A Framework for Practical Parallel Fast Matrix Multiplication

Austin R. Benson
Institute for Computational and Mathematical Engineering
Stanford University
Stanford, CA USA
arbenson@stanford.edu

Grey Ballard
Sandia National Laboratories
Livermore, CA USA
gmballa@sandia.gov

Abstract

Matrix multiplication is a fundamental computation in many scientific disciplines. In this paper, we show that novel fast matrix multiplication algorithms can significantly outperform vendor implementations of the classical algorithm and Strassen's fast algorithm on modest problem sizes and shapes. Furthermore, we show that the best choice of fast algorithm depends not only on the size of the matrices but also the shape. We develop a code generation tool to automatically implement multiple sequential and shared-memory parallel variants of each fast algorithm, including our novel parallelization scheme. This allows us to rapidly benchmark over 20 fast algorithms on several problem sizes. Furthermore, we discuss a number of practical implementation issues for these algorithms on shared-memory machines that can direct further research on making fast algorithms practical.

Categories and Subject Descriptors G.4 [*Mathematical software*]: Efficiency; G.4 [*Mathematical software*]: Parallel and vector implementations

Keywords fast matrix multiplication, dense linear algebra, parallel linear algebra, shared memory

1. Introduction

Matrix multiplication is one of the most fundamental computations in numerical linear algebra and scientific computing. Consequently, the computation has been extensively studied in parallel computing environments (*cf.* [3, 20, 34] and references therein). In this paper, we show that fast algorithms for matrix-matrix multiplication can achieve higher performance on sequential and shared-memory parallel architectures for modestly sized problems. By fast algorithms, we mean ones that perform asymptotically fewer floating point operations and communicate asymptotically less data than the classical algorithm. We also provide a code generation framework to rapidly implement sequential and parallel versions of over 20 fast algorithms. Our performance results in Section 5 show that several fast algorithms can outperform the Intel Math Kernel Library (MKL) `dgemm` (double precision general matrix-matrix multiplication) routine and Strassen's algorithm [32]. In parallel implementations, fast algorithms can achieve a speedup of 5% over Strassen's original fast algorithm and greater than 15% over MKL.

However, fast algorithms for matrix multiplication have largely been ignored in practice. For example, numerical libraries such as Intel's MKL [19], AMD's Core Math Library (ACML) [1], and the Cray Scientific Libraries package (LibSci) [8] do not provide implementations of fast algorithms, though we note that IBM's Engineering and Scientific Subroutine Library (ESSL) [18] does include Strassen's algorithm. Why is this the case? First, users of numerical libraries typically consider fast algorithms to be of only theoretical interest and never practical for reasonable problem sizes. We argue that this is *not* the case with our performance results in Section 5. Second, fast algorithms do not provide the same numerical stability guarantees as the classical algorithm. In practice, there is some loss in precision in the fast algorithms, but they are not nearly as bad as the worst-case guarantees [14, 27]. Third, the LINPACK benchmark[1] used to rank supercomputers by performance forbids fast algorithms. We suspect that this has driven effort away from the study of fast algorithms.

Strassen's algorithm is the most well known fast algorithm, but this paper explores a much larger class of recursive fast algorithms based on different base case dimensions. We review these algorithms and methods for constructing them in Section 2. The structure of these algorithms makes them amenable to code generation, and we describe this process and other performance tuning considerations in Section 3. In Section 4, we describe three different methods for parallelizing fast matrix multiplication algorithms on shared-memory machines. Our code generator implements all three parallel methods for each fast algorithm. We evaluate the sequential and parallel performance characteristics of the various algorithms and implementations in Section 5 and compare them with MKL's implementation of the classical algorithm as well as an existing implementation of Strassen's algorithm.

The goal of this paper is to help bridge the gap between theory and practice of fast matrix multiplication algorithms. By introducing our tool of automatically translating a fast matrix multiplication algorithm to high performance sequential and parallel implementations, we enable the rapid prototyping and testing of theoretical developments in the search for faster algorithms. We focus the attention of theoretical researchers on what algorithmic characteristics matter most in practice, and we demonstrate to practical researchers the utility of several existing fast algorithms besides Strassen's, motivating further effort towards high performance implementations of those that are most promising. Our contributions are summarized as follows:

PPoPP'15, February 7–11, 2015, San Francisco, CA, USA
Copyright 2015 ACM 978-1-4503-3205-7/15/02...$15.00
http://dx.doi.org/10.1145/2688500.2688513

