How To Write Fast Numerical Code: A Small Introduction

Srinivas Chellappa, Franz Franchetti, and Markus Püschel

Electrical and Computer Engineering
Carnegie Mellon University
{schellap, franzf, pueschel}@ece.cmu.edu

Abstract. The complexity of modern computing platforms has made it extremely difficult to write numerical code that achieves the best possible performance. Straightforward implementations based on algorithms that minimize the operations count often fall short in performance by at least one order of magnitude. This tutorial introduces the reader to a set of general techniques to improve the performance of numerical code, focusing on optimizations for the computer's memory hierarchy. Further, program generators are discussed as a way to reduce the implementation and optimization effort. Two running examples are used to demonstrate these techniques: matrix-matrix multiplication and the discrete Fourier transform.

1 Introduction

The growth in the performance of computing platforms in the past few decades has followed a reliable pattern usually referred to as Moore's Law. Moore observed in 1965 [1] that the number of transistors per chip roughly doubles every 18 months and predicted—correctly—that this trend would continue. In parallel, due to the shrinking size of transistors, CPU frequencies could be increased at roughly the same exponential rate. This trend has been the big supporter for many performance demanding applications in scientific computing (such as climate modeling and other physics simulations), consumer computing (such as audio, image, and video processing), and embedded computing (such as control, communication, and signal processing). In fact, these domains have a practically unlimited need for performance (for example, the ever growing need for higher resolution videos), and it seems that the evolution of computers is well on track to support these needs.

However, everything comes at a price, and in this case it is the increasing difficulty of writing the fastest possible software. In this tutorial, we focus on *numerical* software. By that we mean code that mainly consists of floating point computations.

The problem. To understand the problem we investigate Fig. 1, which considers various Intel architectures from the first Pentium to the (at the time of this writing) latest Core2 Extreme. The x-axis shows the year of release. The y-axis, in log-scale, shows both the CPU frequency (in MHz) and the single/double precision theoretical peak performance (in Mflop/s = Mega FLoating point OPerations per Second) of the respective machines.

First we note, as explained above, the exponential increase in CPU frequency. This results in a "free" speedup for numerical software. In other words, legacy code written for an obsolete predecessor will run faster without any extra programming effort. However, the theoretical performance of computers has evolved at a faster pace due to increases in the processors' parallelism. This parallelism comes in several forms, including pipelining, superscalar processing, vector processing and multi-threading. Single-instruction multiple-data (SIMD) vector instructions enable the execution of an operation on 2, 4, or more data elements in parallel. The latest generations are also "multicore," which means 2, 4, or more processing cores[1] exist on a single chip. Exploiting parallelism in numerical software is not trivial, it requires implementation effort. Legacy code typically neither includes vector instructions, nor is it multi-threaded to take advantage of multiple processor cores or multiple processors. Ideally, compilers would take care of this problem by automatically vectorizing and parallelizing existing source code. However, while much outstanding compiler research has attacked these problems (e.g., [2–4]), they are in general still unsolved. Experience shows that this is particularly true for numerical problems. The reason is, for numerical problems, taking advantage of the platform's available parallelism often requires an algorithm structured differently than the one that would be used in the corresponding sequential code. Compilers cannot be made to change or restructure algorithms since doing so requires knowledge of the algorithm domain.

Similar problems are caused by the computer's memory hierarchy, independently of the available parallelism. The fast processor speeds have made it increasingly difficult to "feed all floating point execution units" at the necessary rate to keep them busy. Moving data from and to memory has become the bottleneck. The memory hierarchy, consisting of registers and multiple levels of cache, aims to address this problem, but can only work if data is accessed in a suitable order. One cache miss may incur a penalty of 20–100s CPU cycles, a time in which 100 or more floating point operations could have been performed. Again, compilers are inherently limited in optimizing for the memory hierarchy since optimization may require algorithm restructuring or an entirely different choice of algorithm to begin with.

Adding to these problems is the fact that CPU frequency scaling is approaching its end due to limits to the chip's possible power density (see Fig. 1): since 2004 it has hovered around 3 GHz. This implies *the end of automatic speedup*; future performance gains will be exclusively due to increasing parallelism.

In summary, two main problems can be identified from Fig. 1:

- Years of exponential increase in CPU frequency meant free speed-up for existing software but also have caused and worsened the processor-memory bottleneck. This means to achieve the highest possible performance, code has to be restructured and tuned to the memory hierarchy.

[1] At the time of this writing 8 cores per chip is the best commonly available multicore CPU configuration.

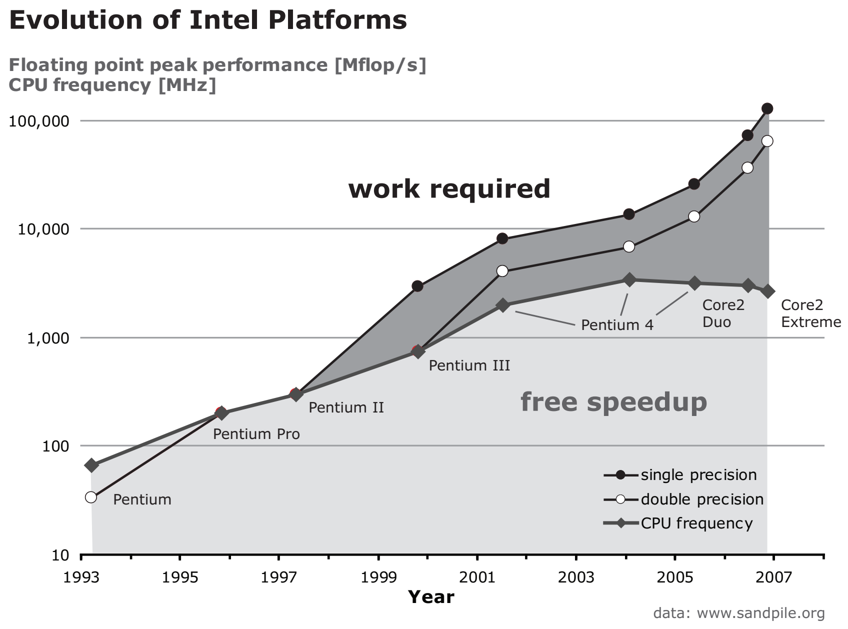


Fig. 1. The evolution of computing platform's peak performance versus their CPU frequency explains why high performance software development becomes increasingly harder.

- The times of free speed-up are over; future performance gains are due to parallelism in various forms. This means, code has to be rewritten using vector instructions and multiple threads and in addition has to be optimized for the memory hierarchy.

To quantify the problem we look at two representative examples, which are among the most important numerical kernels used: the discrete Fourier transform (DFT) and the matrix-matrix multiplication (MMM). The DFT is used across disciplines and is the most important tool used in signal processing; MMM is the crucial kernel in most dense linear algebra algorithms.

It is well-known that the complexity of the DFT for input size n is $O(n \log(n))$ due to the availability of fast Fourier transform algorithms (FFTs) [5]. Fig. 2 shows the performance of four different FFT implementations on an Intel Core platform with four cores. The x-axis is the input size $n = 2^4, \dots, 2^{18}$. The y-axis is the performance in Gflop/s. For all implementations, the operations count is estimated as $5n \log_2(n)$, so the numbers are proportional to inverse runtime. The bottom line shows the performance of the implementation by Numerical Recipes [6] compiled with the best available compiler (the Intel vendor compiler icc 10.1 in this case) and all optimizations enabled. The next line (best scalar) shows the performance of the fastest standard C implementation for the DFT and is roughly 5 times faster due to optimizations for the memory hierarchy. The next line (best vector) shows the performance when vector instructions are used in addition, for a further gain of a factor of 3. Finally, for large sizes, another factor of 2

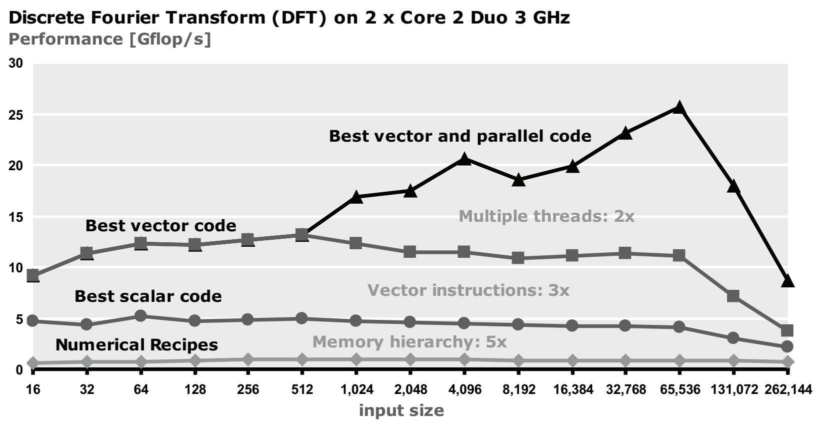


Fig. 2. Performance of four single precision implementations of the discrete Fourier transform. The operations count is roughly the same.

can be gained by writing multi-threaded code to use all processor cores. Note that all four implementations *have roughly the same operations count* for a given size but the performance difference is a factor of 12 for small sizes, and a factor of up to 30 for large sizes. The uppermost three lines correspond to code generated by Spiral [7, 8]; a roughly similar performance is achieved by FFTW [9–11].

Fig. 3 shows a similar plot for MMM (assuming square matrices), where the bottom line corresponds to a standard, triple loop implementation. Here the performance difference with respect to the best code can be as much as 160 times, including a factor of 5-20 solely due to optimizations for the memory hierarchy. All the implementations have exactly the same floating point operations count of $2n^3$. The top two lines are from Goto BLAS [12]; the best scalar code is generated using ATLAS [13].

To summarize the above discussion, the task of achieving the highest performance with an implementation usually lies to a great extent with the programmer. For a given problem, he or she has to carefully consider different algorithms and possibly restructure them to adapt to the given platform's memory hierarchy and available parallelism. This is very difficult, time-consuming, and requires interdisciplinary knowledge about algorithms, software optimizations, and the hardware architecture. Further, the tuning process is platform-dependent: an implementation optimized for one computer will not necessarily be the fastest one on another, since performance depends on many microarchitectural features including but not restricted to the details of the memory hierarchy. Consequently, to achieve highest performance, tuning has to be repeated with the release of each new platform. Since the times of a free speedup (due to frequency scaling) are over, this retuning has become mandatory if any performance gains are desired. Needless to say, the problem is not merely an academic one, but one that affects the software industry as a whole.

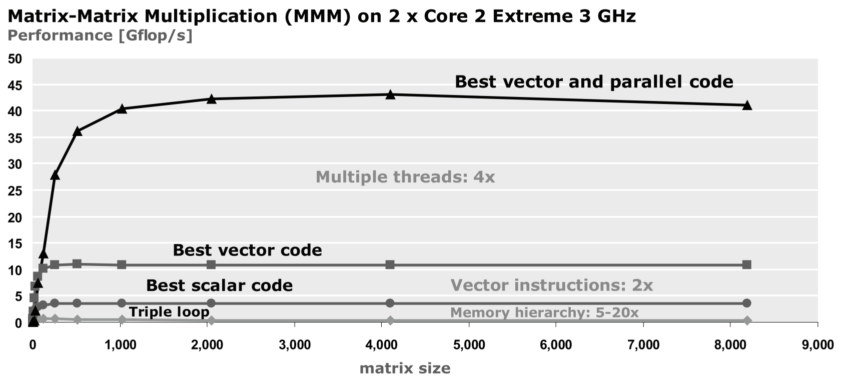


Fig. 3. Performance of four double precision implementations of matrix-matrix multiplication. The operations count is exactly the same.

Automatic performance tuning. A number of research efforts have started to address this problem in a new area called "automatic performance tuning" [14]. The general idea is to at least partially automate the implementation and optimization procedure. Two basic approaches have emerged so far in this area: adaptive libraries and source code generators.

Examples of adaptive libraries include FFTW [10] for the discrete Fourier transform and adaptive sorting libraries [15, 16]. In both cases, the libraries are highly optimized, and beyond that, have degrees of freedom with regard to the chosen divide-and-conquer strategy (both DFT and sorting are done recursively in these libraries). This strategy is determined at runtime, on the given platform, using a search mechanism. This way, the library can dynamically adapt to the computer's memory hierarchy. Sparsity and OSKI from the BeBOP group [17–20] is are other examples of such a libraries, used for sparse linear algebra problems.

On the other hand, source code generators produce algorithm implementations from scratch. They are used to generate either crucial components, or libraries in their entirety. For instance, ATLAS (Automatically Tuned Linear Algebra Software) and its predecessor PHiPAC [21, 18, 22] generate the kernel code for MMM and other basic matrix routines. They do so by generating many different variants arising from different choices of blocking, loop unrolling, and instruction ordering. These are all measured and the fastest one is selected using search methods.

FFTW also uses a generator to produce small size DFT kernels [23]. Here, no search is used, but many optimizations are performed before the actual code is output. Spiral [7, 24] is a library generator for arbitrary sized linear transforms including the DFT, filters, and others. Besides enumerating alternatives, and in contrast to other work, Spiral uses an internal domain-specific mathematical language to optimize algorithms at a high level of abstraction before source code is generated. This includes algorithm restructuring for the memory hierarchy, vector instructions, and multi-threaded code

[24–26]. FLAME considers dense linear algebra algorithm and is in spirit similar to Spiral. It represents algorithms in a structural form and shows how to systematically derive alternatives and parallelize them [27–29].

Other automatic performance tuning efforts include [17] for sparse linear algebra and [30] for tensor computations.

This new research is promising but much more work is needed to automate the implementation and optimization of a large set of library functionality. We believe that program generation techniques will prove crucial for this area of research.

Summary. We summarize the main points of this section:

– *End of free-speedup for legacy code.* CPU frequencies have hit the power wall and stalled. Future performance gains in computers will be obtained by increasing parallelism. This means that code has to be rewritten to take advantage of the available parallelism and performance.

– *Minimizing operations count does not mean maximizing performance*. Floating-point operations are much cheaper than cache misses. Fastest performance requires code that is adapted to the memory hierarchy, uses vector instructions and multiple cores (if available). As a consequence, we have the following problem.

– *The performance difference between a straightforward implementation and the best possible can be a factor of 10, 20, or more.* This is true even if the former is based on an algorithm that is optimal in its (floating-point) operations count.

– *It is very difficult to write the fastest possible code.* The reason is that performance-optimal code has to be carefully optimized for the platform's memory hierarchy and available parallelism. For numerical problems, compilers cannot perform these optimizations, or can only perform them to a very limited extent.

– *Performance is in general non-portable.* The fastest code for one computer may perform poorly on another.

– *Overcoming these problems* by automation is a challenge at the core of computer science. To date this research area is still in its infancy. One crucial technique that emerges in this research area is generative programming.

Goal of this tutorial. The goal of this tutorial is twofold. First, it provides the reader with a small introduction to the performance optimization of numerical problems, focussing on optimizations for the computer's memory hierarchy, i.e., the dark area in Fig. 1 is not discussed. The computers considered in this tutorial are COTS (commercial off-the-shelf) desktop computers with the latest microarchitectures such as Core2 Duo or the Pentium from Intel, the Opteron from AMD, and the PowerPC from Apple and Motorola. We assume that the reader has the level of knowledge of a junior (third year) student in computer science or engineering. This includes basic knowledge of computer architecture, algorithms, matrix algebra, and solid C programming skills.

Second, we want to raise awareness and bring this topic closer to the program generation community. Generative programming is an active field of research (e.g., [31, 32]), but has to date mostly focused on reducing the implementation effort in producing correct code. Numerical code and performance optimization have not been considered. In contrast, in the area of automatic performance tuning, program generation has started to emerge as one promising tool as briefly explained in this tutorial. However, much more research is needed and any advances have high impact potential.

The tutorial is in part based on the course [33].

Organization. Section 2 provides some basic background information on algorithm analysis, the MMM and the DFT, features of modern computer systems relevant to this tutorial, and compilers and their correct usage. It also identifies data access patterns that are necessary for obtaining high performance on modern computer systems. Section 3 first introduces the basics of benchmarking numerical code and then provides a general high-level procedure for attacking the problem of performance optimization given an existing program that has to be tuned for performance. This procedure reduces the problem to the optimization of performance-critical kernels, which is first studied in general in Section 4 and then in Sections 5 and 6 using MMM and the DFT as examples. The latter two sections also explain how program generators can be applied in this domain using ATLAS (for MMM) and Spiral (for the DFT) as examples. We conclude with Section 7.

Along with the explanations, we provide programming exercises to provide the reader with hands-on experience.

2 Background

In this section we provide the necessary background for this tutorial. We briefly review algorithm analysis, introduce MMM and the DFT, discuss the memory hierarchy of off-the-shelf microarchitectures, and explain the use of compilers. The following standard books provide more information on algorithms [34], MMM and linear algebra [35], the DFT [5, 36], and computer architecture and systems [37, 38].

2.1 Cost Analysis Of Algorithms

The starting point for any implementation of a numerical problem is the choice of algorithm. Before an actual implementation, algorithm analysis, based on the number of operations performed, can give a rough estimate of the performance to be expected. We discuss the floating point operations count and the degree of reuse.

Cost: asymptotic, exact, and measured. It is common in algorithm analysis to represent the asymptotic runtime of an algorithm in O-notation as $O(f(n))$, where n is the input size and f, a function [34]. For numerical algorithms, $f(n)$ is typically determined from the number of floating point operations performed. The O-notation neglects constants and lower order terms; for example, $O(n^3 + 100n^2) = O(5n^3)$. Hence it is only suited to describe the performance *trend* but not the *actual* performance itself. Further, it makes a statement only about the asymptotic behavior, i.e., the behavior as n goes to infinity. Thus it is in principle possible that an $O(n^3)$ algorithm performs better than an $O(n^2)$ algorithm for all practically relevant input sizes n.

A better form of analysis for numerical algorithms is to compute the *exact* number of floating point operations, or at least the exact highest order term. However, this may be difficult in practice. In this case, profiling tools can be used on an actual implementation to determine the number of operations actually performed. The latter can also be used to determine the computational bottleneck in a given implementation.

However, even if the exact number of operations of an algorithm and its implementation is known, it is very difficult to determine the actual runtime. As an example consider Fig. 3: all four implementations require exactly $2n^3$ operations, but the runtime differs by up to two orders of magnitude.

Reuse: CPU bound vs. memory bound. Another useful measure of an algorithm is the degree of reuse. The asymptotic reuse for an $O(f(n))$ algorithm is given by $O(f(n)/n)$ if n is the input size. Intuitively, the degree of reuse measures how often a given input value is used in a computation during the algorithm. A high degree of reuse implies that an algorithm may perform better (in terms of operations per second) on a computer with memory hierarchy, since the number of computations dominates the number of data transfers from memory to CPU. In this case we say that the algorithm is *CPU bound*. A low degree of reuse implies that the number of data transfers from memory to CPU is high compared to the number of operations and the performance (in operations per second) may deteriorate: in this case we say that the algorithm is *memory bound*.

A CPU bound algorithm will run faster on a machines with a faster CPU. A memory bound algorithm will run faster on a machine with a faster memory bus.

2.2 Matrix-Matrix Multiplication

Matrix-matrix multiplication (MMM) is arguably the most important numerical kernel functionality. It is used in many linear algebra algorithms such as solving systems of linear equations, matrix inversion, eigenvalue computations, and many others. We will use MMM, and the DFT (Section 2.3) as examples to demonstrate optimizations for performance.

