{n^3}{\sqrt{M}} \cdot t_m).$$

There is a theorem, which we will not prove, that says that up to constant factors, we cannot do fewer slow memory references than this (while doing the usual $2n^3$ arithmetic operations):

Theorem (Hong + Kung, 1981, 13th Symposium on the Theory of Computing [4]): Any implementation of matrix multiplication using $2n^3$ arithmetic operations performs at least $\mathcal{O}(n^3/\sqrt{M})$ slow memory references.

In practice, this technique is very important to get matrix multiplication to run as fast as possible. But careful attention must also be paid to other details of the instruction set, arithmetic units, and so on. If there are more levels of memory hierarchy (two levels of cache), then one might use this technique recursively, dividing s-by-s blocks into yet smaller blocks to exploit the next level of memory hierarchy. However, this goes beyond the scope of this project and course.

4.3 Optimizing square matrix multiplication

Your task is now to write a blocked matrix multiplication (as outlined above) and optimize it for a single core of the AMD EPYC 7763 on Euler.

Skeleton code

Download the skeleton code provided on the Moodle page. It contains the following:

- `dgemm-naive.c` – For illustrative purposes, a naive implementation of matrix multiplication using three nested loops.
- `dgemm-blas.c` – A wrapper which calls an optimized BLAS implementation of matrix multiplication (OpenBLAS).
- `dgemm-blocked.c` – A skeleton for a blocked implementation of matrix multiplication. It is your task to implement and optimize the `square_dgemm` function in this file.
- `benchmark.c` – A driver program that generates matrices of a number of different sizes and benchmarks the performance. It outputs the performance in GFlops/s and in a percentage of theoretical peak performance attained. You should not need to modify this file, except perhaps to change the peak performance constant `MAX_SPEED` if you wish to test on another computer.
- `Makefile` – A simple makefile to build the executables for the benchmarking of the naive, optimized BLAS and your blocked implementation.
- `run_matrixmult.sh` – A job script that executes all three executables and produces log files (`*.data`) that contain the performance logs. It also plots the data in the performance logs and produces `timing.pdf` showing the results.

Familiarize yourself with the code and the used conventions. In particular, note that we use the *column major order* storage scheme to conform with BLAS' Fortran origins.

Running the code

The skeleton code should run out of the box and a file listing should look as:

```
[user@eu-login-39]$ ls
benchmark.c   dgemm-blocked.c  Makefile           timing.gp
dgemm-blas.c  dgemm-naive.c    run_matrixmult.sh
```

We will use the GNU Compiler Collection and the OpenBLAS implementation, which can be loaded on Euler as follows:

```
[user@eu-login-39]$ module load gcc openblas
```

Building the code

```
[user@eu-login-39]$ make
```

generates the three executables `benchmark-naive`, `benchmark-blas` and `benchmark-blocked`. We submit to the batch system with

```
[user@eu-login-39]$ sbatch run_matrixmult.sh
Submitted batch job 49100417
[user@eu-login-39]$ squeue
     JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
  49100417 normal.4h matrixmu     user  R       0:08      1 eu-a2p-532
```

When our job is finished, you will find new files in the directory containing the output of the benchmark programs. For example, we will find the files `matrixmult-jobid.out` and `matrixmult-jobid.err`. The first file contains the standard output and the second file contains the standard error. Additionally, the performance data are stored in `*.data` files and `timing.pdf` is a plot of the performance.

Optimizing square matrix multiplication [30 points]

- **Implementation**: Implement blocking for square matrix multiplication in `dgemm_blocked.c`.
- **Optimization**: Optimize your code to maximize its performance. Consider various strategies, such as compiler options, tuning data access patterns (including block size adjustments), using `#pragma` directives, vectorization, loop unrolling, and more.
- **Documentation and Analysis**: Document the used or attempted optimizations in your report, including performance graphs and, if applicable, tables. Provide a detailed description of your experimental setup. Compare your optimized implementation to the OpenBLAS library.
- **[Bonus 20 points]**: Employ any advanced techniques from library implementations as described in the references provided in the motivation Section 4.1, including any references therein or other relevant resource. However, remember to accurately cite all your sources.

Additional notes and submission details

Submit **all the source code files** together with your used build files (e.g., `Makefile`(s)) and other scripts in an archive file (e.g., tar, zip, etc.) and summarize your results and observations for all sections by writing a detailed LaTeX report. Use the LaTeX template from the webpage and upload the report as a PDF to Moodle.

- Your submission should be an archive, formatted like `project_number_lastname_firstname.zip/tgz`. It must contain:
  - All the source codes of your solutions.
  - Build files an scripts. If you have modified the provided build files or scripts, make sure they still build the sources an run correctly. We will use them to grade your submission.
  - `project_number_lastname_firstname.pdf`, your report with your name.
  - Follow the provided guidelines for the report.
- Submit your archive file through Moodle.

Please follow these instructions and naming conventions. Failure to comply results in additional work for the TAs, which makes the TAs sad...

References

[1] Guillermo Alaejos, Adrián Castelló, Héctor Martínez, Pedro Alonso-Jordá, Francisco D. Igual, and Enrique S. Quintana-Ortí. Micro-kernels for portable and efficient matrix multiplication in deep learning. *The Journal of Supercomputing*, 79(7):8124–8147, May 2023. doi:10.1007/s11227-022-05003-3.

[2] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3):1–25, May 2008. doi:10.1145/1356052.1356053.

[3] Georg Hager and Gerhard Wellein. Introduction to high performance computing for scientists and engineers. *Chapman & Hall/CRC Computational Science*, July 2010. URL: http://dx.doi.org/10.1201/EBK1439811924, doi:10.1201/ebk1439811924.

[4] Jia-Wei Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, STOC '81, pages 326–333, New York, NY, USA, May 1981. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/800076.802486, doi:10.1145/800076.802486.

[5] Intel. Intel oneAPI DPC++/C++ Compiler Developer Guide and Reference. https://cdrdv2.intel.com/v1/dl/getContent/792222?fileName=dpcpp-cpp-compiler_developer-guide-reference_2024.0-767253-792222.pdf, 2024. [Online; accessed 24-February-2024].

[6] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Orti. Analytical Modeling Is Enough for High-Performance BLIS. *ACM Transactions on Mathematical Software*, 43(2):12:1–12:18, August 2016. URL: https://dl.acm.org/doi/10.1145/2925987, doi:10.1145/2925987.

[7] John Markoff. Writing the Fastest Code, by Hand, for Fun: A Human Computer Keeps Speeding Up Chips. *The New York Times*, 2005. [Online; accessed 24-February-2024]. URL: https://www.nytimes.com/2005/11/28/technology/writing-the-fastest-code-by-hand-for-fun-a-human-computer-keeps.html?smid=url-share.

[8] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, aug 1969. doi:10.1007/bf02165411.

[9] Robert van de Geijn, Margaret Myers, and Devangi Parikh. LAFF - On Programming for High Performance. https://www.cs.utexas.edu/users/flame/laff/pfhp/, November 2021. [Online; accessed 24-February-2024].

[10] Field G. Van Zee, Tyler M. Smith, Bryan Marker, Tze Meng Low, Robert A. Van De Geijn, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John A. Gunnels, and Lee Killough. The BLIS framework. *ACM Transactions on Mathematical Software*, 42(2):1–19, jun 2016. doi:10.1145/2755561.