[1] `http://www.top500.org`

- By using new fast matrix multiplication algorithms, we achieve better performance than Intel MKL's `dgemm`, both sequentially and with 6 and 24 cores on a shared-memory machine.
- We demonstrate that, in order to achieve the best performance for matrix multiplication, the choice of fast algorithm depends on the size and shape of the matrices. Our new fast algorithms outperform Strassen's on the multiplication of rectangular matrices.
- We show how to use code generation techniques to rapidly implement sequential and shared-memory parallel fast matrix multiplication algorithms.
- We provide a new hybrid parallel algorithm for shared-memory fast matrix multiplication.
- We implement a fast matrix multiplication algorithm with asymptotic complexity $O(N^{2.775})$ for square $N \times N$ matrices (discovered by Smirnov [31]). In terms of asymptotic complexity, this is the fastest matrix multiplication algorithm implementation to date. However, our performance results show that this algorithm is not practical for the problem sizes that we consider.

Overall, we find that Strassen's algorithm is hard to beat for square matrix multiplication, both in serial and in parallel. However, for rectangular matrices (which occur more frequently in practice), other fast algorithms can perform much better. The structure of the fast algorithms that perform well tend to "match the shape" of the matrices, an idea that we will make clear in Section 5. We also find that bandwidth is a limiting factor towards scalability in shared-memory parallel implementations of fast algorithms. Our parallel implementations of fast algorithms suffer when memory bandwidth does not scale linearly with the number of cores. Finally, we find that algorithms that are theoretically fast in terms of asymptotic complexity do not perform well on problems of modest size that we consider on shared-memory parallel architectures. We discuss these conclusions in more detail in Section 6.

Finally, all of the software used for this paper is available at `https://github.com/arbenson/fast-matmul`.

1.1 Related Work

There are several sequential implementations of Strassen's fast matrix multiplication algorithm [2, 11, 17], and parallel versions have been implemented for both shared-memory [9, 25] and distributed-memory architectures [3, 13]. For our parallel algorithms in Section 4, we use the ideas of breadth-first and depth-first traversals of the recursion trees, which were first considered by Kumar *et al*. [25] and Ballard *et al*. [3] for minimizing memory footprint and communication.

Apart from Strassen's algorithm, a number of fast matrix multiplication algorithms have been developed, but only a small handful have been implemented. Furthermore, these implementations have only been sequential. Hopcroft and Kerr showed how to construct recursive fast algorithms where the base case is multiplying a $p \times 2$ by a $2 \times n$ matrix [16]. Bini *et al*. introduced the concept of arbitrary precision approximate (APA) algorithms for matrix multiplication and demonstrated a method for multiplying 3×2 by 2×2 matrices which leads to a general square matrix multiplication APA algorithm that is asymptotically faster than Strassen's [5]. Schönhage also developed an APA algorithm that is asymptotically faster than Strassen's, based on multiplying square 3×3 matrices [30]. These APA algorithms suffer from severe numerical issues—both lose at least half the digits of accuracy with each recursive step. While no exact solution can have the same complexity as Bini's algorithm [16], it is still an open question if there exists an exact fast algorithm with the same complexity as Schönhage's. Pan used factorization of trilinear forms and a base case of 70×70 square matrix multiplication to construct an exact algorithm asymptotically faster than Strassen's algorithm [29]. This algorithm was implemented by Kaporin [22], and the running time was competitive with Strassen's algorithm on a sequential machine. Recently, Smirnov presented optimization tools for finding many fast algorithms based on factoring bilinear forms [31], and we will use these tools for finding our own algorithms in Section 2.

There are several lines of theoretical research (*cf.* [12, 36] and references therein) that prove existence of fast APA algorithms with much better asymptotic complexity than the algorithms considered here. Unfortunately, there remains a large gap between the substantial theoretical work and what we can practically implement.

Renewed interest in the practicality of Strassen's and other fast algorithms is motivated by the observation that not only is the arithmetic cost reduced when compared to the classical algorithm, the communication costs also improve asymptotically [4]. That is, as the relative cost of moving data throughout the memory hierarchy and between processors increases, we can expect the benefits of fast algorithms to grow accordingly. We note that communication lower bounds [4] apply to all the algorithms presented in this paper, and in most cases they are attained by the implementations used here.