Definition. Given a $k \times m$ matrix $A = [a_{i,j}]$ and an $m \times n$ matrix $B = [b_{i,j}]$, the product $C = AB$ is a $k \times n$ matrix with entries

$$c_{i,j} = \sum_{k=1}^{m} a_{i,k} b_{k,j}.$$

For actual applications, usually $C = C + AB$ is implemented instead of $C = AB$.

Complexity and analysis. Given two $n \times n$ matrices A, B, MMM computed as $C = C + AB$ by definition requires n^3 multiplications and n^3 additions for a total of $2n^3 =$

$O(n^3)$ floating point operations. Since the input data (the matrices) have size $O(n^2)$, the reuse is given by $O(n^3/n^2) = O(n)$.

Asymptotically better MMM algorithms do exist. Strassen's algorithm [39] requires only $O(n^{\log_2 7}) \approx O(n^{2.808})$ operations. The actual crossover point (i.e., when it requires less operations than the computation by definition) is at $n = 655$. However, the more complicated structure of Strassen's algorithm and its weaker numerical stability reduce its applicability. The best-known algorithm for MMM is due to Coppersmith-Winograd and requires $O(n^{2.376})$ [40]. The large hidden constant and a complicated structure have so far made this algorithm impractical for real applications.

Direct implementation. A direct implementation of MMM is the triple loop shown below.

```
// MMM - direct implementation
for(i=0; i<m; i++)
  for(j=0; j<p; j++)
    for(k=0; k<n; k++)
      c[i][j] += a[i][k] * b[k][j];
```

BLAS and LAPACK. BLAS (Basic Linear Algebra Subprogram) is a set of standardized basic linear algebra operations, including MMM [41]. Implementations of BLAS are provided by packages such as ATLAS and Goto BLAS. BLAS routines are used as kernels in fundamental linear algebra algorithms such as linear equation solving, eigenvalue computations, singular value decompositions, LU/Cholesky/QR decompositions, and others. Such higher level functions are implemented by the LAPACK (Linear Algebra PACKage) library, [42] using MMM and other BLAS routines as kernels (see Fig. 4). The idea behind this two-level design is to redesign and/or re-optimize the BLAS implementations for new hardware architectures, while reusing LAPACK without a need for modification. The performance improvements from the BLAS implementation then translate into performance gains for the LAPACK library. This design has proven very successful until the release of multicore systems, which appears to require a redesign of LAPACK.

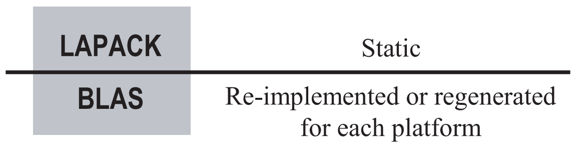


Fig. 4. LAPACK is implemented on top of BLAS.

Further reading.

– *Linear algebra.* General information about numerical linear algebra can be found in [35, 38].

- *BLAS.* ATLAS provides an implementation of BLAS, as does Goto BLAS. Further information on ATLAS is available in [13, 21, 43]. Details on Goto BLAS can be found at [12, 44].
- *Linear algebra libraries.* LAPACK is described in [45, 42]. The distributed memory extension ScaLAPCK is described in [46, 47]. An alternative approach is pursued by PLAPACK [48, 49] and FLAME [28, 27, 50].

2.3 Discrete Fourier Transform

The discrete Fourier transform (DFT) is another numerical kernel of importance in a wide range of disciplines. In particular, in the field of signal processing, the DFT is arguably the most important tool used. Even though the DFT seems at first glance based on linear algebra, it is in its nature fundamentally different from MMM. In particular, it is never computed by definition–fast algorithms are always used, instead. The techniques used by these fast algorithms are different from the techniques used to speed up MMM.

Definition. The discrete Fourier transform (DFT) of an input vector x of length n is defined as the matrix-vector product

$$y = \mathrm{DFT}_n\, x, \quad \mathrm{DFT}_n = [\omega_n^{k\ell}]_{0\leq k,\ell<n}, \quad \omega_n = e^{-2\pi i/n}, \quad i = \sqrt{-1}.$$

In words, ω_n is a primitive nth root of unity. In this tutorial we assume that n is a two-power.

Complexity and analysis. Computing the DFT by definition requires $O(n^2)$ many operations, and is never done in practice. There exists a number of fast algorithms, called fast Fourier transforms (FFTs), that reduce the runtime to $O(n\log(n))$ for all sizes n [5]. For $n = 2^k$, the FFTs used in practice require between $4n\log_2(n) + O(n)$ and $5n\log_2(n) + O(n)$ many operations. The best known FFT has a cost of $\frac{34}{9}n\log_2 n + O(n)$ [51]. The degree of reuse is hence $O(\log(n))$, less than for MMM, which explains the performance drop in Fig. 2 for large sizes when the working set is too large for the L2 cache.

We defer a detailed introduction of FFTs to Section 6.

Direct implementation. In contrast to MMM, a straightforward implementation of the DFT is not done by definition, but performed by a direct implementation of an FFT. One example is the so-called iterative radix-2 FFT algorithm as implemented by Numerical Recipes [6], whose performance was shown in Fig. 2. The corresponding code is shown below.

```
#include <math.h>

#define SWAP(a,b) tempr=a;a=b;b=tempr
void four1(float *data, int *nn, int *isign)
{ /* altered for consistency with original FORTRAN.
  /* Press, Flannery, Teukolsky, Vettering "Numerical
```

```
 * Recipes in C" tuned up ; Code works only when *nn is
 * a power of 2  */
int n, mmax, m, j, i;
double wtemp, wr, wpr, wpi, wi, theta, wpin;
double tempr, tempi, datar, datai,
    data1r,data1i;
n = *nn * 2;
j = 0;
for(i = 0; i < n; i += 2)
{ if (j > i) {            /* could use j>i+1 to help
                           * compiler analysis */
      SWAP(data[j], data[i]);
      SWAP(data[j + 1], data[i + 1]);
  }
  m = *nn;
  while (m >= 2 && j >= m) {
      j -= m;
      m >>= 1;
  }
  j += m;
}
theta = 3.141592653589795 * .5;
if (*isign < 0)
    theta = -theta;
wpin = 0;                   /* sin(+-PI) */
for(mmax = 2; n > mmax; mmax *= 2)
{ wpi = wpin;
  wpin = sin(theta);
  wpr = 1 - wpin * wpin - wpin * wpin;
  /* cos(theta*2) */
  theta *= .5;
  wr = 1;
  wi = 0;
  for(m = 0; m < mmax; m += 2)
  { j = m + mmax;
    tempr = (double) wr *(data1r = data[j]);
    tempi = (double) wi *(data1i = data[j + 1]);
    for(i = m; i < n - mmax * 2; i += mmax * 2)
    { /* mixed precision not significantly more
       * accurate here; if removing double casts,
       * tempr and tempi should be double */
      tempr -= tempi;
      tempi = (double) wr *data1i + (double) wi *data1r;
      /* don't expect compiler to analyze j > i+1 */
      data1r = data[j + mmax * 2];
      data1i = data[j + mmax * 2 + 1];
      data[i] = (datar = data[i]) + tempr;
      data[i + 1] = (datai = data[i + 1]) + tempi;
      data[j] = datar - tempr;
      data[j + 1] = datai - tempi;
```

```
          tempr = (double) wr *data1r;
          tempi = (double) wi *data1i;
          j += mmax * 2;
        }
        tempr -= tempi;
        tempi = (double) wr *data1i + (double) wi *data1r;
        data[i] = (datar = data[i]) + tempr;
        data[i + 1] = (datai = data[i + 1]) + tempi;
        data[j] = datar - tempr;
        data[j + 1] = datai - tempi;
        wr = (wtemp = wr) * wpr - wi * wpi;
        wi = wtemp * wpi + wi * wpr;
      }
    }
  }
```

Further reading.

- *FFT algorithms.* [52, 36] give an overview of FFT algorithms. [5] uses the Kronecker product formalism to describe many different FFT algorithms, including parallel and vector algorithms. [53] uses the Kronecker formalism to parallelize and vectorize FFT algorithms.

- *FFTW.* FFTW can be downloaded at [11]. The latest version, FFTW3, is described in [10]. The previous version FFTW2 is described in [9] and the codelet generator *genfft* in [23].

- *SPIRAL.* Spiral is a program generation system for transforms. The core system is described in [7] and on the web at [8]. Using Kronecker product manipulations, SIMD vectorization is described in [54, 24], shared memory (SMP and multicore) parallelization in [25], and message passing (MPI) in [55].

- *Open source FFT libraries.* FFTPACK [56] is a mixed-radix Fortran FFT library. The GNU Scientific library (GSL) [57] contains a C port of FFTPACK. UHFFT [58, 59] is an adaptive FFT library. Numerical Recipes [6] contains the radix-2 FFT implementation shown above. FFTE [60] provides a parallel FFT library for distributed memory machines.

- *Proprietary FFT libraries.* The AMD Core Math Library (ACML) [61] is the vendor library for AMD processors. Intel provides fast FFT implementations as a part of their Math Kernel Library (MKL) [62] and Integrated Performance Primitives (IPP) [63]. IBM's IBM Engineering and Scientific Software Library (ESSL) [64] and the parallel version (PESSL) contain FFT functions optimized for IBM machines. The vDSP library contains FFT functions optimized for AltiVec. The libraries of the Numerical Algorithms Group (NAG) [65] and the International Mathematical and Statistical Library (IMSL) [66] also contain FFT functionality.

2.4 State-Of-The-Art Desktop and Laptop Computer Systems

Modern computers include several performance enhancing microarchitectural features like cache systems, a memory hierarchy, virtual memory, and CPU features like vector and parallel processing. While these features usually increase the achievable performance, they also make the optimization process more complex. This section introduces several microarchitectural features relevant to writing fast code. For further reading, refer to [37, 38].

Memory hierarchy. Most computer systems use a *memory hierarchy* to bridge the speed gap between the processor(s) and its connection to main memory. As shown in Fig. 5, the highest levels of the memory hierarchy contain the fastest and the smallest memory systems, and vice versa.

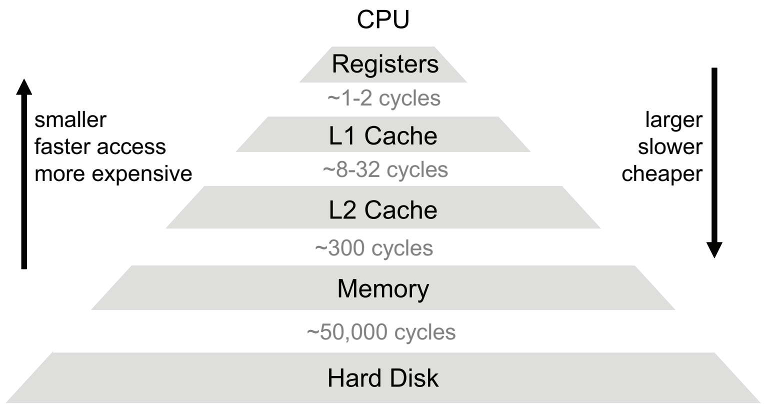


Fig. 5. Memory hierarchy. Typical latencies for data transfers from the CPU to each of the levels are shown. The numbers shown here are only an indication, and the actual numbers will depend on the exact architecture under consideration.

A hierarchical memory enables the processor to take advantage of the memory locality of computer programs. Optimizing numerical programs for the memory hierarchy is one of the most fundamental approaches to producing fast code, and the subject of this tutorial. Programs typically exhibit temporal and spatial memory locality. Temporal locality means that a memory location that is referenced by a program will likely be referenced again in the near future. Spatial locality means that the likelihood of referencing a memory location by a program is higher if a nearby location was recently referenced. High performance computer software must be designed so that the hardware can easily take advantage of locality. Thus, this tutorial focuses on writing fast code by designing programs to exhibit maximal temporal and spatial localities.

Registers. Registers inside the processor are the highest level of the memory hierarchy. Any value (address or data) that is involved in computation has to eventually be placed into a register. Registers may be designed to hold only a specific type of value (special purpose registers), or only floating point values (eg., double FP registers), vector values (vector registers), or any value (general purpose registers). The number of registers in a processor varies by architecture. A few examples are provided in Table 1. When an active computation requires more values to be held than the register space will allow, a *register spill* occurs, and the register contents are written to lower levels of memory from which they will be reloaded again. Register spills are expensive. To avoid them and speed up computation, a processor might make use of internal registers that are not visible to the programmer. Many optimizations that work on other levels of the memory hierarchy can typically also be extended to the register level.

Processor	Integer Registers	Double FP Registers
Core2 Extreme	16	16
Itanium 2	128	128
UltraSPARC T2	32	32
POWER6	32	32

Table 1. Sample scalar register space (per core) in various architectures. In addition to integer and FP registers, the Core2 Extreme also has 16 multimedia registers.

Cache memory. Cache memory is a small, fast memory that resides between the main memory and the processor. It reduces average memory access times by taking advantage of spatial and temporal locality. When the processor initially requests data from a memory location (called a cache miss), the cache fetches and stores the requested data and data spatially close. Subsequent accesses, called *hits*, can be serviced by the cache without needing to access main memory. A well designed cache system has a low miss to hit ratio (also known as just the miss ratio or miss rate).

Level/Type	Size	Associativity
L1 Data (per core)	32 kB	8-way set associative
L1 Instruction (per core)	32 kB	8-way set associative
L2 Unified (common)	4 MB	8-way set associative

Table 2. Cache system example: Intel Core2 Duo, Merom Notebook processor.

Caches, as shown in Fig. 6 are divided into cache lines (also known as blocks) and sets. Data is moved in and out of cache memory in chunks equal to the line size. Cache lines exist to take advantage of spatial locality. Multiple levels of caches and separate data and instruction caches may exist, as shown in Table 2. Caches may be direct mapped (every main memory location is mapped to a specific cache location) or k-way set associative (every main memory location can be mapped to precisely k possible cache locations).

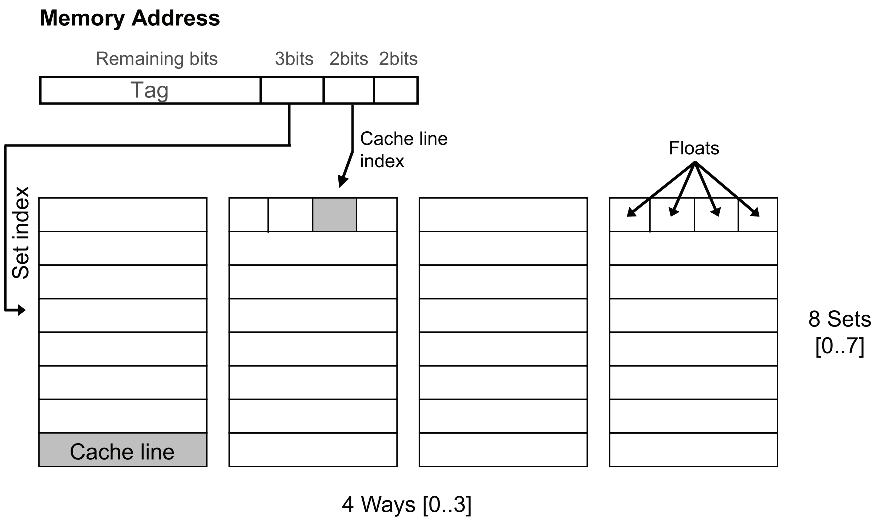


Fig. 6. 4-way set associative cache with cache line size of 4 single precision floats (4 bytes per float) per line, and cache size of 128 floats (total cache size is 512 bytes). The figure also illustrates the parts of a memory address used to index into the cache. Since each data element under consideration is 4 bytes long, the two least significant bits are irrelevant in this case. The number of bits used to address into the cache line would be different for double precision floats.

In addition to misses caused due to data being brought in for the first time (compulsory misses) and those due to cache capacity constraints (capacity misses), caches that are not fully associative can incur conflict misses [67].

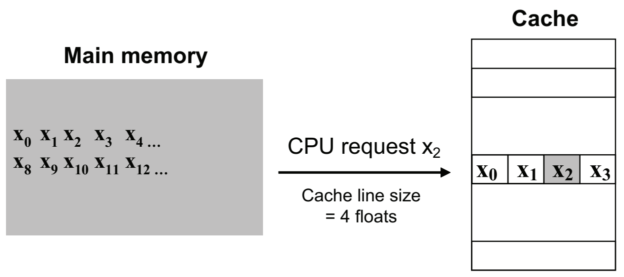


Fig. 7. Neighbor use and reuse: When the CPU requests x_2, x_0, x_1, and x_3 are also brought into the cache since the cache line size holds 4 floats.

Since cache misses are typically expensive, writing fast code involves designing programs to have low miss rates. This is accomplished using two important guiding principles, illustrated in Fig. 7 and described below:

– **Reuse: Temporal locality.** Once data is brought into the cache, the program should reuse it as much as possible before it gets evicted. In other words, programs must try to avoid scattering computations made on a particular data location throughout the execution of the program. Otherwise, the same data (or data location) has to go through several cycles of being brought into the cache and subsequently evicted, which increases runtime.

– **Neighbor use (using all data brought in): Spatial locality.** Data is always brought into the cache in chunks the size of a cache line. This is seen in Fig. 7, where one data element x_2 was requested, and three others are also brought in since they belong to the same cache line. To take advantage of this, programs must be designed to perform computations on neighboring data (physically close in memory) before the line is evicted. This might involve reordering loops, for instance, to work on data in small chunks.

These two principles work at multiple levels. For instance, code can be designed to use and reuse all data within a single cache block, as well as within an entire cache level. In fact, these principles hold throughout the memory hierarchy, and thus can be used at various cache and memory levels. Depending on the computation being performed, techniques that use these principles may not be trivial to design or implement.

In scientific or numerical computing, data typically consists of floating point numbers. Therefore, it helps to view the cache organization, lines, and sets in terms of the number of floating point numbers that can be held. For instance, the cache shown in Fig. 6 is a 512 byte, 4-way set associative cache with a line size of 16 bytes. There are a total of 32 lines (512 bytes / 16 bytes per line), and 8 sets (32 lines / 4 lines per set). If we note that each cache line can hold 4 floats (16 bytes / 4 bytes per float), we can immediately see that the cache can hold a total of 128 floats. This means that datasets larger than 128 floats will not fit in the cache. Also, if we make an initial access to 128 consecutive floats, there will be a total of 32 cache misses and 96 cache hits (since 4 floats in a line are loaded on each cache miss). This gives us a rough estimate of the runtime of such a set of accesses, which is useful both in designing programs and in performing sanity checks.

Cache analysis. We now consider three examples of accessing an array in various sequences, and analyze their effects on the cache.