1.2 Notation and Tensor Preliminaries

We briefly review basic tensor preliminaries, following the notation of Kolda and Bader [24]. Scalars are represented by lowercase Roman or Greek letters (a), vectors by lowercase boldface ($\mathbf{x}$), matrices by uppercase boldface ($\mathbf{A}$), and tensors by boldface Euler script letters ($\mathcal{T}$). For a matrix $\mathbf{A}$, we use $\mathbf{a}_k$ and a_{ij} to denote the kth column and i, j entry, respectively. A tensor is a multi-dimensional array, and in this paper we deal exclusively with order-3, real-valued tensors; *i.e.*, $\mathcal{T} \in \mathbb{R}^{I \times J \times K}$. The kth *frontal slice* of $\mathcal{T}$ is $\mathbf{T}_k = t_{:,:,k} \in \mathbb{R}^{I \times J}$. For $\mathbf{u} \in \mathbb{R}^I$, $\mathbf{v} \in \mathbb{R}^J$, $\mathbf{w} \in \mathbb{R}^K$, we define the outer product tensor $\mathcal{T} = \mathbf{u} \circ \mathbf{v} \circ \mathbf{w} \in \mathbb{R}^{I \times J \times K}$ with entries $t_{ijk} = u_i v_j w_k$. Addition of tensors is defined entry-wise. The *rank* of a tensor $\mathcal{T}$ is the minimum number of rank-one tensors that generate $\mathcal{T}$ as their sum. Decompositions of the form $\mathcal{T} = \sum_{r=1}^{R} \mathbf{u}_r \circ \mathbf{v}_r \circ \mathbf{w}_r$ lead to fast matrix multiplication algorithms (Section 2.2), and we use $[\![\mathbf{U}, \mathbf{V}, \mathbf{W}]\!]$ to denote the decomposition, where $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{W}$ are matrices with R columns given by $\mathbf{u}_r$, $\mathbf{v}_r$, and $\mathbf{w}_r$. Of the various flavors of products involving tensors, we will need to know that, for $\mathbf{a} \in \mathbb{R}^I$ and $\mathbf{b} \in \mathbb{R}^J$, $\mathcal{T} \times_1 \mathbf{a} \times_2 \mathbf{b} = \mathbf{c} \in \mathbb{R}^K$, with $c_k = \mathbf{a}^T \mathbf{T}_k \mathbf{b}$, or $c_k = \sum_{i=1}^{I} \sum_{j=1}^{J} t_{ijk} a_i b_j$.

2. Fast Matrix Multiplication

We now review the preliminaries for fast matrix multiplication algorithms. In particular, we focus on factoring tensor representations of bilinear forms, which will facilitate the discussion of the implementation in Sections 3 and 4.

2.1 Recursive Multiplication

Matrices are self-similar, *i.e.*, a submatrix is also a matrix. Arithmetic with matrices is closely related to arithmetic with scalars, and we can build recursive matrix multiplication algorithms by manipulating submatrix blocks. For example, consider multiplying $C = A \cdot B$,

$$\begin{bmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix},$$

where we have partitioned the matrices into four submatrices. Throughout this paper, we denote the block multiplication of $M \times K$ and $K \times N$ matrices by $\langle M, K, N \rangle$. Thus, the above computation is $\langle 2, 2, 2 \rangle$. Multiplication with the classical algorithm proceeds by

combining a set of eight matrix multiplications with four matrix additions:

$$\mathbf{M}_1 = \mathbf{A}_{11} \cdot \mathbf{B}_{11} \quad \mathbf{M}_2 = \mathbf{A}_{12} \cdot \mathbf{B}_{21} \quad \mathbf{M}_3 = \mathbf{A}_{11} \cdot \mathbf{B}_{12} \quad \mathbf{M}_4 = \mathbf{A}_{12} \cdot \mathbf{B}_{22}$$
$$\mathbf{M}_5 = \mathbf{A}_{21} \cdot \mathbf{B}_{11} \quad \mathbf{M}_6 = \mathbf{A}_{22} \cdot \mathbf{B}_{21} \quad \mathbf{M}_7 = \mathbf{A}_{21} \cdot \mathbf{B}_{12} \quad \mathbf{M}_8 = \mathbf{A}_{22} \cdot \mathbf{B}_{22}$$
$$\mathbf{C}_{11} = \mathbf{M}_1 + \mathbf{M}_2 \quad \mathbf{C}_{12} = \mathbf{M}_3 + \mathbf{M}_4 \quad \mathbf{C}_{21} = \mathbf{M}_5 + \mathbf{M}_6 \quad \mathbf{C}_{22} = \mathbf{M}_7 + \mathbf{M}_8$$

The multiplication to form each $\mathbf{M}_i$ is recursive and the base case is scalar multiplication. The number of flops performed by the classical algorithm for $N \times N$ matrices, where N is a power of two, is $2N^3 - N^2$.