Consider a simple direct mapped 16 byte data cache with two cache lines, each of size 8 bytes (two floats per line). Consider the following code sequence, in which the array X is cache-aligned (that is, $X[0]$ is always loaded into the beginning of the first cache line) and accessed twice in consecutive order:

```
float X[8];
for(int j=0; j<2; j++)
  for(int i=0; i<8; i++)
    access(X[i]);
```

The top row on Fig. 8 shows the states of the cache after every two (out of the total of sixteen) accesses for this example. To analyze the cache footprint and pattern of this code sequence, we first observe that the size of the array (8 floats) exceeds the size of

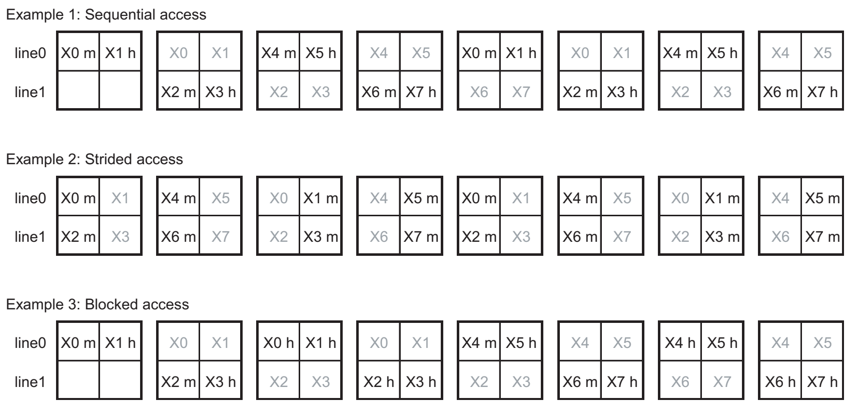


Fig. 8. Cache access analysis: The state of the complete cache for each example is shown after every two accesses, along with whether the two accesses resulted in hits or misses (shown by h or m). The two requests just made are shown in black, while the remaining parts of the cache are shown in gray. To save space, square brackets are not shown: $X0$ refers to $X[0]$.

the cache (4 floats). We then observe that a total of 16 accesses are made to the array. To calculate how many result in hits, and how many in misses, we observe the cache access pattern of the code. The pattern is “0123456701234567” (only the indices of X accessed are shown). We note that an access to any even index of X results in that element and the subsequent element being loaded since they are in the same cache line. Thus, accessing $X[0]$ loads $X[0]$ and $X[1]$ into the cache. We can then compute the hit/miss pattern to be: “MHMHMHMHMHMHMHMH”. So in all, there are 8 hits and 8 misses.

We now look at another code sequence that again accesses the same array twice (similar to the last example), albeit with a stride of 2:

```
float X[8];
for(int j=0; j<2; j++)
{ for(int i=0; i<7; i+=2)
    access(X[i]);
  for(int i=1; i<8; i+=2)
    access(X[i]);
}
```

The middle row on Fig. 8 shows the corresponding cache states for this example. The access pattern here is “0246135702461357”. A similar analysis shows us that the miss ratio is even worse: every single access in this pattern results in a miss (with a total of 16 misses and 0 hits). This example illustrates an important point: strided accesses generally result in poor cache efficiency, since they effectively “make the cache smaller.”

Finally, let us consider a third code sequence that again accesses the same array twice:

```
float X[8];
for(i=0; i<2; i++)
  for(k=0; k<2; k++)
    for(j=0; j<4; j++)
      access(X[j+(i*4]);
```

The bottom row on Fig. 8 shows the corresponding cache states for this example. The access pattern here is "0123012345674567". Counting the hits and misses, ("MHMHHHHHMHMHHHHH"), we observe that there are 12 hits and 4 misses. We also note that if this rearrangement is legal, it is a cache optimized version of the original code sequence. In fact, this rearrangement is an example of both of the previously mentioned principles behind optimizing for the memory hierarchy: reuse and neighbor use. Unlike the first example, the "0123" block is reused here before being evicted. Unlike the second example, every time an even-indexed element is accessed, the succeeding odd-indexed element which is a part of the same cache line is also immediately accessed. Thus, analyzing the cache can help us estimate and improve the cache performance of a program.

CPU features. Modern microprocessors also contain other performance enhancing features. Most processors contain pipelined superscalar out-of-order cores with multiple execution units. Pipelining is a form of parallelism where different parts of the processor work simultaneously on different components of different instructions. Superscalar cores can retire more than one instruction per processor clock cycle. Out-of-order processing cores can detect instruction dependencies and reschedule the instruction sequence for performance. The programmer has to be cognizant of these features in order to be able to optimize for a particular architecture.

Most such aggressive cores also contain multiple execution units (for instance, floating point units) for increased performance. This means that a processor might be able to, for instance, simultaneously retire one floating point add instruction every cycle, and one floating point multiplication instruction every other cycle. It is up to the programmer and the compiler to keep the processor's execution units adequately busy (primarily via instruction scheduling and memory locality) in order to achieve maximum performance.

The theoretical rate at which a processor can perform floating point operations is know as the processor's *theoretical peak performance*. This is measured in flop/s (FLoating point OPerations per Second). For instance, a processor running at 1 GHz that can retire one addition every cycle, and one multiplication every other cycle has a theoretical peak of 1.5 Gflop/s. The theoretical peak of a Core2 Extreme processor operating under various modes is shown in Table 3.

In practice, cache misses, pipeline stalls due to dependencies, branches, branch mispredictions, and the fact that meaningful programs contain instructions other than floating point instructions, do not allow a processor to perform at its theoretical peak performance. Further, the achievable performance also depends on the inherent limitations of the algorithm, such as reuse. For example, MMM, with a reuse degree of $O(n)$ can

	1 core	2 cores	4 cores
x87 double	6	12	24
SSE2 double	12	24	48
x87 float	6	12	24
SSE float	24	48	96

Table 3. Core2 Extreme: Peak performance (in Gflop/s) for a 3 GHz Core2 Extreme processor in various operation modes.

achieve close to the peak performance of 48 Gflop/s (as seen in Fig. 3), whereas the DFT with a reuse degree of $O(\log(n))$ reaches only about 50% (as seen in Fig. 2).

In summary, knowing a processor's theoretical peak and an algorithm's degree of reuse gives us a rough estimate of the extent to which a program could potentially be improved.

Modern processors also contain two major explicit forms of parallelism: vector processing and multicore processing, which are important for writing fast code, but beyond the scope of this tutorial.

Further reading.

– *General computer architecture.* [37, 38].

– *CPU/architecture specific.* [68, 69].

2.5 Using Compilers

To produce fast code it is not sufficient to write and optimize source code–the programmer must also ensure that the code that is written gets compiled into an efficient binary executable. This involves the careful selection and use of compiler flags, use of language extensions, and monitoring and analyzing the compiler's output. Furthermore, in some situations, it is best to let the compiler know of all the degrees of freedom it has, so it can optimize well. In other situations, it is best to direct the compiler to do exactly what is required. This section goes over the compile process, what to keep in mind before, while, and after compiling, and some of the common pitfalls related to the compiling process.

Variable declaration: memory allocation. Understanding how C handles the allocation of space for variables is beneficial. C assigns variables to different *storage class specifiers* by default, based on where in the source code they appear. The default storage class for a variable can be overridden by preceding a variable declaration with the desired storage class specifier.

Variables that are shared among source files use the `extern` storage class. Global variables belong to the `static` storage class, and typically exist in static memory (as do extern variables), which means that they exist as long as the program executes. Local variables belong to the `auto` (automatic) storage class, which means that they are allocated on the stack automatically upon entering the local block within which they are defined, and destroyed upon exit. The `register` storage class requests that the compiler allocates space for the variable directly in the CPU registers. These are useful to eliminate load/store latencies on heavily used variables. Keep in mind that depending on the compiler being used, care should be taken to initialize variables before usage.

Variable declaration: qualifiers. Most compilers provide further means to specify variable attributes through *qualifiers*. A `const` qualifier specifies that a variable's value will never change. A `volatile` qualifier is used to refer to variables whose values might be influenced by sources external to the compiler's knowledge. Operations involving volatile variables will not be optimized by the compiler, in order to preserve correctness. A `restrict` qualifier is especially useful to writing fast code, since it tells the compiler that a certain memory address will be restricted to access via the specified pointer. This allows for effective compiler optimization.

Finally, memory alignment can also be specified by qualifiers. Such qualifiers are specific to the compiler being used. For instance, `__attribute__ ((aligned(128)))` requests a variable to be aligned at the specified 128-byte memory boundary. Such requests allow variables to be aligned to cache line boundaries or virtual memory pages as desired. Similar qualifiers can be used to tell the compiler that the address pointed to by a pointer is memory aligned.

Dynamic memory allocation. Dynamic memory allocation, using `malloc` for example, involves allocating memory in the *heap*, and returning a pointer to the allocated memory. If alignment is of importance, many libraries provide a `memalign` function (the Intel equivalent is `_mm_malloc`) to allocate memory aligned to a specified boundary. The alternative is to allocate more memory than required, and to then check and shift the returned pointer adequately to achieve the required alignment.

Inline assembly and intrinsics. Sometimes, it is best to write assembly code to access powerful features of the machine which may not be available via C. Assembly can be included as a part of any program in C using inline assembly. However, inline assembly use must be minimized as it might interfere with compiler optimizations. Architecture vendors typically provide C language extensions to allow programmers to access special machine instructions. These extensions, called *intrinsics*, are similar to function calls that allow the programmer to avoid writing inline assembly. Importantly, intrinsics allow the compiler to understand what data and/or control the programmer is manipulating, thus allowing for better optimization. As an example, Intel's MMX and SSE extensions to the x86 ISA can be accessed via C intrinsics provided by Intel.

Compiler flags. Most compilers are highly configurable via a plethora of command line options and flags. In fact, finding the right set of compiler options that yield optimal performance is non-trivial. However, there are some basic ideas to keep in mind while using a compiler, as listed below. Note that these ideas apply to most compilers.

- *C standards.* A compiler can be set to follow a certain C standard such as C99. Certain qualifiers and libraries might need specific C standards to work. By switching to a newer standard, the programmer can typically communicate more to the compiler, thus enabling it to work better.

- *Architecture specifications.* Most compilers will compile and optimize by default for a basic ISA standard to maximize compatibility. Machine and architecture specific optimizations may not be performed as a result. For instance, a compiler on an AMD Athlon processor may compile to the x86 standard by default, and not perform Athlon-specific optimizations. Instructing the compiler to compile for the correct target architecture may result in considerable performance gains. Additional flags may be required for these optimizations. For example, gcc requires the "`-sse`" flag to include vector instructions.

- *Optimization levels.* Most compilers usually define several optimization levels that can be selected. Determining the optimization level that yields maximum performance is a black art usually done by trial and error. A more aggressive optimization level doesn't necessarily yield better performance. Optimization levels are usually a shortcut to turn on or off a large set of compiler flags (discussed next).

- *Specialized compiler options.* Compilers typically perform numerous optimizations, many which can be selectively turned on or off and configured through command line flags. Loop unrolling, function inlining, instruction scheduling, and other loop optimizations are only some of the available configurable optimizations. Usually, finding the right optimization level is sufficient, but sometimes, inspection of assembly code provides insights that can be used to fine-tune compiler optimizations.

Compiler output. The output of the compiler is usually an executable binary. As mentioned earlier, the compiler can also be used to produce various intermediate stages, including the preprocessed source, assembly code, and the object code. Sometimes, it is important and useful to visually inspect the assembly code to better understand both the performance of an executable and the behavior of the compiler.

Compilers also output warnings, which can be controlled through compiler flags. Sometimes, a seemingly innocuous warning might provide excellent insights into the source of a bug, which makes these warnings a significant debugging tool.

Optimization reports are an important part of the compiler output that must be inspected. For instance, a vectorizing compiler will inform the programmer of whether it was able to successfully vectorize or not. A failure to vectorize a program that was expected to be vectorized is a reason for examining the program carefully, and modifying or annotating the code as appropriate.

In conclusion, it is important for programmers to be knowledgeable about the compiler that they use in order to be able to use the compiler efficiently, and to ensure that poor compiler usage does not diminish the results of code designed for high performance.

Further reading.

- *Gnu compiler collection (gcc).* [70].
- *Intel compiler.* [71].

2.6 Exercises

1. **Direct implementations.** Implement, execute, and verify:

   - a direct implementation of MMM (code snippet given in Section 2.2),
   - the Numerical Recipes code for the DFT as given in [6],

   This code will also be used in the exercises of later sections.

2. **Determining hardware information.** In this exercise, you will determine the relevant hardware configuration of your computer. You will use this information in later exercises.

   Determine the following information about your computer:

   - CPU type and clock speed
   - For each cache: size, associativity, and cache line size
   - Size of main memory
   - System bus speed

   Here are a few tips on how to determine this information:

   - Look in the computer's manual.
   - Look in the CPU manufacturer's manual.
   - To obtain CPU information in Linux, execute `cat /proc/cpuinfo`.
   - To obtain cache information in Linux, search for lines with the word "cache" in the kernel ring buffer. You can do so by typing: `dmesg | grep '^CPU.*cache'` on most systems.

3. **Loop optimization for the cache.** Consider a 2-way set associative cache with a cache size of 32KB, a cache line size of 32B, and a FIFO (First In, First Out) replacement policy (this means if one of the two candidate cache lines has to be replaced, it will be the one that was first brought into the cache). Consider two single-precision floating point arrays (single precision float = 4B), A and B with n elements, where n is much larger than the cache and is a multiple of the cache size. Further, assume that A and B are both fully cache-aligned, i.e., $A[0]$ and $B[0]$ map to the first position in the first cache line.

   Now consider the following pseudocode snippet:

```
for(i from 0 to n-1)
  A[i] = A[i] + B[f(i)]
```

where $f(i)$ is an index mapping function that reads B at a stride of 8. (If for example, B was 16 elements long, then reading it at stride 8 would result in this access pattern: $f(i) = 0, 8, 1, 9, 2, 10, 3, 11, 4, 12, 5, 13, 6, 14, 7, 15$).

Assume an empty cache for each part of this exercise.

(a) (Disregard the code snippet for this part) What is the expected number of cache misses incurred by streaming once completely through array A alone in sequential order, given the cache parameters above?

(b) (Disregard the code snippet for this part) What is the expected number of cache misses incurred by streaming once completely through array B alone at stride of 8 given the cache parameters above?

(c) How many cache misses is the given pseudocode snippet expected to incur? (Assume, for simplicity, that index variables are not cached).

(d) Rewrite the code (without changing the semantics, i.e., overall computation) to reduce the number of cache misses as much as possible. (Assume, for simplicity, that index variables are not cached).

3 Performance Optimization: The Basics

In this section we will review the basic steps required to assess the performance of a given implementation, also known as "benchmarking." We focus on runtime benchmarking as the most important case. (Other examples of benchmarking includes assessing the usage of memory or other resources.)

For a given program, the basic procedure consists of three steps:

1. finding the hotspots (hotspots are the most frequently executed code regions),
2. timing the hotspots, and
3. analyzing the measured runtimes.

It is essential to find the parts of the program that perform the bulk of the computation and restrict further investigation to these *hotspots*. Optimizing other parts of the program will have little to no effect on the overall runtime. In order to obtain a meaningful runtime measurement, one has to build a test environment for each hotspot that exercises and measures it in the correct way. Finally, one has to assess the measured data and relate it to the cost analysis of the respective hotspot. This way one can make efficiency statements and target the correct (inefficient) hotspot for further optimization.

3.1 Finding The Hotspots

The first step in benchmarking is to find the parts of the program where most time is spent. Most development platforms contain a *profiling* tool. For instance, the development environment available on the GNU/Linux platform contains the GNU gprof profiler. On Windows platforms, the Intel VTune tool [72] that plugs into Microsoft's Visual Studio [73] can be used to profile applications.

If no profiling tool is available, obtain first-order profiling information can be obtained by inserting statements throughout the program that print out the current system time. In this case, less is more, as inserting too many time points may have side effects on the measured program.

Example: GNU tool chain. We provide a small example of using the GNU tool chain to profile a sample program.

Consider the following program:

```
#include <stdio.h>

float function1()
{ int i; float retval=0;
  for(i=1; i<1000000; i++)
    retval += (1/i);
  return(retval);
}

float function2()
{ int i; float retval=0;
  for(i=1; i<10000000; i++)
    retval += (1/(i+1));
  return(retval);
}

void function3() { return; }

int main()
{ int i;
  printf("Result: %.2f\n", function1());
  printf("Result: %.2f\n", function2());
  if (1==2) function3();
  return(0);
}
```

Our final objective is to optimize this program. In order to do so, we first need to find where the program spends most of its execution time, using `gprof`.

As specified in the `gprof` manual [74], three steps are involved in profiling using `gprof`:

1. Compile and link with profiling enabled:

```
gcc -O0 -lm -g -pg -o ourProgram ourProgram.c
```

The resulting executable is instrumented. This means that in addition to executing your program, it will also write out profiling data when executed. (Note: We use the `-O0` flag to prevent the compiler from inlining our functions and performing other optimizing transforms that might make it difficult for us to make sense of the profile output. For profiling to provide us with meaningful information, we would need to compile at the level of optimization that we intend to finally use, with the understanding that mapping the profiler's output back to the source code in this case might involve some effort.)

2. Execute the program to generate the profile data file

   ```
   ./ourProgram
   ```

   The program executes and writes the profile data to `gmon.out`.

3. Run `gprof` on the profile data file to analyze the profile data

   ```
   gprof ourProgram gmon.out > profile.txt
   ```

The analysis is now contained in `profile.txt`. This file shows you how many times each function was executed, and how much time was spent in each function, and plenty of other detail. For our example program, we obtain:

```
 %   cumulative   self              self     total
time   seconds   seconds    calls  ms/call  ms/call  name
92.68      0.38     0.38        1   380.00   380.00  function2
 7.32      0.41     0.03        1    30.00    30.00  function1
```

We can see that most of the program runtime was spent in executing `function2`, with relatively little spent on `function1`. This tells us that it is most important to optimize the runtime of `function2`.

Further down in `profile.txt`, we see that `gprof` also tells us if the time taken by a function was spent inside the function or inside other function calls made by the function. Note that `gprof` can take several other arguments to produce different kinds of profiling analyses for the executable, including the number of times a certain line in the source code was executed.

3.2 Timing a Hotspot

Once the hotspots have been found, we need to measure their runtime for further analysis. Each hotspot must be timed separately with an appropriate timing routine. The general idea is the following:

1. Read the current time (start time) from the appropriate time source.
2. Execute the kernel/hotspot. Iterate an adequate number of times to obtain a meaningful value off the time source.

3. Read the current time (end time) from the appropriate time source.
4. Execution time of the kernel/hotspot = $\dfrac{\text{End time} - \text{Start time}}{\text{Number of iterations}}$.

We first discuss time sources, and reading the time from them; then we explain how to write a timing routine to get meaningful results.

Time functions. Depending on the system one is using, a variety of time sources to "get the current time" may be available:

– Most Unix systems define `gettimeofday()` to portably query the current time (as defined in IEEE Std 1003.1).

– ANSI C defines `ctime()` and `clock()` as portable ways of obtaining the current time.

– On Intel processors, the `rdtsc` instruction reads the time stamp counter which allows near-cycle accurate timing. On PowerPC processors, the `mfspr` instruction reads the time-base register.

Generally, portable time functions have much less precision than cycle-counter-based methods. The pros and cons of various timing methods are listed below:

Timer type	Advantages	Disadvantages
Wall clock; Unix: `gettimeofday()`	Simple to use, highly portable	Low resolution, does not account for background tasks
System timer; Unix: `time` command	Gives wall clock, user-cpu, and system-cpu times	Relatively low resolution
Hardware timestamp counter (discussed below)	High resolution, most precise and accurate	Does not account for background system load (effectively, wall clock time), best for kernels with short runtimes; non-portable

We give a simplified example of a timing macro based on `rdtsc` (a hardware timestamp counter) for a 32-bit Intel processor to be used with Microsoft VisualStudio:

```
typedef union
{ __int64 int64;
  struct {__int32 lo, hi;} int32;
} tsc_counter;

#define RDTSC(cpu_c)              \
{ __asm rdtsc                     \
  __asm mov (cpu_c).int32.lo,eax  \
  __asm mov (cpu_c).int32.hi,edx  \
}
```

The corresponding code sequence in GNU C looks slightly different:

```
typedef union
{ unsigned long long int64;
  struct {unsigned int lo, hi;} int32;
} tsc_counter;

#define RDTSC(cpu_c)                    \
  __asm__ __volatile__ ("rdtsc" :      \
  "=a" ((cpu_c).int32.lo),             \
  "=d"((cpu_c).int32.hi))
```

Timing routine. A timing routine calls the function that is to be timed without executing the original program. The objective is to isolate the kernel and measure the runtime of the kernel with the least disturbance and highest accuracy possible. A timing routine consists of the following steps:

- Initialize kernel-related data structures.
- Initialize kernel input data.
- Call kernel a few times to put microarchitectural components into steady state.
- Read current time.
- Call kernel multiple times to obtain an adequately precise value from the timing source used.
- Read current time.
- Divide the time difference by the number of kernel calls.