The idea of fast matrix multiplication algorithms is to perform fewer recursive matrix multiplications at the expense of more matrix additions. Since matrix multiplication is asymptotically more expensive than matrix addition, this tradeoff results in faster algorithms. The most well known fast algorithm is due to Strassen, and follows the same block structure:

$$\mathbf{S}_1 = \mathbf{A}_{11} + \mathbf{A}_{22} \quad \mathbf{S}_2 = \mathbf{A}_{21} + \mathbf{A}_{22} \quad \mathbf{S}_3 = \mathbf{A}_{11} \quad \mathbf{S}_4 = \mathbf{A}_{22}$$
$$\mathbf{S}_5 = \mathbf{A}_{11} + \mathbf{A}_{12} \quad \mathbf{S}_6 = \mathbf{A}_{21} - \mathbf{A}_{11} \quad \mathbf{S}_7 = \mathbf{A}_{12} - \mathbf{A}_{22}$$
$$\mathbf{T}_1 = \mathbf{B}_{11} + \mathbf{B}_{22} \quad \mathbf{T}_2 = \mathbf{B}_{11} \quad \mathbf{T}_3 = \mathbf{B}_{12} - \mathbf{B}_{22} \quad \mathbf{T}_4 = \mathbf{B}_{21} - \mathbf{B}_{11}$$
$$\mathbf{T}_5 = \mathbf{B}_{22} \quad \mathbf{T}_6 = \mathbf{B}_{11} + \mathbf{B}_{12} \quad \mathbf{T}_7 = \mathbf{B}_{21} + \mathbf{B}_{22}$$
$$\mathbf{M}_r = \mathbf{S}_r \mathbf{T}_r, \quad 1 \le r \le 7$$
$$\mathbf{C}_{11} = \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \quad \mathbf{C}_{12} = \mathbf{M}_3 + \mathbf{M}_5$$
$$\mathbf{C}_{21} = \mathbf{M}_2 + \mathbf{M}_4 \quad \mathbf{C}_{22} = \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6$$

We have explicitly written out terms like $\mathbf{T}_2 = \mathbf{B}_{11}$ to hint at the generalizations provided in Section 2.2. Strassen's algorithm uses 7 matrix multiplications and 18 matrix additions. The number of flops performed by the algorithm is $7N^{\log_2 7} - 6N^2 = O(N^{2.81})$, when we assume a base case of $N = 1$.

There are natural extensions to Strassen's algorithm. We might try to find an algorithm using fewer than 7 multiplications; unfortunately, we cannot [37]. Alternatively, we could try to reduce the number of additions. This leads to the Strassen-Winograd algorithm, which reduces the 18 additions down to 15. We explore such methods in Section 3.3. We can also improve the constant on the leading term by choosing a bigger base case dimension (and using the classical algorithm for the base case). This turns out not to be important in practice because the base case will be chosen to optimize performance rather than flop count. Lastly, we can use blocking schemes apart from $\langle 2, 2, 2 \rangle$, which we explain in the remainder of this section. This leads to a host of new algorithms, and we show in Section 5 that they are often faster in practice.

2.2 Fast Algorithms as Low-Rank Tensor Decompositions

The approach we use to devise fast algorithms exploits an important connection between matrix multiplication (and other bilinear forms) and tensor computations. We detail the connection in this section for completeness; see [7, 23] for earlier explanations.

A bilinear form on a pair of finite-dimensional vector spaces is a function that maps a pair of vectors to a scalar and is linear in each of its inputs separately. A bilinear form $B(\mathbf{x}, \mathbf{y})$ can be represented by a matrix $\mathbf{D}$ of coefficients: $B(\mathbf{x}, \mathbf{y}) = \mathbf{x}^\mathsf{T}\mathbf{D}\mathbf{y} = \sum_i \sum_j d_{ij} x_i y_j$, where we note that $\mathbf{x}$ and $\mathbf{y}$ may have different dimensions. In order to describe a set of K bilinear forms $B_k(\mathbf{x}, \mathbf{y}) = z_k$, $1 \le k \le K$, we can use a three-way tensor $\mathcal{T}$ of coefficients:

$$z_k = \sum_{i=1}^{I} \sum_{j=1}^{J} t_{ijk} x_i y_j, \tag{1}$$

or, in more succinct tensor notation, $\mathbf{z} = \mathcal{T} \times_1 \mathbf{x} \times_2 \mathbf{y}$.