To obtain more stable timing results, one often has to run multiple timings and take the average or minimum value.

We give an example timing routine for an MMM function computing $C = C + AB$, assuming all matrices are square $N \times N$. The RDTSC macro is defined above.

```
double time_MMM(int N, double *A, double *B, double *C)
{ // init C
  for(i=0; i<N; i++)
    C[i] = 0.0;

  //  put microarchitecture in steady state
  MMM(A,B,C);

  // time
  RDTSC(t0);
  for(int i=0; i<TIMING_REPETITIONS; i++)
    MMM(A,B,C);
  RDTSC(t1);

  // compute runtime in cycles
```

```
    return (double)((t1.int64-t0.int64)/TIMING_REPETITIONS);
}
```

Known problems. The following problems may occur when timing numerical kernels:

- Too few iterations of the function to be timed are executed between the two time stamp readings, and the resulting timing is inaccurate due to poor timer resolution.
- Too many iterations are executed between the two time stamp readings, and the resulting timing is affected by system events.
- The machine is under load and the load has side effects on the measured program.
- Multiple timing jobs are executed concurrently, and they interfere with one another.
- Data alignment of input and output triggers cache problems.
- Virtual-to-physical memory translation makes timing irreproducible.
- The time stamp counter overflows and either triggers an interrupt or produces a meaningless value.
- Reading the timestamp counters requires hundred(s) of cycles, which itself affects the timing.
- The linking order of object files changes locality of static constants and this produces cache interference.
- The machine was not rebooted in a long time and the operating system state causes problems.
- The control flow in the numerical kernel being timed is data-dependent and the test data is not representative.
- The kernel is in-place (e.g., the input is a vector x and the output is written back to x), and the norm of the output is larger than the norm of the input. Repetitive application of the kernel leads to an exponential growth of the norm and finally triggers floating-point exceptions which interfere with the timing.
- The transform is timed with a zero vector, and the operating system is "smart," and responds to a request for a large zero-vector dynamic memory allocation by returning a special zero-valued copy-on-write virtual memory page. Read accesses to this "page" would be much faster than accesses to a page that is actually allocated, since this page is a special one maintained by the operating system for efficiency.

One needs to be very careful when timing numerical kernels to rule out these problems. Getting highly accurate, reproducible, stable timing results for the full range of problem sizes is often nontrivial. Note that small problem sizes may suffer from timer resolution issues, while large problem sizes with longer runtimes may suffer from the effects of intervening processes.

3.3 Analyzing the Measured Runtime

We now know how to calculate the theoretical peak performance and the memory bandwidth for our target platform, and how to obtain the operations count and the runtime for our numerical kernel. The next step is to use these to conduct a performance analysis that answers two questions:

- What is the limiting resource, i.e., is the kernel CPU-bound or memory-bound? This provides an idea of the various optimization methods that can be used.
- How efficient is the implementation with respect to the limiting resource? This shows the potential performance increase we can expect through optimization.

Normalization. To assess the runtime behavior of a kernel as function of the problem size, the runtime (or inverse runtime) has to be normalized with the asymptotic or exact operations count. For instance, FFT performance is usually reported in pseudo Mflop/s. This value is computed as $5n\log_2(n)$/runtime for DFT_n; $5n\log_2(n)$ is the operations count of the radix-2 FFT. For MMM, the situation is easier, since all currently relevant implementations have the exact operations count $2n^3$.

Let us now take a look at at Fig. 2. The Numerical Recipes FFT program achieves almost the same pseudo Mflop/s value, independently of the problem size. This means that all problem sizes run approximately at the same level of (in)efficiency. In contrast, the best code shows a wide variation of performance, generally at a much higher pseudo Mflop/s level. In particular, the performance ramps up to 25 Gflop/s and then drops dramatically. This means, the DFT becomes more and more efficient with larger problems, but only up to a certain size. Analysis shows that the breakdown occurs once the whole working set of the computation does not fit into the L2 cache any more and the problem switches from being CPU-bound to memory-bound, since the DFT's reuse is only $O(\log(n))$.

In contrast, Fig. 3 shows that MMM maintains the performance even for out-of-cache sizes. This is possible since MMM has a reuse of $O(n)$, higher than the DFT.

Fig. 2 shows that performance plots for high-performance implementations can feature unanticipated characteristics. That is especially true if the kernel changes behavior, for instance, if it slowly changes from being CPU-bound to memory-bound as the kernel size is varied.

Relative performance. Absolute performance only tells a part of the story. Comparing the measured performance to the theoretical peak performance shows how efficient the implementation is. A low efficiency for an algorithm with sufficiently high reuse means there is room for optimization.

We continue examining our examples from Fig. 2 and Fig. 3, with the target machine being a Core2 Extreme at 3 GHz.

In Fig. 2, Numerical Recipes is a single-core single-precision x87 implementation and thus the corresponding peak performance is 6 Gflop/s (see Table 3). As Numerical Recipes reaches around 1 pseudo Gflop/s it runs at about 16% of the peak. Note that if

SSE (4-way vector) instructions and all four cores are used, the peak performance goes up by a factor of 16. (see Table 3). The best scalar code achieves around 4 Gflop/s or about 60% of the x87 peak. The fastest overall code uses SSE and 4 cores and reaches up to 25 Gflop/s or 25% of the quad-core SSE peak.

In Fig. 2, the overall fastest code reaches and sustains about 42 Gflop/s or about 85% of the quad-core SSE2 peak. This is much higher than the DFT and also due to the higher degree of reuse in MMM compared to the DFT.

3.4 Exercises

1. **Performance analysis.** In this exercise, you will measure and analyze the performance of the naive implementations of MMM and the DFT from Exercise 1 in Section 2. The steps you will need to follow to complete this exercise are given below. For this exercise, use the hardware configuration of your computer as you determined in Exercise 2 on page 22.

   (a) *Determine your computer's theoretical peak performance.* The theoretical peak performance is the number of floating point operations that can be done in a second. This is found by determining the CPU clock speed, and examining the microarchitecture to look at the throughput of floating point operations. For instance, a CPU running at 900 MHz that can retire 2 floating point operations per cycle, has a theoretical peak performance of 1800 Mflop/s. If the type of instructions that the CPU can retire at the same rate includes FMA (fused multiply add) instructions, the theoretical peak would be 3600 Mflop/s (2 multiplies and 2 adds per cycle = 4 operations per cycle). For this exercise, do not consider vector operations.

   (b) *Measure runtimes.* Use your implementations of the MMM and DFT as completed in Exercise 1 on page 22. Use the techniques described in Section 3.2 to measure the runtimes of your implementations using at least two different timers.

   (c) *Determine performance and interpret results.*

      – Performance: The performance of your implementation is its number of floating point operations per unit time, measured in flop/s. For the DFT, the number of operations should be assumed $5n \log(n)$.

      – Percentage peak performance: This is simply the percentage of theoretical peak performance. For instance, if your measured code runs at 1.2 Gflop/s on a machine with a peak performance of 3.6 Gflop/s, this implies that your implementation achieves 33.3% of peak performance.

2. **Micro-benchmarks: mathematical functions.** We assume a Pentium compatible machine. Determine the runtime (in cycles) of the following computations (x, y are doubles) as accurately as possible:

- $y = x$
- $y = 7.12x$
- $y = x + 7.12$
- $y = \sin(x), x \in \{0.0, 0.2, 4.1, 170.32\}$
- $y = \log(x), x \in \{0.001, 1.00001, 10.65, 2762.32\}$
- $y = \exp(x), x \in \{-1.234e - 17, 0.101, 3.72, 1.234e25\}$

There are a total of 15 runtimes. Explain the results. The benchmark setup should be as follows:

(a) Allocate two vector doubles `x[N]` and `y[N]` and initialize all `x[i]` to be one of the values from above.

(b) Use

```
for(i=0; i<N; i++)
  y[i] = f(x[i]);
```

to compute `y[i] = f(x[i])` , with `f()` being one of the functions above and time this `for` loop.

(c) Choose N such that all data easily fits into L1 cache but there are enough iterations to obtain a reasonable amount of work.

(d) Use the x86 time stamp counter via the interface provided by `rdtsc.h`, as listed in Section 3.2.

To accurately measure these very short computations, use the following guidelines:

- Only time the actual work, leave everything else (initializations, timing related computations, etc.) outside the timing loop.
- Use the C preprocessor to produce a parameterized implementation to easily check different parameters.
- You may have to run your `for(N)` loop multiple times to obtain reasonable timing accuracy.
- You may have to take the minimum across multiple such measurements to obtain stable results. Thus, you might end up with three nested loops.
- You must put microarchitectural components into steady state before the experiment: variables where you store the timing results, the timed routine and the data vectors should all be loaded into the L1 cache, since cache misses might result in inaccurate timing results.
- Alignment of your data vectors on cache line sizes or page sizes can influence the runtime significantly.

- The use of CPUID to serialize the CPU before reading the RDTSC as explained in the Intel manual produces a considerable amount of overhead and may be omitted for this exercise.

4 Optimization for the Memory Hierarchy

In this section we describe methods for optimizations targeted at the memory hierarchy of a state-of-the-art computer system. We divide the discussion into four sections:

- Performance-conscious programming.
- Optimizations for cache.
- Optimizations for the registers and CPU.
- Parameter-based performance tuning.

We first overview the general concepts, and then apply them to MMM and the DFT later.

4.1 Performance-Conscious Programming

Before we discuss specific optimizations, we need to ensure that our code does not yield poor performance because it violates certain procedures fundamental to writing fast code. Such procedures are discussed in this section. It is important to note that programming for high performance may go to some extent against standard software engineering principles. This is justified if performance is critical.

Language: C. For high performance implementations, C is a good choice, as long as one is careful with the language features used (see below). The next typical choice for high-performance numerical code is Fortran, which tends to be more cumbersome to use than C when dynamic memory and dynamic data structures are used.

Object-oriented programming (C++) must be avoided for performance-critical parts since using object oriented features such as operator overloading and late binding incurs significant performance overhead. Languages that are not compiled to native machine code (like Java) should also be avoided.

Arrays. Whenever possible, one-dimensional arrays of scalar variables should be used. If the size of the object is not known at compile time, accessing an element in a higher-dimensional array requires multiple pointer dereferencing operations, as in this case `double **a` behaves differently from `double a[10][15]`. Hence, higher dimensional objects should be linearized: an $m \times n$ matrix should be represented by a vector of length mn, with the matrix element (i, j) mapped to the vector element $in + j$.

Records. Accessing `struct` elements may introduce additional index computation. It may also prevent compiler optimization, as a `struct` is a derived data type. Hence, complicated `struct` and `union` data types should be avoided. Multiple arrays should be favored over one array with `struct` entries. For example, to represent vectors of complex numbers, vectors of real numbers of twice the size should be used, with the real and imaginary parts appearing as pairs along the vector.

Dynamic data structures. Dynamically generated data structures like linked lists and trees must be avoided if the algorithm using them can be implemented on array structures instead. Heap storage must be allocated in large chunks, as opposed to separate allocations for each object.

Control flow. Unpredictable conditional branches are computationally expensive on machines with long pipelines. Hence, `while` loops and loops with complicated termination conditions must be avoided. `for` loops with loop counters and loop bounds known at compile-time must be used whenever possible. `switch`, `?:`, and `if` statements must be avoided in hot spots and inner loops, as they may be translated into conditional branches. For small, repetitive tasks, macros are a better choice than functions. Macros are expanded before compilation while the compiler must perform analysis on inline functions.

4.2 Cache Optimization

For lower levels in the memory hierarchy (L1, L2, L3 data cache, TLB = translation lookaside buffer) the overarching optimization goal is to reuse data as much as possible once brought in. The architecture of a set-associative cache (Fig. 6) suggests three major optimization methods that target different hardware restrictions.

– Blocking: working on data in chunks that fit into the respective cache level, to overcome restrictions due to cache capacity,
– Loop merging: merging consecutive loops that sweep through data into one loop to reuse data in the cache and hence make the best use of the restricted memory bandwidth, and,
– Buffering: copying data into contiguous temporary buffers to overcome conflict cache misses due to cache associativity.

The actual optimization process applies one or more of these ideas to some of the levels of the memory hierarchy. It is not always a good idea to apply all methods to all levels, as code complexity may increase dramatically.

Finally, the correct parameters for blocking and/or buffering on the targeted computer system have to be found. A good approach is to write the program parameterized, i.e., collect all parameters as named constants. Then it is easy to try different parameter settings by hand or using a script to find the variant that achieves the highest performance.

Blocking. The basic idea of blocking is to perform the computation in “blocks” that operate on a subset of the input data to achieve memory locality. This can be achieved in different ways. For example, loops in loop nests, like the triple loop MMM in Section 2.2 may be split and swapped (a transformation called tiling) so that the working set of the inner loops fits into the targeted memory hierarchy level, whereas the outer loop jumps from block to block. Another way to achieve blocking is to choose a recursive algorithm to start with. Recursive algorithms naturally divide a large problem into smaller problems that typically operate on subsets of the data. If designed and parameterized well, at some level all sub-problems fit into the targeted memory level and blocking is achieved implicitly. An example of such an algorithm is the recursive Cooley-Tukey FFT introduced later in in (3).

Loop merging. Numerical algorithms often have multiple stages. Each stage accesses the whole data set before the next stage can start, which produces multiple sweeps through the working set. If the working set does not fit into the cache this can dramatically reduce performance.

In some algorithms the dependencies do not require that *all* operations of a previous stage are completed before *any* operation in a later stage can be started. If this is the case, loops can be merged and the number of passes through the working set can be reduced. This optimization is essential for implementing high-performance DFT functions.

Buffering. When working on multi-dimensional data like matrices, logically close elements can be far from each other in linearized memory. For instance, matrix elements in one column are stored at a distance equal to the number of columns of that matrix. Cache associativity and cache line size get into conflict if one wants to hold, for instance, a small rectangular section of such a matrix in cache, leading to cache thrashing. This means the elements accessed by the kernel are mapped to the same cache locations and hence are moved in and out during computation.

One simple solution is to copy the desired block into a contiguous temporary buffer. That incurs a one-time cost but alleviates cache thrashing. This optimization is often called buffering.

4.3 CPU and Register Level Optimization

Optimization for the highest level in the memory hierarchy, the registers, is to some extent similar to optimizations for the cache. However it also needs to take into account microarchitectural properties of the target CPU. Current high-end CPUs are superscalar, out-of-order, deeply pipelined, feature complicated branch prediction units, and many other performance enhancing technologies. From a high-level point of view, one can summarize the optimization goals for a modern CPU as follows. An efficient C program should:

– have inner loops with adequately large loop bodies,

– have many independent operations inside an inner loop body,

– use automatic variables whenever possible,

– reuse loaded data elements to the extent possible,

– avoid math library function calls inside an inner loop if possible.

Some of these goals might conflict with others, or are constrained by machine parameters. The following methods help us achieve the stated goals:

– Blocking

– Unrolling and scheduling

– Scalar replacement

– Precomputation of constants

We now discuss these methods in detail.

Blocking. Register-level blocking partitions the data into chunks on which the computation can be performed within the register set. Only initial loads and final stores but no register spilling is required. Sometimes a small amount of spilling can be tolerated. We show the blocking of a single loop as example. Consider the example code below.

```
for(i=0; i<8; i++)
{ y[2*i]   = x[2*i] + x[2*i+1];
  y[2*i+1] = x[2*i] - x[2*i+1];
}
```

We block the `i` loop, obtaining the following code.

```
for(i1=0; i1<4; i1++)
  for(i2=0; i2<2; i2++)
  { y[4*i1+2*i2]   = x[4*i1+2*i2] + x[4*i1+2*i2+1];
    y[4*i1+2*i2+1] = x[4*i1+2*i2] - x[4*i1+2*i2+1];
  }
```

On many machines registers are only addressable by name but not indirectly via other registers (holding loop counters). In this case, once the data fits into registers, either loop unrolling or software pipelining with register rotation (as supported by Itanium) is required to actually take advantage of register-blocked computation.

Unrolling and scheduling. Unrolling produces larger basic blocks. That allows the compiler to apply strength reduction to simplify expressions. It decreases the number of conditional branches thus decreasing potential branch mispredictions and condition evaluations. Further it increases the number of operations in the basic block and allows the compiler to better utilize the register file. However, too much unrolling may increase the code size too much and overflow the instruction cache. The following code is the code above with unrolled inner loop `i2`.

```
for(i1=0; i1<4; i1++)
{ y[4*i1]   = x[4*i1] + x[4*i1+1];
  y[4*i1+1] = x[4*i1] - x[4*i1+1];
  y[4*i1+2] = x[4*i1+2] + x[4*i1+3];
  y[4*i1+3] = x[4*i1+2] - x[4*i1+3];
}
```

Unrolling exposes an opportunity to perform instruction scheduling. With unrolled code, it becomes easy to determine data dependencies between instructions. Issuing an instruction right after a preceding instruction that it is dependent upon will lead to the CPU pipeline being stalled until the former instruction completes. Instruction scheduling is the process of rearranging code to include independent instructions in between two dependent instructions to minimize pipeline stalls.

Scheduling large basic blocks with complicated dependencies may be too challenging for the compiler. In this case source scheduling may help. Source scheduling is the (legal) reordering of statements in the unrolled basic block. Different scheduling algorithms apply different rules, aiming at, e.g., minimizing distance between producer and consumer (which may potentially not be too short), and/or minimizing the number of live variables for each statement in the basic block. It is sometimes better to source schedule basic blocks and turn off aggressive scheduling by the compiler.

The number of registers, quality of the C compiler, and size of the instruction cache limit the amount of unrolling, that increases performance. Experiments show that on current machines, roughly 1,000 operations are the limit. Note, that unrolling always increases the size of the loop body, but not necessarily the instruction-level parallelism. Depending on the algorithm, more complicated loop transformations may be required. One example is the MMM, discussed later.

Scalar replacement. In C compilers, pointer analysis is complicated, and using even the simplest pointer constructs can prevent "obvious" optimizations. This observation extends to arrays with known sizes. It is very important to replace arrays that are fully inside the scope of an innermost loop by one automatic, scalar variable per array element. This can be done as the array access pattern does not depend on any loop variable and will help compiler optimization tremendously. As an example, consider the following code:

```
double t[2];
for(i=0; i<8; i++)
{ t[0] = x[2*i] + x[2*i+1];
  t[1] = x[2*i] - x[2*i+1];
  y[2*i]   = t[0] * D[2*i];
  y[2*i+1] = t[0] * D[2*i];
}
```

Scalarizing `t` will result in code that the compiler can better optimize:

```
double t0, t1;
for(i=0; i<8; i++)
{ t0 = x[2*i] + x[2*i+1];
  t1 = x[2*i] - x[2*i+1];
  y[2*i]   = t0 * D[2*i];
  y[2*i+1] = t1 * D[2*i];
}
```

The difference is that `t0` and `t1` are automatic variables and can be held in registers whereas the array `t` will most likely be allocated in memory, and loaded and stored from memory for each operation.

If an input value `x[i]` or precomputed data `D[i]` is reused it makes sense to first copy the value into an automatic variable (`xi` or `Di`, respectively), and then reuse the automatic variable.

```
double t0, t1, x0, x1, D0;
for(i=0; i<8; i++)
{ x0 = x[2*i];
  x1 = x[2*i+1];
  D0 = D[2*i];
  t0 = x0 + x1;
  t1 = x0 - x1;
  y[2*i]   = t0 * D0;
  y[2*i+1] = t1 * D0;
}
```

If the value of `y[i]` is used as source in operations like `y[i] += t0`, one should use scalar replacement for `y[i]`.

Precomputation of constants. In a CPU-bound kernel, all constants that are known ahead of time should be precompute at compile time or initialization time and stored in a data array. At execution time, the kernel simply loads the precomputed data instead of needing to invoke math library functions. Consider the following example.

```
for(i=0; i<8; i++)
  y[i] = x[i] * sin(M_PI * i / 8);
```

The program contains an function call to the math library in the inner loop. Calling `sin()` can cost multiple thousands of cycles on modern CPUs. However, all the constants are known before entering the kernel and thus can be precomputed.

```
static double D[8];
void init()
{ for(int i=0; i<8; i++)
    D[i] = sin(M_PI * i / 8);
}

...
// in the kernel
for(i=0; i<8; i++)
  y[i] = x[i] * D[i];
```

The initialization function needs to be called only once. If the kernel is used over and over again, precomputation results in enormous savings. If the kernel is used only once, chances are that performance does not matter.

4.4 Parameter-Based Performance Tuning and Program Generation

Many of the optimizations for the memory hierarchy discussed above have inherent degrees of freedom such as the block size for blocking or the degree of unrolling the code. While it may be possible to derive a reasonable estimation of these parameters, the complexity of modern microarchitecture makes an exact prediction impossible. In fact, often the best value may come as a surprise to the programmer. As a consequence, it makes sense to perform an empirical search to find those parameters. This means creating the variants, ideally through a set of scripts, through parameterized coding (for instance, defining all parameters as C preprocessor constants in a separate header file), or through other program generation techniques, and measuring their performance to find the best choice. Since the result may depend on the target platform, the search should be repeated for each new platform.

This parameter-based performance optimization is one of the techniques used in recent research on automatic performance tuning [14].

However, parameter based tuning is inherently not extensible in the sense that new forms of code or algorithm restructuring cannot be incorporated easily. Examples could be transformations for various forms of parallelism. A better solution than parameter-based tuning may be properly designed domain-specific languages used in tandem with rewriting systems. We will see the difference between these two approaches later in Section 5.4 and 6.6 where we discuss program generation for MMM and the DFT.

5 MMM

In this section, we optimize matrix-matrix multiplication (MMM) for the memory hierarchy. We explain the optimizations implemented by the ATLAS [13], and organize the steps as in Section 4. ATLAS is a program generator for MMM and other BLAS routines and also performs other optimizations not discussed here. It is introduced in Section 5.4.

Our presentation closely follows the one in Yotov et al. [75], which presents a model-based version of ATLAS.

For the rest of this section, we will assume the dimensions of the input matrices A and B to be $N \times K$ and $K \times M$ respectively, which implies an $N \times M$ output matrix C. For simplicity, we will further assume that various optimization parameters are perfectly divisible by these dimensions whenever such a division is necessary. The computation considered is $C = C + AB$.

Naive Implementation. Matrix-matrix multiplication (MMM), as defined in Section 2.2, is naively implemented using the triple loop shown below. We use 2D array notation (for instance, `C[i][j]`) to keep the code more readable. However, in an implementation where the matrix sizes are not known at compile time, one should resort to a linearized representation of C, A, and B (see Section 4.1).

```
// K, M, N are compile-time constants
double C[N][M], A[N][K], B[K][M];
// Assume C is initialized to zero
for(i=0; i<N; i++)
  for(j=0; j<M; j++)
  { for(k=0; k<K; k++)
      C[i][j] += A[i][k] * B[k][j];
  }
```

The C language stores two-dimensional arrays in row-major order. Therefore, a cache miss to a (memory aligned) matrix element causes that element and adjacent elements in the same row being loaded into one cache line of the cache (see Fig. 7). Thus, accessing a large matrix by rows is cache efficient, while accessing it by columns is not.

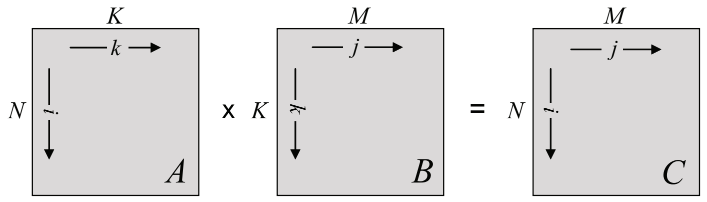


Fig. 9. Data access pattern for the naive MMM.

Fig. 9 illustrates the data access pattern of the naive implementation. From this figure, we see the output locality of the computation: all accesses to each element in C are consecutive, and C is completed element by element, row by row. However, unless all input and output arrays fit into the cache, the naive implementation has poor locality with respect to A and B.

We analyze the naive implementation by counting the number of cache misses. We assume a cache line size of 64 bytes, or 8 (double precision) floating point values, and that N is large with respect to the cache size. To compute the first entry in C, we need to access the entire first row of A and the entire first column of B. Accessing a row of A results in $N/8$ misses (one for each group of 8) due to the row-major storage order, while accessing a column of B results in a full N misses, yielding a total of $(9/8)N$ misses for the first entry in C.

To analyze the computation of the second entry of C, we first observe that the parts of A and B that will be accessed first are not in the cache. That is, since N is much larger than the cache, the first few elements of the first row of A were in cache but were eventually overwritten. Similarly, the first elements of the second column of B were already in cache (each element shared a cache line with its neighbor in the first column) but also have been overwritten. This is illustrated in Fig. 10, which shows in gray the parts of A and B that are in cache after the first entry of C is computed. Consequently, the number of misses involved in computing the second entry (and every subsequent entry of C), produces also $(9/8)N$ misses. Therefore, the total number of misses generated by this

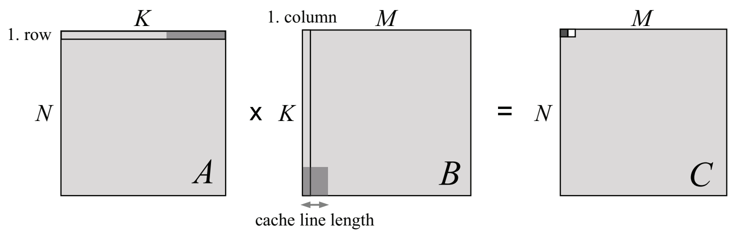


Fig. 10. The state of the cache at the end of computation of the first element of C (small black square) is shown. Areas of the input matrices marked in gray are cache resident at this point. The next element of C to be computed is shown as small white square.

algorithm (for the N^2 entries in C) is $(9/8)N^3$. In summary, there is no reuse and no neighbor use, a problem resolved to the extent possible by the optimizations in the next sections.

5.1 Cache Optimization

Blocking. One of the most important optimizations for MMM (and linear algebra problems in general) is blocking, as introduced in Section 4.2. Blocking involves performing the addition and multiplication operations on *blocks* of the original matrix, instead of individual elements. The idea is to increase locality by restricting the computation at any point to work on small chunks that fit entirely into the cache. We will also see that blocking essentially increases reuse and neighbor use, the concepts previously presented in Section 2.4.

The compiler loop transformation that implements blocking is known as *tiling* [76, 13, 75]. Blocking or tiling the MMM for each level of the memory hierarchy involves adding three more nested loops to the basic triple loop implementation. The code for the MMM blocked for one memory level with block size N_B follows.

```
// MMM loop nest (j, i, k)
for(i=0; i<N; i+=NB)
  for(j=0; j<M; j+=NB)
    for(k=0; k<K; k+=NB)
      // mini-MMM loop nest (i0, j0, k0)
      for(i0=i; i0<(i + NB); i0++)
         for(j0=j; j0<(j + NB); j0++)
           for(k0=k; k0<(k + NB); k0++)
             C[i0][j0] += A[i0][k0] + B[k0][j0];
```

Fig. 11 shows the data access pattern of blocking for the cache. The three additional innermost loops cause each matrix to be divided into blocks of size $N_B \times N_B$. Notice the similarity in the access pattern to the naive implementation, except at the block level instead of at the element level.

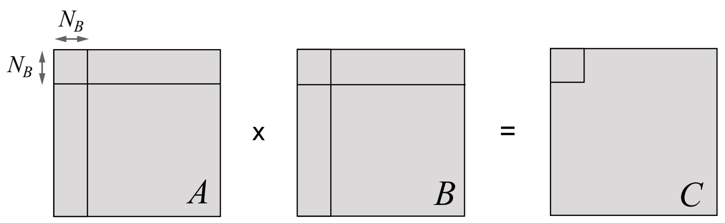


Fig. 11. Blocking for the cache: mini-MMMs.

We now analyze this version of the MMM to determine the impact on the number of cache misses. We assume that the block size is larger than the cache line size, and for now that several blocks can fit into the cache. This implies that accessing a block results only in $N_B^2/8$ misses, regardless of the access sequence.

Computing the first *block* of C requires the first block row of A, and the first block column of B. This results in $(N_B^2/8 + N_B^2/8)(N/N_B)$ cache misses. Similar to the reasoning used in the analysis of the naive version, computing each block of C results in the same amount of misses, and therefore, the total number of misses generated by this algorithm (for the $(N/N_B)^2$ blocks in C) is $N^3/(4N_B)$, which is significantly less than the $(9/8)N^3$ misses in the naive version.

We call the smaller blocks operations mini-MMMs, following [75]. N_B is an optimization parameter that must be chosen such that the working set of the mini-MMM fits entirely into the cache. A simple translation of our assumption that blocks from the two input and output matrices (our *working set*) fit into a fully associative cache is expressed by the following equation: $3N_B^2 \leq C_s$, where C_s is the cache size. ATLAS determines N_B by searching and trying different arbitrary values and picking the one that results in the best performance.

In contrast, [75] use a model based approach, and chooses N_B based directly on cache parameters. Their careful examination of the data access pattern of the blocked MMM reveals that the working set at a finer granularity consists only of a single element in C (since each element in C is reused completely by the innermost `k0` loop before it moves on to the next element), a single row of A (since a row is fully reused before the program moves on to the next row), and the entire B. Therefore, the following relationship needs to hold: $N_B^2 + N_B + 1 \leq C_s$. Thus, a good choice for N_B is the largest value that satisfies this inequality.

Blocking for MMM works because it increases cache reuse and neighbor use, our guiding principles discussed in Section 2. Cache reuse is increased because once a block is brought into the cache, it is used several times before being overwritten. Neighbor use is increased for the input matrix B, since all elements in the cache line are used before eviction.

Typically, MMM is blocked for the L1 cache but blocking for the L2 cache may be superior in certain cases [75].

An additional optimization that can be done for the cache is to exchange the i and the j loops, depending upon the relative sizes of the A and B matrices.

Loop merging. Loop merging is not applicable to the MMM.

Buffering. Buffering (also known as copying) for MMM is applicable for large sizes. The basic idea behind buffering is to copy tiles of the input and output matrices into sequential order in memory to minimize cache conflict misses (and TLB misses if the matrices span multiple pages), inside each mini-MMM. The following code illustrates buffering. The matrix B is fully buffered at the beginning since it is accessed in full during each iteration of the outermost i loop. Vertical panels of A are used during each iteration of j, and are buffered just before the j loop begins. Finally, in some cases, it might be beneficial to copy a single tile of C before the k loop, since a single tile is reused by each iteration of the k loop. Note that the benefits of buffering have to outweigh the costs, which might not hold true for very small or very large matrices.

```
// Buffer full B here
for(i=0; i<M; i+=NB)
  // Buffer a panel of A here
  for(j=0; j<N; j+=NB)
    // Copy a block (tile) of C here
    for(k=0; k<K; k+=NB)
        // mini-MMM loop nest as before (i0, j0, k0)
        ...
```

5.2 CPU and Register Level Optimization

We now look at optimizing the MMM for the CPU. We continue with our MMM example from the previous section.

Blocking. Blocking for the registers looks similar to blocking for the cache. Another set of nested triple loops is added. The resulting code is shown below:

```
// MMM loop nest (j, i, k)
for(i=0; i<N; i+=NB)
  for(j=0; j<M; j+=NB)
    for(k=0; k<K; k+=NB)
      // mini-MMM loop nest (i0, j0, k0)
      for(i0=i; i0<(i + NB); i0+=MU)
        for(j0=j; j0<(j + NB); j0+=NU)
          for(k0=k; k0<(k + NB); k0+=KU)
            // micro-MMM loop nest (j00, i00)
            for(k00=k0; k00<=(k0 + KU); k00++)
              for(j00=j0; j00<=(j0 + NU); j00++)
                for(i00=i0; i00<=(i0 + MU); i00++)
                  C[i00][j00]+=A[i00][k00]*B[k00][j00];
```

Note that the innermost loop nest now has the loop order `kij`; this is explained later. As Fig. 12 shows, each mini-MMM is now computed by blocking it into a sequence

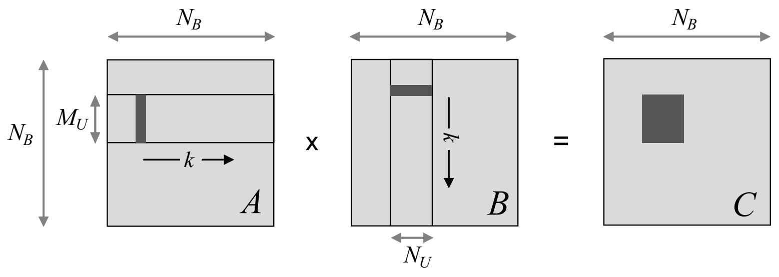


Fig. 12. mini-MMMs and micro-MMMs (from [75]).

of micro-MMMs. Each micro-MMM multiplies an $M_U \times 1$ block of A by a $1 \times N_U$ block of B, with the output being a $M_U \times N_U$ block of C. At this level of blocking, we have a degree of freedom in choosing M_U and N_U (The K_U parameter controls the degree of unrolling, and is discussed soon). These parameters must be chosen so that a micro-MMM fits into register space (thus avoiding register spills).

ATLAS searches over arbitrary values for these parameters to choose the ones that result in the fastest code. In [75], with a reasoning that is similar to the one used in choosing N_B in the previous section, selects these parameters based on the inequality $M_U + N_U + (M_U \times N_U) \leq N_R$, where N_R is the number of data (integer or floating point) registers. This equality is then further refined.

Locality is not the only objective of blocking for register space. Note that in the code above, the micro-MMM have a loop order of `kij`. While this reduces output locality, it also provides better instruction level parallelism (all the $M_U N_U$ addition/multiplication pairs are independent) when combined with loop unrolling discussed next.

Unrolling and scheduling. Loop unrolling and scheduling, as discussed in Section 4.3, can be used to further optimize MMM. We unroll the two innermost loops to get $M_U \times N_U$ additions and multiplications. Note that these instructions are of the form $C+ = AB$. As mentioned in [21], such an instruction will not execute well on machines without a fused multiply-add unit, since the addition is dependent on the multiplication, and will cause a pipeline stall until the multiplication is completed. Thus, it may be beneficial to separate the addition and the multiplication operations here, and schedule them with unrelated intervening instructions to minimize pipeline stalls.

The $k00$ loop can also be unrolled completely to reduce loop overhead. K_U controls the degree of unrolling, and is chosen so that the fully unrolled loop body (of the $k0$ loop) still fits into the L1 instruction cache.

Scalar replacement. When the innermost loops are unrolled, each array element appears multiple times in the unrolled code. For the reasons discussed earlier in Section 4.3, replacing array references by scalar variables in unrolled code enables compiler optimizations to work better. As the MMM has a good reuse ratio, references to input arrays are also replaced by first copying the value to automatic variables and then reusing the automatic variable.

Precomputation of constants. Since the MMM does not have constants that can be precomputed, this optimization does not apply.

5.3 Parameter-Based Performance Tuning

The above discussion identifies several parameters that can be used for tuning. ATLAS performs this tuning automatically by generating the variants and selecting the fastest using a search procedure.

Blocking for cache. N_B is the main optimization parameter used to control the block size of the mini-MMMs. If several levels of blocking are desired, additional blocking parameters arise.

Blocking for registers. When blocking for the registers, M_U, and N_U are the main tunable parameters, and must be chosen such that the micro-MMM does not produce register spills. K_U specifies the degree of unrolling and should be chosen as large as possible without overflowing the instruction cache.

Besides that, several other parameters can be identified for performance tuning and platform adaptation [21, 75].

5.4 Program Generation for MMM: ATLAS

The parameters shown in the previous section are only a small subset of all the parameters that can be used to tune the MMM. In theory, searching over the space of all tunable parameters will lead to the fastest code. Obviously, such a search would take an impractical amount of time to complete due to the vast search space. The best approach in this scenario is to prune the search space in a reasonable way and to automate the search over the remaining space. This in essence is the approach followed by ATLAS [21], which is briefly discussed in this section. In terms of the language previously used in this tutorial, ATLAS generates a mini-MMM with the highest performance, which is then used as a kernel in a generic MMM function.

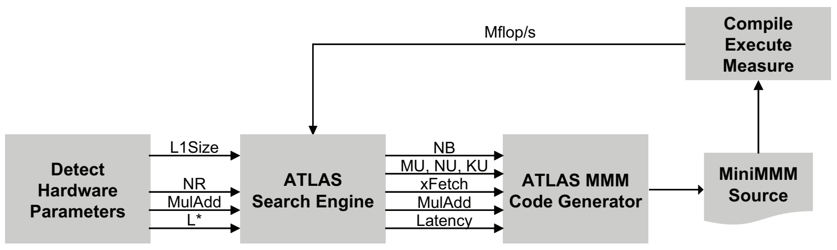


Fig. 13. Architecture of ATLAS (from [75]).

Fig. 13 shows the architecture of ATLAS. When ATLAS is first installed on a platform, it runs a set of micro-benchmarks to determine a set of hardware parameters, including the L1 cache size and the number of registers N_R. These parameters are then used to prune the originally unbounded search space to a finite one. ATLAS then proceeds by searching the space of possible mini-MMMs using a feedback loop. In this feedback loop, a search engine decides on the parameters that specify a mini-MMM, the corresponding code is generated, its performance evaluated, and the next set of parameters is tried.

Since the search space is too large, ATLAS uses an *orthogonal line search* to find the optimal values for the set of parameters it searches over. Given a function $y = f(x_1, x_2, \ldots, x_n)$ to optimize, orthogonal line search determines an approximation by solving a *sequence* of n 1-dimensional optimization problems, where each problem corresponds to one of the n parameters. When optimizing for x_i, the set of optimal values already found for $x_1 \ldots x_{i-1}$ are used, and *reference values* are used for the remaining parameters $x_{i+1} \ldots x_n$. ATLAS provides the parameter sequence, and ranges and reference values for each of the parameters, using a combination of built-in defaults and the determined microarchitectural parameters.

It has been shown that a suitably designed model, based on a detailed understanding of the microarchitecture, can replace the search in ATLAS to find the best parameters deterministically [75].

Discussion. ATLAS has been very successful in generating very fast MMM code for many architectures and has been widely used. In fact, ATLAS, and its predecessor PHiPAC [22], were the first efforts on automatic performance tuning in the area of numerical computing; as such, it raised awareness to the increasing difficulty of deciding on coding choices and achieving high performance in general on machines with deep memory hierarchies. As we have seen, in this case, using program generation is crucial to efficiently evaluate the many possible choices of parameters.

However, since ATLAS is based on (properly chosen) parameters it is not clear how to extend its approach to novel architectural paradigms such as vector instructions, multicore processing, or others. To date, these are not supported by ATLAS. We argue that the reason is the lack of an internal domain-specific language that can express all the necessary transformations at a higher abstraction level, which also enables the inclusion of new transformations. This is the approach taken by Spiral, a program generator for the domain of linear transforms discussed later in Section 6.6.

5.5 Exercises

1. **Mini-MMM.** The goal of this exercise is to implement a fast mini-MMM to multiply two square $N_B \times N_B$ matrices (N_B is a parameter), which is then used within an MMM.

   (a) *Based on definition.* Use your naive implementation of the MMM as mini-MMM (code from Exercise 1 in Section 2.

(b) *Register blocking.* Block into micro MMMs with $M_U = N_U = 2,\ K_U = 1$. The inner triple loop must have the `kij` order. Manually unroll the innermost `i` and `j` loops and schedule your code to perform alternating additions and multiplications (one operation per line of code). Perform scalar replacement on this unrolled code manually.

(c) *Unrolling.* Unroll the innermost `k` loop by a factor of 2 and 4 ($K_U = 2, 4$, which doubles and quadruples the loop body) and again do scalar replacement. Assume that 4 divides N_B.

(d) *Performance plot, search for best block size.* Determine the L1 data cache size C (in doubles, i.e., 8B units) of your computer. Measure the performance (in Mflop/s) of your four codes for all N_B with $16 \leq N_B \leq \min(80, \sqrt{C})$ with 4 dividing N_B. Create a plot with the x-axis showing N_B, and y-axis showing performance. The plot should contain 4 lines: one line for each of the programs (MMM by definition, register blocking, and unrolling by a factor of 2 and 4). Discuss the plot, including answers to the following questions: which N_B and which code yields the maximum performance? What is the percentage of peak performance in this case?

(e) *Loop order.* Does it improve if in the best code so far you switch the outermost loop order from `ijk` to `jik`? Create a plot to show the answer.

(f) *Blocking for L2 cache.* Consider now your L2 cache instead. What is its size (in doubles)? Can you improve the performance of your fastest code so far by further increasing the block size N_B to block for L2 cache instead? Answer through an appropriate experiment and performance plot.

2. **MMM.**

(a) Implement an MMM for multiplying two square $N \times N$ matrices assuming N_B divides N, blocked into $N_B \times N_B$ blocks. Use your best mini-MMM code from Exercise 1.

(b) Create a performance plot comparing this implementation and the implementation based on definition above for an interesting range of N (up to sizes where the matrices do not fit into the L2 cache). Plot the size N on the x-axis, against the performance (in Mflop/s or Gflop/s) on the y-axis.

(c) Analyze and discuss the plot.

6 DFT

In this section we describe the design and implementation of a high-performance function to compute the FFT. The approach we must take is different from the one taken to optimize the MMM in Section 5: we do not start with a naive implementation that is transformed into an optimized form, but design the code from scratch. This is due to the more complex structure of the available FFT algorithms. Note that, in contrast to MMM, an implementation based on the definition of the DFT is not competitive.

The first main problem is the choice of a suitable FFT algorithm, since many different variants are available that differ vastly in structure. It makes no sense to start with the wrong FFT algorithm and optimize the implementation step by step. In particular, when targeting a machine with a memory hierarchy, starting the optimization with the iterative radix-2 FFT used in Numerical Recipes (Section 2.3) is suboptimal since it requires $\log_2(\text{input size})$ many sweeps through the input data, which results in poor cache locality. Further, no unrolled and optimized basic block is used for optimal register performance.

In our discussion below we design a recursive radix-4 FFT implementation. Generalization to a mixed-radix recursive implementation is relatively straightforward in concept, but technically complex. The optimization steps taken follow to a large extent the design of FFTW 2.x [9]. FFTW uses a program generator in addition, to automatically implement optimized unrolled basic blocks [23].

In all our DFT code examples the (complex) data is assumed to be stored in interleaved complex double-precision arrays (alternating real and imaginary parts of the vector elements). We pass around pointers of type `double`, and two neighboring `double` elements are one complex number. All strides are relative to complex numbers.

6.1 Background

In this section we provide background on the DFT and FFTs. We explain these algorithms using the Kronecker product formalism. We start with restating the DFT definition from Section 2.3. For code readability we denote the size of the input vector with N. As usual, matrices are written as $A = [a_{k,\ell}]$, where $a_{k,\ell}$ are the matrix elements. An index range for k, ℓ may be given in the subscript.

Definition. The discrete Fourier transform (DFT) of a complex input vector x of length N is defined as the matrix-vector product

$$y = \mathrm{DFT}_N\, x, \quad \mathrm{DFT}_N = [\omega_N^{k\ell}]_{0 \le k,\ell < N}, \quad \omega_N = e^{-2\pi i/N}.$$

Kronecker product formalism. We describe fast algorithms for the DFT using the Kronecker product formalism [5]. There are several reasons for using this formalism: First, the representation is visual and index free and hence readable by humans. Second, it is easy to translate algorithms expressed this way directly into code, as we shall see later. Third, in this representation, algorithm variants are easily obtained by both inserting recursions into each other and manipulating algorithms to *match* them to a specific hardware architecture. For instance, the algorithms can be mapped to vector and multicore architectures this way [26, 25].

These are also the reasons why the program generator Spiral (explained in Section 6.6) uses this formalism as its internal domain-specific language.

We define I_n as the $n \times n$ identity matrix. The tensor (or Kronecker) product of matrices is defined as

$$A \otimes B = [a_{k,\ell}B]_{k,\ell} \quad \text{with} \quad A = [a_{k,\ell}]_{k,\ell}\,.$$

In particular,

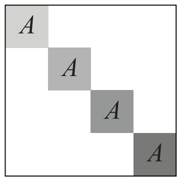
$$I_n \otimes A = \begin{bmatrix} A & & & \\ & A & & \\ & & \ddots & \\ & & & A \end{bmatrix}$$

is block-diagonal. We also introduce the iterative direct sum

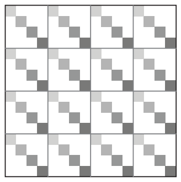
$$\bigoplus_{i=0}^{n-1} A_i = \begin{bmatrix} A_0 & & & \\ & A_1 & & \\ & & \ddots & \\ & & & A_{n-1} \end{bmatrix},$$

which generalizes $I_n \otimes A$.

We visualize $I_4 \otimes A$ below; the four As are shown with different shades of gray.

$I_4 \otimes A =$  (1)

Now we look at the tensor product $A \otimes I_n$. This matrix also contains n blocks of A, but they are spread out and interleaved at stride n. This is best understood by visualization: the equivalent of (1) is

$A \otimes I_4 =$  (2)

where we assume that A is 4×4. All elements with the same shade of gray taken together constitute one A, so the matrix again contains four As. The pattern shows that multiplying (2) to an input vector x is equivalent to multiplying A to four subvectors of x, extracted at stride 4, and writing the result into the same locations.

The stride permutation matrix L_m^{mn} permutes an input vector x of length mn as

$$in + j \mapsto jm + i, \quad 0 \le i < m,\ 0 \le j < n.$$

If x is viewed as an $n \times m$ matrix, stored in row-major order, then L_m^{mn} performs a transposition of this matrix.

Recursive FFT. Using the above formalism, the well-known Cooley-Tukey FFT in its recursive form can be written as a factorization of the DFT_N matrix into a product of sparse matrices. That is, for $N = mn$,

$$\mathrm{DFT}_{mn} = (\mathrm{DFT}_m \otimes I_n) D_{m,n} (I_m \otimes \mathrm{DFT}_n) L_m^{mn}. \tag{3}$$

Here $D_{m,n}$ is the diagonal “twiddle” matrix defined as

$$D_{m,n} = \bigoplus_{j=0}^{m-1} \mathrm{diag}(\omega_{mn}^0, \omega_{mn}^1, \ldots, \omega_{mn}^{n-1})^j. \tag{4}$$

Equation (3) computes a DFT of size mn in four steps. First, the input vector is permuted by L_m^{mn}. Second, m DFTs of size n are computed recursively on segments of the vector. Third, the vector is scaled element wise by $D_{m,n}$. Lastly, n DFTs of size m are computed recursively at stride m.

The recursively called smaller DFTs are computed similarly until the base case $n = 2$ is reached, which is computed by definition using an addition and a subtraction:

$$\mathrm{DFT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{5}$$

In summary, (3) and (5) are sufficient to compute DFTs of arbitrary two-power sizes. To compute DFTs of other sizes, other FFT algorithms are required [5].

Algorithms and formulas. There is a degree of freedom in applying (3) to recursively compute a DFT, namely in factoring the given DFT input size N. For instance one can factor $8 \rightarrow 2 \times 4 \rightarrow 2 \times (2 \times 2)$ using two recursive applications of (3). The complete FFT algorithm for this factorization could then be written as the following *formula*:

$$\mathrm{DFT}_8 = (\mathrm{DFT}_2 \otimes I_4) D_{8,4} \big(I_2 \otimes (\mathrm{DFT}_2 \otimes I_2) D_{4,2} (I_2 \otimes \mathrm{DFT}_2) L_2^4\big) L_2^8. \tag{6}$$

Direct implementation. A straightforward implementation of (3) can be easily obtained since the occurring matrix formulas have a direct interpretation in terms of code as shown in Table 4. The implementation of (3) would hence have four steps corresponding to the four factors in (3).

Observe in Table 4 that the multiplication of a vector by a tensor product containing an identity matrix can be computed using loops. The working set for each of the m iterations of $y = (I_m \otimes A_n)x$ (see (1)) is a contiguous block of size n and the base address is increased by n between iterations. In contrast, the working sets of size m of the n iterations of $y = (A_m \otimes I_n)x$ (see (2)) are interleaved, leading to stride n within one iteration and a unit stride base update across iterations.

Cost analysis. Computing the DFT using (3) requires, independent of the recursion strategy, $n \log_2(n) + O(n)$ complex additions and $\frac{1}{2} n \log_2(n) + O(n)$ complex multiplications.

The exact number of real operations depends on the chosen factorizations of n and is at most $5n \log_2(n) + O(n)$.

formula	code

$y = (A_n B_n)x$	`t[0:1:n-1] = B(x[0:1:n-1]);`
	`y[0:1:n-1] = A(t[0:1:n-1];)`

$y = (I_m \otimes A_n)x$	`for(i=0;i<m;i++)`
	`  y[i*n:1:i*n+n-1] =`
	`    A(x[i*n:1:i*n+n-1]);`

$y = (A_m \otimes I_n)x$	`for(i=0;i<m;i++)`
	`  y[i:n:i+m-1] =`
	`    A(x[i:n:i+m-1]);`

$y = \left(\bigoplus_{i=0}^{m-1} A_n^i\right) x$	`for(i=0;i<m;i++)`
	`  y[i*n:1:i*n+n-1] =`
	`    A(i, x[i*n:1:i*n+n-1]);`

$y = D_{m,n}x$	`for(i=0;i<m*n;i++)`
	`  y[i] = Dmn[i]*x[i];`

$y = L_m^{mn}x$	`for(i=0;i<m;i++)`
	`  for(j=0;j<n;j++)`
	`    y[i+m*j]=x[n*i+j];`

Table 4. Translating formulas to code. x denotes the input and y the output vector. The subscript of A and B specifies the size of the (square) matrix. We use Matlab-like notation: `x[b:s:e]` denotes the subvector of x starting at `b`, ending at `e` and extracted at stride `s`.

Iterative FFTs. The original FFT by Cooley and Tukey [77] was not the recursive algorithm (3), but an iterative equivalent and for $N = 2^n$. It can be obtained by expanding the DFT recursively always using the factorization $N = 2 \cdot N/2$, and then rearranging the parentheses and fusing adjacent permutations. The result is the iterative FFT

$$\mathrm{DFT}_N = \left(\prod_{i=1}^{k} (I_{2^{i-1}} \otimes \mathrm{DFT}_2 \otimes I_{N/2^i}) D'_{N,i} \right) R_N, \tag{7}$$

where the $D'_{N,i}$ are diagonal matrices and R_N is the bit-reversal permutation [5]. Numerical Recipes implements a variant of (7), shown in Section 2.3.

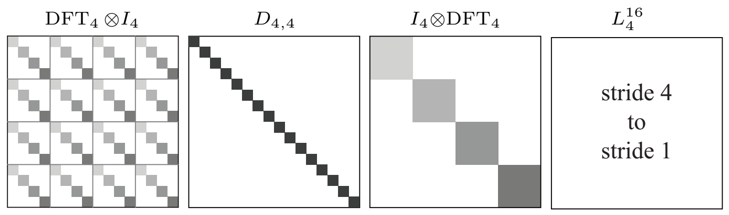
6.2 Cache Optimization

In this section we derive the recursive skeleton and the kernel specification for our DFT implementation.

Blocking. Blocking a DFT algorithm is done by choosing the recursive Cooley-Tukey FFT algorithm (3) as starting point instead of the iterative FFT used by the Numerical Recipes code in Section 2.3. The block size is the chosen *radix* m in (3), which is a degree of freedom. We assume a radix-4 implementation with $N = 4^n$, i.e., we factor $N = 4 \cdot 4^{n-1}$. The corresponding recursion is

$$\mathrm{DFT}_{4^n} = (\mathrm{DFT}_4 \otimes I_{4^{n-1}}) D_{4,4^{n-1}} (I_4 \otimes \mathrm{DFT}_{4^{n-1}}) L_4^{4^n}. \tag{8}$$

We visualize (8) below for $n = 2$. We see four stages, corresponding to the four factors in the matrix factorization.

$\mathrm{DFT}_{16} =$ 

(9)

For $n > 1$ our implementation will recursively apply (8) to the terms $\mathrm{DFT}_{4^{n-1}}$ in the right side of (8). The terms DFT_4 are recursion leaves and not implemented using (8). We will discuss their implementation in Section 6.3.

This recursion is *right-expanded*—the first stage gets recursively expanded while the second stage uses radix-4 kernels. Right-expanded recursive implementations have superior data locality as only a small amount of temporary storage is needed and the second stage can be implemented in-place.

Loop merging. A naive implementation of (8) leads to a recursive function with four stages (corresponding to the four matrix factors) and thus four sweeps through the data. However, the stride permutation $L_4^{4^n}$ is just a data reordering and thus is a candidate for loop merging. Similarly, the twiddle factor matrix $D_{4,4^{n-1}}$ is a diagonal matrix and can be merged with the subsequent stage.

We now sketch the derivation of a recursive implementation of (8). We partition (8) into two expressions as

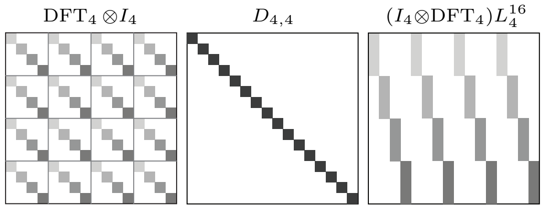
$$\mathrm{DFT}_{4^n} = \Big((\mathrm{DFT}_4 \otimes I_{4^{n-1}}) D_{4,4^{n-1}} \Big) \cdot \Big((I_4 \otimes \mathrm{DFT}_{4^{n-1}}) L_4^{4^n} \Big), \tag{10}$$

which become two stages (instead of four) in the recursive function

```
  void DFT(int N, double *Y, double *X);
```

that implements (8).

For $n = 2$ we visualize the merging of the stride permutation with the adjacent tensor product, $\mathrm{DFT}_4 \otimes I_4$, in (11) below. The merging of the diagonal $D_{4,4}$ with the adjacent tensor product $I_4 \otimes \mathrm{DFT}_4$ cannot easily be visualized.

$\mathrm{DFT}_{16} =$  (11)

$\mathrm{DFT}_4 \otimes I_4$
$D_{4,4}$
$(I_4 \otimes \mathrm{DFT}_4) L_4^{16}$

The first stage of (10), $y = (I_4 \otimes \mathrm{DFT}_{4^{n-1}}) L_4^{4^n} x$, is handled as follows. According to Table 4, the tensor product $I_4 \otimes \mathrm{DFT}_{4^{n-1}}$ alone is translated into a loop with 4 iterations. The same is true for $(I_4 \otimes \mathrm{DFT}_{4^{n-1}}) L_4^{4^n}$; only, as (11) shows, the input is now read at stride 4 but the output is still written at stride 1. This means that the corresponding DFT function needs to have the stride as an additional parameter and has to be implemented out-of-place, i.e., x and y need to be different memory regions. Hence it is of the form

```
  void DFT_rec(int N, int n, double *Y, double *X, int s)
```

We pass n together with N to avoid computing the logarithm.

Now the function `DFT` above just becomes a special case of `DFT_rec` and can hence be implemented using a C macro (`log4()`, computes n from 4^n):

```
#define DFT(N, Y, X) DFT_rec(N, log4(N), Y, X, 1)
```

For $N = 4$, we reach the leaf of the recursion and call a base case kernel function.

```
  void DFT4_base(double *Y, double *X, int s);
```

The second stage, $y = (\mathrm{DFT}_4 \otimes I_{4^{n-1}}) D_{4,4^{n-1}} x$, first scales the input by a diagonal matrix and then sweeps with a DFT_4 kernel over it, applied at a stride. More precisely, DFT_4 operates on x_j, $x_{j+4^{n-1}}$, $x_{j+2\cdot 4^{n-1}}$, and $x_{j+3\cdot 4^{n-1}}$, where j is the loop iteration number.

Again, we merge these two steps, this time by replacing the DFT_4s in $\mathrm{DFT}_4 \otimes I_{4^{n-1}}$ by $\mathrm{DFT}_4\, D_j$, where D_j is a 4×4 diagonal matrix containing the proper diagonal elements from $D_{4,4^{n-1}}$. Inspection shows that D_j (as a function of the problem size 4^n) is given by

$$D_j = \mathrm{diag}(\omega_{4^n}^0, \omega_{4^n}^j, \omega_{4^n}^{2j}, \omega_{4^n}^{3j}), \quad 0 \le j < 4^{n-1}. \tag{12}$$

Hence, the function implementing $y = (\mathrm{DFT}_4\, D_j)x$ also needs a stride as parameter, and j to compute the elements of D_j. Also, it can be in-place since it reads from and writes to the same locations of input and output vector. Hence it takes the form:

```
    void DFT4_twiddle(double *Y, int s, int n, int j);
```

The final recursive function is given below. There are some address multiplications by 2, required to implement arrays of complex numbers as arrays (of twice the size) of real numbers.

```
// recursive radix-4 DFT implementation

// compute the exponent
#include <math.h>
#define log4(N) (int)(log(N)/log(4))

// top-level call to DFT function
#define DFT(N, Y, X) DFT_rec(N, log4(N), Y, X, 1)

// DFT kernels
void DFT4_base(double *Y, double *X, int s);
void DFT4_twiddle(double *Y, int s, int N, int j);

// recursive radix-4 DFT function
// N: problem size
// Y: output vector
// X: input vector
// s: stride to access X
void DFT_rec(int N, int n, double  *Y, double *X, int s)
{ int j;

  if (N==4)
    // Y = DFT_4 X
    DFT4_base(Y, X, s);
  else {
    // Y = (I_4 x DFT_N/4)(L^N_4) X
    for(j=0; j<4; j++)
      DFT_rec(N/4, n-1, Y+(2*(N/4)*j), X+2*j*s, s*4);
    // Y = (DFT_4 x I_{N/4})(D_N,4) Y
    for(j=0; j<N/4; j++)
      DFT4_twiddle(Y+2*j, N/4, n, j);
  }
}
```

Buffering. The kernel `DFT4_twiddle` accesses both input and output in a stride. For large sizes $N = 4^n$, this stride is a large two-power, which means that all elements accessed by the kernel are mapped to the same set in the cache (see Fig. 6). If the cache does not have sufficient associativity, cache thrashing occurs. Namely, each iteration of the `DFT4_twiddle` loop has to load 4 cache lines and all these cache lines get evicted before the next iteration of the `DFT4_twiddle` loop can use the already loaded remaining cache lines.

Buffering alleviates these problems to a certain degree. An initial and final copy operation introduce overheads, but all intermediate steps are done on contiguous data, preventing cache thrashing.

As an example, buffering is performed on the second loop of the preceding code, leading to the following code. We assume a cache line size of `LS` complex numbers (= 4 doubles). (If `LS` is larger than the radix size, one needs special cases for some recursion steps.) To implement buffering, we first split the `j` loop into `N/(2*LS)` $\times$ `LS` iterations. We add copying to the body of the *outer* tiled `j1` loop. Our copy operation handles cache lines and thus data for multiple DFTs. In particular, we copy 4 sets of `LS` consecutive complex elements (4 cache lines) into a local buffer. The inner tiled `j2` loop performs `LS` DFTs on the local contiguous buffer. The large, performance degrading complex stride 4^{n-1} in the original `j` loop gets replaced by a small complex stride `LS` in the `j2` loop at the cost of two copy operations that copy whole cache lines. The threshold parameter `th` controls the sizes for which the second loop gets buffered.

```
// cache line size = 2 complex numbers (16 bytes)
# define LS   2

// recursive radix-4 DFT function with buffering
// N: problem size
// Y: output vector
// X: input vector
// s: stride to access X
// th: threshold size to stop buffering
void DFT_buf_rec(int N, int n, double  *Y, double *X, int s, int th)
{ int i, j, j1, j2, k;
  // local buffer
  double buf[8*LS];

  if (N==4)
    // Y = DFT_4 X
    DFT4_base(Y, X, s);
  else
  { // Y = (I_4 x DFT_{N/4})(L^N_4) X
    if (N > th)
      for(j=0; j<4; j++)
        DFT_buf_rec(N/4, n-1, Y+(2*(N/4)*j), X+2*j*s, s*4, th);
    else
      for(j=0; j<4; j++)
        DFT_rec(N/4, n-1, Y+(2*(N/4)*j), X+2*j*s, s*4);

    // Y = (DFT_4 x I_{N/4})(D_{N,4}) Y, buffered for LS
    // j loop tiled by LS
    for(j1=0; j1<N/(4*LS); j1++)
    { // copy 4 chunks of 2*LS double to local buffer
      for(i=0; i<4; i++)
        for(k=0; k<2*LS; k++)
          buf[2*LS*i+k] = Y[(2*LS*j1)+(2*(N/4)*i)+k];

      // perform LS DFT4 on contiguous data
      // buf = (DFT4 Dj x I_LS) buf
      for(j2=0; j2<LS; j2++)
        DFT4_twiddle(buf+2*j2, LS, n, j1*LS+j2);

      // copy 4 chunks of 2*LS double to output
      for(i=0; i<4; i++)
```

```
        for(k=0; k<2*LS; k++)
          Y[(2*LS*j1)+(2*(N/4)*i)+k] = buf[2*LS*i+k];
    }
  }
}
```

One can perform a similar buffering operation on the input X for the call to DFT_rec, as X is accessed at a large stride. This buffering must take place as special case for $N = 16$ in DFT_rec and requires a third variant of the recursive function DFT_rec.

6.3 CPU and Register Level Optimization

This section describes the design and implementation of optimized DFT base cases (kernels). We again restrict the discussion to the recursive radix-4 FFT algorithm. Extensions to mixed-radix implementations requires different kernel sizes, all implemented following the ideas presented in this section. High-performance implementations may use kernels of up to size 64 [23, 78].

Blocking. We apply (3) to the DFT_4:

$$\mathrm{DFT}_4 = (\mathrm{DFT}_2 \otimes I_2) D_{4,2} (I_2 \otimes \mathrm{DFT}_2) L^4_2. \tag{13}$$

As (13) is a recursive formula, an implementation based on (13) is automatically blocked.

Unrolling and scheduling. We implement (13) according to the rules summarized in Table 4. We aim at implementing recursion leafs. Thus the code needs to be unrolled. Due to the recursive nature of (13), kernels derived from (13) are automatically reasonably scheduled.

For DFT kernels, larger unrolled kernels lead to slightly less operations, as more twiddle factors are known at optimization time and one can take better advantage of trivial complex multiplications. However, larger kernels do not increase the available instruction level parallelism as much as in MMM, since the DFT data flow is more complicated and imposes stronger constraints on the operation ordering.

Scalar replacement. We next apply scalar replacement as described in Section 4.3. Every element in the input array X is only referenced twice, and every location of the output array Y is written once. Hence, we only replace the temporary array t by scalar variables, but do not replace accesses to X and Y. Experiments suggest that this strategy is sufficient for obtaining maximum performance. This leads to the following code for DFT4_base. From the discussion in Section 6.2 we know that this function loads at complex stride s from *X and writes at unit stride to *Y. We obtain the following code:

```
// DFT4 implementation
void DFT4_base(double  *Y, double  *X, int s)
{ double t0, t1, t2, t3, t4, t5, t6, t7;
  t0 = (X[0] + X[4*s]);
  t1 = (X[2*s] + X[6*s]);
  t2 = (X[1] + X[4*s+1]);
  t3 = (X[2*s+1] + X[6*s+1]);
  t4 = (X[0] - X[4*s]);
  t5 = (X[2*s+1] - X[6*s+1]);
  t6 = (X[1] - X[4*s+1]);
  t7 = (X[2*s] - X[6*s]);
  Y[0] = (t0 + t1);
  Y[1] = (t2 + t3);
  Y[4] = (t0 - t1);
  Y[5] = (t2 - t3);
  Y[2] = (t4 - t5);
  Y[3] = (t6 + t7);
  Y[6] = (t4 + t5);
  Y[7] = (t6 - t7);
}
```

Precomputation of constants. The kernel `DFT4_twiddle` computes $y = (\mathrm{DFT}_4\, D_j)x$, which contains multiplication with the complex diagonal D_j as defined in (12). The entries of D_j are complex roots of unity (twiddle factors) that depend on the recursion level and the loop counter `j`. Computing the actual entries of D_j requires evaluations of $\sin \frac{k\pi}{N}$ and $\cos \frac{k\pi}{N}$ for suitable values of k and N, which requires expensive calls to the math library. Hence these numbers should be precomputed.

We introduce an initialization function `init_DFT` that precomputes all diagonals required for size N and stores pointers to the tables (one table for each recursion level) in the global variable `double **DN`, as shown below.

```
#define PI    3.14159265358979323846
// twiddle table, initialized by init_DFT(N)
double **DN;

void init_DFT(int N)
{ int i, j, k, size_Dj = 16, n_max = log4(N);
  DN = malloc(sizeof(double*)*(n_max-1));

  for (j=1; j<n_max; j++, size_Dj*=4)
  { double *Dj = DN[j-1] = malloc(2*sizeof(double)*size_Dj);
    for (k=0; k<size_Dj/4; k++)
      for (i=0; i<4; i++)
      { *(Dj++) = cos(2*PI*i*k/size_Dj);
        *(Dj++) = sin(2*PI*i*k/size_Dj);
      }
  }
}
```

The function `DFT4_twiddle` is shown below.

```
// C macro for complex multiplication
#define CMPLX_MULT(cr, ci, a, b, idx, s) \
{ double ar, ai, br, bi;                    \
  ar = a[2*s*idx]; ai = a[2*s*idx+1];       \
  br = b[2*idx]; bi = b[2*idx+1];           \
  cr = ar*br - ai*bi;                       \
  ci = ar*bi + ai*br;                       \
}

// DFT4*D_j implementation
void DFT4_twiddle(double *Y, int s, int n, int j)
{ double t0, t1, t2, t3, t4, t5, t6, t7,
    X0, X1, X2, X3, X4, X5, X6, X7;
  double *Dj;

  // complex multiplications from D_N
  Dj = DN[n-2]+8*j;
  CMPLX_MULT(X0, X1, Y, Dj, 0, s);
  CMPLX_MULT(X2, X3, Y, Dj, 1, s);
  CMPLX_MULT(X4, X5, Y, Dj, 2, s);
  CMPLX_MULT(X6, X7, Y, Dj, 3, s);

  // operations from DFT4
  t0 = (X0 + X4);
  t1 = (X2 + X6);
  t2 = (X1 + X5);
  t3 = (X3 + X7);
  t4 = (X0 - X4);
  t5 = (X3 - X7);
  t6 = (X1 - X5);
  t7 = (X2 - X6);
  Y[0] = (t0 + t1);
  Y[1] = (t2 + t3);
  Y[4*s] = (t0 - t1);
  Y[4*s+1] = (t2 - t3);
  Y[2*s] = (t4 - t5);
  Y[2*s+1] = (t6 + t7);
  Y[6*s] = (t4 + t5);
  Y[6*s+1] = (t6 - t7);
}
```

6.4 Performance Evaluation

We now evaluate the performance of the recursive radix-4 FFT derived in this section and compare it to the Numerical Recipes and the sequential, scalar (single core, x87) version of FFTW 3.1.2. All implementations are run on a single core of a 2.66 GHz Intel Core2 Duo, with a theoretical scalar peak performance of 5.32 Gflop/s. We compile all implementation with the Intel C++ compiler 10.0 with options “/O3 /QxT” to obtain maximum optimization. The radix-4 implementation was copied directly from the code listings above. The Numerical Recipes FFT implementation is in single-precision and inplace while both our radix-4 FFT and FFTW are double-precision and out-of-place. This gives a slight performance advantage to the Numerical Recipes FFT implementation.

Fig. 14 shows the performance results for the three FFT implementations. We see that Numerical Recipes reaches about 1 Gflop/s and drops sharply to 160 Mflop/s when the memory footprint for the problems is too large for the L2 cache. The radix-4 FFT implementation we derived in this tutorial reaches about 2 Gflop/s for problem sizes that fit into the L2 cache. For larger sizes the performance drops down to about 1 Gflop/s. FFTW 3.1.2 in sequential scalar mode shows the upper bound for practically achievable performance when using x87 instructions and a single core. FFTW reaches about 2.5–3 Gflop/s for cache-resident sizes and 1.6 Gflop/s for out-of-cache sizes.

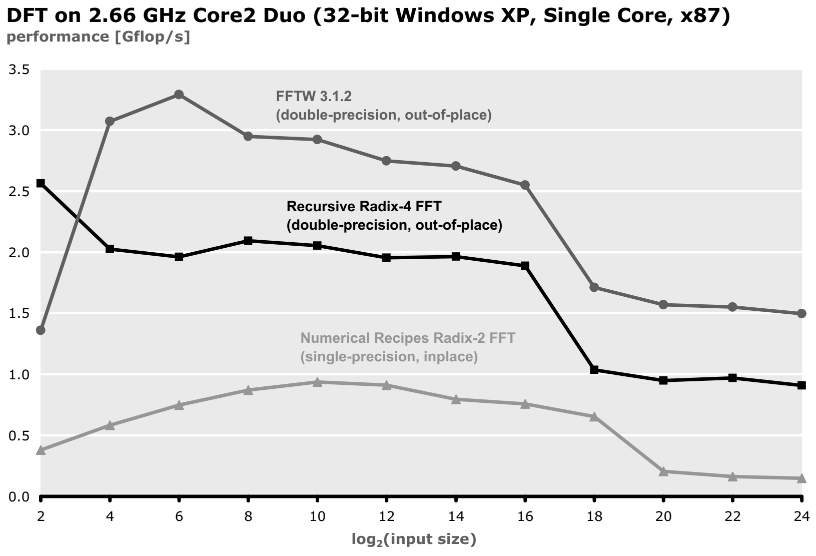


Fig. 14. Performance results for three FFT implementations on a 2.66 GHz Intel Core2 Duo. All implementations are sequential and scalar (single core, x87). Higher is better.

Analysis of the above data can be summarized as follows.

– The recursive radix-4 FFT is twice as fast as Numerical Recipes for in-cache sizes and about 6 times faster for out-of-cache sizes.

– The radix-4 FFT implementation reaches more than two thirds of the performance of scalar FFTW. The performance difference is mainly due to FFTW's larger basic block sizes (codelets), its ability to choose different radices at different recursion steps, and a few additional loop optimizations.

– There is still a lot of room for further improvement using Intel's SSE instructions and both cores (see Fig. 2).

In addition, our experiments show that buffering does not produce any performance gain in this case, since the cache associativity on the Core2 architecture is 8, which is large enough for a radix-4 kernel.

6.5 Parameter-Based Performance Tuning

We now discuss the parameters in our DFT implementation that can be tuned to the memory hierarchy.

Base case sizes. The most important parameter tuning is the selection of base cases. To allow for multiple base cases `DFT_base` and `DFT_twiddle`, the program structure must become more complex, as a data structure describing the recursion and containing function pointers to the appropriate kernels replaces the two parameters `N` and `n` in `DFT_rec`. The resulting program would be very similar to FFTW 2.x.

After this infrastructural change the system can apply any function `DFT_twiddle` in the second stage of the recursion and any function `DFT_base` as recursion leaf. The tuning process needs to find for each recursion step the right kernel size. FFTW uses both a cost estimation and runtime experiments based on dynamic programming to find good parameter choices [10]. Showing the full implementation is beyond the scope of this tutorial.

Threshold for buffering. The second parameter decides the sizes for which buffering should be applied. This depends on the cache size of the target machine, as buffering only becomes beneficial for problem sizes that are not resident in the L2 cache.

Buffer size. Finally, we need to set the buffer size based on the cache line size of the target machine to prevent cache thrashing. The cache line size can be either looked up or found experimentally.

6.6 Program Generation for DFT: Spiral

Spiral [7] is a program generator for linear transforms. It can generate optimized fixed-size and variable-size code for the DFT, the Walsh-Hadamard transform (WHT), the discrete cosine and sine transforms, finite impulse response (FIR) filters, the discrete wavelet transform, and others. Spiral builds on the Kronecker product framework for the DFT, described in Section 6.1, but extends it to the whole domain of linear transforms. Further, Spiral automates the optimization process outlined in Sections 6.2–6.5 as well as many other optimizations including various forms of parallelization [54, 26, 79, 80]. The fastest FFT implementation shown in Fig. 2 is generated using Spiral.

In contrast to ATLAS, Spiral is not based on searching a parameterized space, but on a domain-specific language (DSL) that enables the enumeration and systematic optimization of algorithms. More specifically, there are two key ideas underlying Spiral:

1. *Mathematical, structural, declarative DSL.* Spiral uses a DSL to describe algorithms. The DSL is called SPL [81] and is directly derived from the transform domain: it is precisely (an extension of) the Kronecker formalism described in Section 6.1. The language describes only the structure of algorithms and is hence declarative. This property enables structural algorithm optimizations including parallelization that is not practical to perform on C code.
2. *Optimization through rewriting.* Spiral uses rewriting systems [82] for both the generation of alternative algorithms and the structural optimization of algorithms at a high level of abstraction. The rewriting rules for the former are divide-and-conquer algorithms specified as in (3) and for the latter, they are known matrix identities.

Architecture. The input to Spiral is a formally specified transform (for instance, DFT_{384}); the output is a highly optimized C program implementing the transform. These highly optimized programs may use language extensions or software libraries to access special machine features like multiple cores or SIMD vector instructions. We show the architecture of Spiral in Fig. 15 and discuss it below.

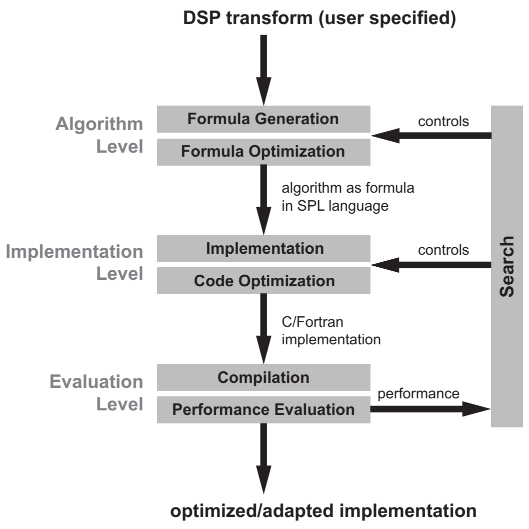


Fig. 15. The architecture of Spiral (from [7]).

– *Algorithm level.* This stage is responsible for generating and optimizing algorithms for the specified transforms.
  - *Formula generation.* A transform like DFT_{384} is considered to be a non-terminal. Spiral uses *breakdown rules* to describe recursive algorithms for linear transforms. For example, (3) and (7) are breakdown rules expressing larger DFTs in terms of smaller DFTs. Base cases terminate the recursion. For instance, (5) is the DFT base rule.

A rewriting system recursively applies breakdown rules to the specified transform to produce alternative algorithms represented as SPL expressions, also called formulas.

- *Formula optimization.* Formulas are structurally optimized, also using a rewriting system. Loop fusion is performed using rewriting rules which essentially perform the same reasoning and restructuring as described in Section 6.2. The loop fusion by rewriting requires the extension of SPL to a more powerful language called Σ-SPL [24]. Further, rewriting is used for various forms of parallelization including the efficient mapping to multiple processor cores or SIMD vector instructions. The next section will provide more details on this topic.

– *Implementation level.* Spiral contains a special-purpose compiler that translates formulas into code. The compiler is based on (an extension of) Table 4. Moreover, it performs all kernel-level optimizations described in Section 6.3. Depending upon an unrolling threshold, subformulas smaller than the threshold are treated as kernels, while larger formulas are implemented using loops.

– *Evaluation level.* This stage is responsible for compiling and measuring the runtime of the generated code.

– *Search.* The measured runtime guides Spiral in picking a new candidate formula by changing the breakdown of the non-terminal. The feedback loop is guided by a search strategy, usually a form of dynamic programming. The main purpose of the search is adaptation to the platform's memory hierarchy.

Structural optimization through rewriting. A core component of Spiral's optimization process is the structural optimization of formulas using a rewriting system. As briefly discussed above, two major optimization goals are achieved through rewriting: 1) loop merging [24], and 2) the mapping of algorithms to parallel architectures like multicore CPUs or SIMD vector extensions [54, 26]. Loop merging is beyond the scope of this tutorial as it requires the introduction of a new language, Σ-SPL. Thus, we only briefly discuss the mapping to parallel architectures.

Analysis of the access pattern of tensor products shows that certain tensor products can be mapped very well to some architectures but only poorly to others. As example, in (3) the construct

$$I_m \otimes \mathrm{DFT}_n \tag{14}$$

has a perfect structure for m-way parallel machines with either shared or distributed memory. However, implementing it with SIMD vector instructions introduces considerable overhead [54]. Similarly, the construct

$$\mathrm{DFT}_m \otimes I_n \tag{15}$$

has a perfect structure for n-way vector SIMD architectures. However, implementing it on shared memory machines leads to false sharing, while on distributed memory machines tiny messages would be required, which degrades performance.

Using algebraic identities [53] one can change the structure of formulas. For instance, the identity

$$\mathrm{DFT}_m \otimes I_n = L_m^{mn}(I_n \otimes \mathrm{DFT}_m)L_n^{mn} \tag{16}$$

replaces a vector formula by a parallel formula and introduces two stride permutations.

Spiral uses a rewriting system to perform formula manipulations like (16), using a tagging mechanism to steer the manipulation toward the final formula optimized for a certain architecture. Spiral's rewriting system consists of three main components.

– *Tags* encode target architecture types and parameters. They contain high-level information about the target architecture. For instance, Spiral uses the tags "vec(ν)" for SIMD vector extensions (ν encodes the vector length of the architecture) and "smp(p, μ)" for shared memory (p is the number of processors and μ the length of cache lines).

– *Base cases* describe formula constructs that are guaranteed to be mapped efficiently to the target hardware. Spiral uses special operator variants to encode base cases. For instance, a p-way parallel base case is denoted by the tagged operator "$\otimes_\|$"; A_n is any $n \times n$ matrix expression.

– *Rewriting rules* encode formula manipulation identities, but in addition "know" the target machine and thus deduce the "right" parameters for identities with degrees of freedom. For instance, the identity (16) is translated into the rewriting rule

$$\underbrace{A_m \otimes I_n}_{\text{smp}(p,\mu)} \to \underbrace{L_m^{mn}}_{\text{smp}(p,\mu)} \big(I_p \otimes_\| (I_{n/p} \otimes A_m)\big) \underbrace{L_n^{mn}}_{\text{smp}(p,\mu)} .$$

This rule has the additional knowledge of the target system's processor count, and utilizes this knowledge when applying the helper identity

$$I_{mn} = I_m \otimes I_n.$$

The stride permutations L_m^{mn} and L_n^{mn} will be handled by further rewriting.

For every type of parallelism, these three components are added to Spiral to enable the corresponding structural optimization. In addition, every class of target machines may require a small extension of the SPL compiler to translate tagged operators into target code. For instance, "$\otimes_\|$" will be translated into OpenMP parallel for loops, when Spiral generates shared memory code using OpenMP.

Discussion. Spiral fully automates the process of optimizing linear transforms for a large class of state-of-the-art architectures. The code it generates is competitive with expertly hand-tuned implementations and often outperforms these. The key is Spiral's domain-specific, declarative, mathematical language to describe algorithms. Spiral's algorithm (breakdown rule) database contains the algorithmic knowledge of more than a hundred journal papers on transform algorithms. Spiral's rewriting system is the key to structural optimization and parallelization of algorithms. With this approach it is possible to re-target Spiral to new parallel platforms. So far Spiral successfully generated (at least prototypical) fast implementations for SIMD vector extensions, multicore CPUs, cluster computers, graphics processors (GPUs), and the Cell BE processor. In addition, Spiral generates hardware designs for field-programmable gate arrays (FPGAs), and hardware-software partitioned implementations.

While Spiral focuses on transforms, the basic principles underlying it may be applicable to other numerical problem domains.

6.7 Exercises

1. **WHT: Operations count.** The Walsh-Hadamard transform (WHT) is related to the DFT but has a simpler structure and simpler algorithms. The WHT is defined only for 2-power input sizes $N = 2^n$, as given by the matrix

$$\mathrm{WHT}_{2^n} = \underbrace{\mathrm{DFT}_2 \otimes \mathrm{DFT}_2 \otimes \ldots \otimes \mathrm{DFT}_2}_{n \text{ factors}},$$

where DFT_2 is as defined in (5).

   (a) How many entries of the WHT are zeros and why? Determine the number of additions and the number of multiplications required when computing the WHT by definition.

   (b) The WHT of an input vector can be computed iteratively or recursively using the following formulas:

$$\mathrm{WHT}_{2^n} = \prod_{i=0}^{n-1} (I_{2^{n-i-1}} \otimes \mathrm{DFT}_2 \otimes I_{2^i}) \quad \text{(iterative)} \qquad (17)$$

$$\mathrm{WHT}_{2^n} = (\mathrm{DFT}_2 \otimes I_{2^{n-1}})(I_2 \otimes \mathrm{WHT}_{2^{n-1}}) \quad \text{(recursive)} \qquad (18)$$

   (c) Determine the exact operations counts (again, additions and multiplications separately) of both algorithms. Also determine the degree of reuse as defined in Section 2.1.

2. **WHT: Implementation.**

   (a) Implement a recursive implementation of the WHT based on (18).

   (b) Implement the triple loop (iterative) version of the WHT using (17). Create a performance plot (size versus Mflop/s) for sizes 2^1–2^{20} comparing the iterative and the recursive versions. Discuss the plot.

   (c) Create unrolled WHTs of sizes 4 and 8 based on the recursive WHT algorithm. (The number of operations should match the cost computed in Exercise 1c on page 63).

   (d) Now implement recursive radix-4 and radix-8 implementations of the WHT based on the formulas

$$\mathrm{WHT}_{2^n} = (\mathrm{WHT}_4 \otimes I_{2^{n-2}})(I_4 \otimes \mathrm{WHT}_{2^{n-2}}) \quad \text{(radix-4)}$$

$$\mathrm{WHT}_{2^n} = (\mathrm{WHT}_8 \otimes I_{2^{n-3}})(I_8 \otimes \mathrm{WHT}_{2^{n-3}}) \quad \text{(radix-8)}$$

In these implementations, the left hand side WHT (of size 4 or 8) should be your unrolled kernel (which then has to handle input data at a stride) called in a loop; the right hand side is a recursive call (also called in a loop). Further, in both implementations, you may need one step with a different radix to handle all input sizes.

Measure the performance of both implementations, again for sizes 2^1–2^{20} and add it to the previous plot (four lines total).

(e) Try to further improve the code or perform other interesting experiments. For example, what happens if one considers more general algorithms based on

$$\mathrm{WHT}_{2^n} = (\mathrm{WHT}_{2^i} \otimes I_{2^{n-i}})(I_{2^i} \otimes \mathrm{WHT}_{2^{n-i}})$$

The unrolled code could be the WHT on the left hand side of the above equation. Alternatively, one could run a search to find the best radix in each step independently.

7 Conclusions

Writing fast libraries for numerical problems is difficult and requires a thorough understanding of the interaction between algorithms, software, and microarchitecture. Looking ahead, the situation is likely to get worse due to the recent shift to parallelism in mainstream computing, triggered by the end of frequency scaling. We hope this guide conveys the problem, its origin, and a set of basic methods to write fast numerical code.

However, problems also open research opportunities. In this case the problem is the need to automate high performance library development, a difficult challenge that, in its nature, is at the core of computer science. To date this problem has been attacked mostly by the scientific computing and compiler community, and the list of successes is still short. We believe that other areas of computer science need to get involved, including programming languages, and in particular domain-specific languages, generative programming, symbolic computation, and optimization and machine learning. For researchers in these areas, we hope that this tutorial can serve as an entry point to the problem and the existing work on automatic performance tuning.

Acknowledgement

This work was supported by DARPA through the DOI grant NBCH1050009 and the ARO grant W911NF0710416, by NSF through awards 0325687 and 0702386, and by an Intel grant.

References

1. Moore, G.E.: Cramming more components onto integrated circuits. Readings in computer architecture (2000) 56–59
2. Meadows, L., Nakamoto, S., Schuster, V.: A vectorizing, software pipelining compiler for LIW and superscalar architecture. In: Proceedings of Risc. (1992)
3. Group, S.S.C.: SUIF: A parallelizing & optimizing research compiler. Technical Report CSL-TR-94-620, Computer Systems Laboratory, Stanford University (May 1994)
4. Franke, B., O'Boyle, M.F.P.: A complete compiler approach to auto-parallelizing C programs for multi-DSP systems. IEEE Trans. Parallel Distrib. Syst. **16**(3) (2005) 234–245
5. Van Loan, C.: Computational Framework of the Fast Fourier Transform. SIAM (1992)
6. Press, W.H., Flannery, B.P., A., T.S., T., V.W.: Numerical Recipes in C: The Art of Scientific Computing. 2nd edn. Cambridge University Press (1992)
7. Püschel, M., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Singer, B.W., Xiong, J., Franchetti, F., Gačić, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: Code generation for DSP transforms. Proc. of the IEEE **93**(2) (2005) 232–275 Special issue on Program Generation, Optimization, and Adaptation.
8. Website: Spiral (1998) `http://www.spiral.net`.
9. Frigo, M., Johnson, S.G.: FFTW: An adaptive software architecture for the FFT. In: Proc. IEEE Int'l Conf. Acoustics, Speech, and Signal Processing (ICASSP). Volume 3. (1998) 1381–1384
10. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proceedings of the IEEE **93**(2) (2005) 216–231 Special issue on Program Generation, Optimization, and Adaptation.
11. Website: FFTW `http://www.fftw.org`.
12. Goto, K., van de Geijn, R.: On reducing TLB misses in matrix multiplication, FLAME working note 9. Technical Report TR-2002-55, The University of Texas at Austin, Department of Computer Sciences (Nov. 2002)
13. Whaley, R.C., Dongarra, J.: Automatically Tuned Linear Algebra Software (ATLAS). In: Proc. Supercomputing. (1998)
14. Moura, J.M.F., Püschel, M., Padua, D., Dongarra, J.: Scanning the issue: Special issue on program generation, optimization, and platform adaptation. Proceedings of the IEEE, special issue on Program Generation, Optimization, and Adaptation **93**(2) (2005) 211–215
15. Bida, E., Toledo, S.: An automatically-tuned sorting library. Software: Practice and Experience **37**(11) (2007) 1161–1192
16. Li, X., Garzar, M.J., Padua, D.: A dynamically tuned sorting library. In: Proc. International Symposium on Code Generation and Optimization (CGO). (2004) 111–124
17. Im, E.J., Yelick, K., Vuduc, R.: Sparsity: Optimization framework for sparse matrix kernels. Int'l J. High Performance Computing Applications **18**(1) (2004) 135–158
18. Demmel, J., Dongarra, J., Eijkhout, V., Fuentes, E., Petitet, A., Vuduc, R., Whaley, C., Yelick, K.: Self adapting linear algebra algorithms and software. Proceedings of the IEEE **93**(2) (2005) 293–312 Special issue on Program Generation, Optimization, and Adaptation.
19. Website: BeBOP `http://bebop.cs.berkeley.edu/`.
20. Vuduc, R., Demmel, J.W., Yelick, K.A.: OSKI: A library of automatically tuned sparse matrix kernels. In: Proceedings of SciDAC 2005. Journal of Physics: Conference Series, San Francisco, CA, USA, Institute of Physics Publishing (June 2005)
21. Whaley, R., Petitet, A., Dongarra, J.: Automated empirical optimization of software and the ATLAS project. Parallel Computing **27**(1-2) (2001) 3–35
22. Bilmes, J., Asanović, K., whye Chin, C., Demmel, J.: Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In: Proceedings of International Conference on Supercomputing (ICS). (1997)

23. Frigo, M.: A fast Fourier transform compiler. In: Proc. Programming Language Design and Implementation (PLDI). (1999) 169–180
24. Franchetti, F., Voronenko, Y., Püschel, M.: Formal loop merging for signal transforms. In: Proc. Programming Language Design and Implementation (PLDI). (2005) 315–326
25. Franchetti, F., Voronenko, Y., Püschel, M.: FFT program generation for shared memory: SMP and multicore. In: Proc. Supercomputing. (2006)
26. Franchetti, F., Voronenko, Y., Püschel, M.: A rewriting system for the vectorization of signal transforms. In: Proc. High Performance Computing for Computational Science (VECPAR). (2006)
27. Bientinesi, P., Gunnels, J.A., Myers, M.E., Quintana-Orti, E., van de Geijn, R.: The science of deriving dense linear algebra algorithms. ACM TOMS **31**(1) (2005) 1–26
28. Gunnels, J.A., Gustavson, F.G., Henry, G.M., van de Geijn, R.A.: FLAME: Formal linear algebra methods environment. ACM TOMS **27**(4) (2001) 422–455
29. Quintana-Orti, G., Quintana-Orti, E.S., van de Geijn, R., Van Zee, F.G., Chan, E.: Programming algorithms-by-blocks for matrix computations on multithreaded architectures, submitted for publication
30. Baumgartner, G., Auer, A., Bernholdt, D.E., Bibireata, A., Choppella, V., Cociorva, D., Gao, X., Harrison, R.J., Hirata, S., Krishanmoorthy, S., Krishnan, S., Lam, C.C., Lu, Q., Nooijen, M., Pitzer, R.M., Ramanujam, J., Sadayappan, P., Sibiryakov, A.: Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. Proceedings of the IEEE **93**(2) (2005) 276–292 Special issue on Program Generation, Optimization, and Adaptation.
31. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
32. Lämmel, R., Saraiva, J., Visser, J., eds.: Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers. In Lämmel, R., Saraiva, J., Visser, J., eds.: GTTSE. Volume 4143 of Lecture Notes in Computer Science., Springer (2006)
33. Püschel, M.: How to write fast code. `http://www.ece.cmu.edu/~pueschel/teaching/18-645-CMU-spring08/course.html` (2008) Course 18-645, Electrical and Computer Engineering, Carnegie Mellon University.
34. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT Press, Cambridge, MA, USA (2001)
35. Demmel, J.W.: Applied numerical linear algebra. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (1997)
36. Tolimieri, R., An, M., Lu, C.: Algorithms for discrete Fourier transforms and convolution. 2nd edn. Springer (1997)
37. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann (May 2002)
38. Bryant, R.E., O'Hallaron, D.R.: Computer Systems: A Programmer's Perspective. Prentice Hall (2003)
39. Strassen, V.: Gaussian elimination is not optimal. Numerische Mathematik **14**(3) (1969) 354–356
40. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. Journal of Symbolic Computation **9** (1990) 251–280
41. Blackford, L.S., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., Heroux, M., Kaufman, L., Lumsdaine, A., Petitet, A., Pozo, R., Remington, K., Whaley, R.C.: An updated set of Basic Linear Algebra Subprograms (BLAS). ACM Transactions on Mathematical Software **28**(2) (2002) 135–151

42. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide. 3rd edn. Society for Industrial and Applied Mathematics, Philadelphia, PA (1999)
43. Website: ATLAS `http://math-atlas.sourceforge.net/`.
44. Website: Goto BLAS `http://www.tacc.utexas.edu/general/staff/goto/`.
45. Website: LAPACK `http://www.netlib.org/lapack/`.
46. Website: ScaLAPACK `http://www.netlib.org/scalapack/`.
47. Blackford, L.S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK Users' Guide. Society for Industrial and Applied Mathematics, Philadelphia, PA (1997)
48. Website: PLAPACK `http://www.cs.utexas.edu/users/plapack/`.
49. Chtchelkanova, A., Gunnels, J., Morrow, G., Overfelt, J., van de Geijn, R.: Parallel implementation of BLAS: General techniques for level 3 BLAS. Concurrency: Practice and Experience **9**(9) (1997) 837–857
50. Website: FLAME `http://www.cs.utexas.edu/users/flame/`.
51. Johnson, S.G., Frigo, M.: A modified split-radix FFT with fewer arithmetic operations. IEEE Trans. Signal Processing **55**(1) (2007) 111–119
52. Nussbaumer, H.J.: Fast Fourier Transformation and Convolution Algorithms. 2nd edn. Springer (1982)
53. Johnson, J.R., Johnson, R.W., Rodriguez, D., Tolimieri, R.: A methodology for designing, modifying, and implementing FFT algorithms on various architectures. Circuits Systems Signal Processing **9**(4) (1990) 449–500
54. Franchetti, F., Püschel, M.: Short vector code generation for the discrete Fourier transform. In: Proc. IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS). (2003) 58–67
55. Bonelli, A., Franchetti, F., Lorenz, J., Püschel, M., Ueberhuber, C.W.: Automatic performance optimization of the discrete Fourier transform on distributed memory computers. In: Proc. International Symposium on Parallel and Distributed Processing and Applications (ISPA). (2006)
56. Website: FFTPACK `http://www.netlib.org/fftpack/`.
57. GNU: GSL `http://www.gnu.org/software/gsl/`.
58. Mirković, D., Johnsson, S.L.: Automatic performance tuning in the UHFFT library. In: Proc. Int'l Conf. Computational Science (ICCS). Volume 2073 of LNCS., Springer (2001) 71–80
59. Website: UHFFT `http://www2.cs.uh.edu/~mirkovic/fft/parfft.htm`.
60. Website: FFTE `http://www.ffte.jp`.
61. Website: ACML `http://developer.amd.com/acml.jsp`.
62. Website: Intel MKL `http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm`.
63. Website: Intel IPP `http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/302910.htm`.
64. Website: IBM ESSL and PESSL `http://www-03.ibm.com/systems/p/software/essl.html`.
65. Website: NAG `http://www.nag.com/`.
66. Website: IMSL `http://www.vni.com/products/imsl/`.
67. Hill, M.D., Smith, A.J.: Evaluating associativity in CPU caches. IEEE Trans. Comput. **38**(12) (1989) 1612–1630
68. Intel Corporation: Intel 64 and IA-32 Architectures Optimization Reference Manual. (2007) `http://www.intel.com/products/processor/manuals/index.htm`.
69. Advanced Micro Devices (AMD) Inc.: Software Optimization Guide for AMD Athlon 64 and AMD Optero Processors. (2005) `http://developer.amd.com/devguides.jsp`.

70. GNU: Gcc:optimization options http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.
71. Intel: Quick-reference guide to optimization with intel compilers version 10.x `http://cache-www.intel.com/cd/00/00/22/23/222300\_222300.pdf`.
72. Intel: Intel VTune
73. Microsoft: Microsoft Visual Studio
74. GNU: Gnu gprof manual `http://www.gnu.org/software/binutils/manual/gprof-2.9.1/html\_mono/gprof.html`.
75. Yotov, K., Li, X., Ren, G., Garzaran, M., Padua, D., Pingali, K., Stodghill, P.: A comparison of empirical and model-driven optimization. Proceedings of the IEEE **93**(2) (2005) Special issue on Program Generation, Optimization, and Adaptation.
76. Wolfe, M.: Iteration space tiling for memory hierarchies. In: SIAM Conference on Parallel Processing for Scientific Computing. (1987)
77. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Math. of Computation **19** (1965) 297–301
78. Püschel, M., Singer, B., Xiong, J., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Johnson, R.W.: SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. Int'l Journal of High Performance Computing Applications **18**(1) (2004) 21–45
79. D'Alberto, P., Milder, P.A., Sandryhaila, A., Franchetti, F., Hoe, J.C., Moura, J.M.F., Püschel, M., Johnson, J.: Generating FPGA accelerated DFT libraries. In: IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). (2007)
80. Milder, P.A., Franchetti, F., Hoe, J.C., Püschel, M.: Formal datapath representation and manipulation for implementing DSP transforms. In: Design Automation Conference (DAC). (2008)
81. Xiong, J., Johnson, J., Johnson, R., Padua, D.: SPL: A language and compiler for DSP algorithms. In: Proc. Programming Language Design and Implementation (PLDI). (2001) 298–308
82. Dershowitz, N., Plaisted, D.A.: Rewriting. In: Handbook of Automated Reasoning. Volume 1. Elsevier (2001) 535–610