2.2.1 Low-Rank Tensor Decompositions

The advantage of representing the operations using a tensor of coefficients is a key connection between the rank of the tensor to the arithmetic complexity of the corresponding operation. Consider the "active" multiplications between elements of the input vectors (*e.g.*, $x_i \cdot y_j$). The conventional algorithm, following Equation (1), will compute an active multiplication for every nonzero coefficient in $\mathcal{T}$. However, suppose we have a rank-R decomposition of the tensor, $\mathcal{T} = \sum_{i=1}^{R} u_i \circ v_i \circ w_i$, so that

$$t_{ijk} = \sum_{r=1}^{R} u_{ir} v_{jr} w_{kr} \tag{2}$$

for all i, j, k, where $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{W}$ are matrices with R columns each. We will also use the equivalent notation $\mathcal{T} = [\![\mathbf{U}, \mathbf{V}, \mathbf{W}]\!]$. Substituting Equation (2) into Equation (1) and rearranging, we have for $k = 1, \ldots, K$, $z_k = \sum_{r=1}^{R} (s_r \cdot t_r) w_{kr} = \sum_{r=1}^{R} m_r w_{kr}$, where $\mathbf{s} = \mathbf{U}^\mathsf{T}\mathbf{x}$, $\mathbf{t} = \mathbf{V}^\mathsf{T}\mathbf{y}$, and $\mathbf{m} = \mathbf{s} * \mathbf{t}$.[2] This reduces the number of active multiplications (now between linear combinations of elements of the input vectors) to R. Here we highlight active multiplications with $(\cdot)$ notation.

Assuming $R < \text{nnz}(\mathcal{T})$, this reduction of active multiplications, at the expense of increasing the number of other operations, is valuable when active multiplications are much more expensive than the other operations. This is the case for recursive matrix multiplication, where the elements of the input vectors are (sub)matrices, as we describe below.

2.2.2 Tensor Representation of Matrix Multiplication

Matrix multiplication is a bilinear operation, so we can represent it as a tensor computation. In order to match the notation above, we vectorize the input and output matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ using row-wise ordering, so that $\mathbf{x} = \text{vec}(\mathbf{A})$, $\mathbf{y} = \text{vec}(\mathbf{B})$, and $\mathbf{z} = \text{vec}(\mathbf{C})$.

For every triplet of matrix dimensions for valid matrix multiplication, there is a fixed tensor that represents the computation so that $\mathcal{T} \times_1 \text{vec}(\mathbf{A}) \times_2 \text{vec}(\mathbf{B}) = \text{vec}(\mathbf{C})$ holds for all $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$. For example, if $\mathbf{A}$ and $\mathbf{B}$ are both 2×2, the corresponding $4 \times 4 \times 4$ tensor $\mathcal{T}$ has frontal slices

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

This yields, for example, $\mathbf{T}_3 \times_1 \text{vec}(\mathbf{A}) \times_2 \text{vec}(\mathbf{B}) = a_{21} \cdot b_{11} + a_{22} \cdot b_{21} = c_{21}$.

By Strassen's algorithm, we know that although this tensor has 8 nonzero entries, its rank is at most 7. Indeed, that algorithm corresponds to a low-rank decomposition represented by the following triplet of matrices, each with 7 columns:

$$\mathbf{U} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & -1 \end{bmatrix}, \mathbf{V} = \begin{bmatrix} 1 & 1 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & -1 & 0 & 1 & 0 & 1 \end{bmatrix},$$

$$\mathbf{W} = \begin{bmatrix} 1 & 0 & 0 & 1 & -1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

As in the previous section, for example, $s_1 = \mathbf{u}_1^\mathsf{T}\text{vec}(\mathbf{A}) = a_{11} + a_{22}$, $t_1 = \mathbf{v}_1^\mathsf{T}\text{vec}(\mathbf{B}) = b_{11} + b_{22}$, and $c_{11} = (\mathbf{W}\mathbf{m})_1 = m_1 + m_4 - m_5 + m_7$. Note that in the previous section, the elements of the input matrices are already interpreted as submatrices (*e.g.*, $\mathbf{A}_{11}$ and $\mathbf{M}_1$); here we represent them as scalars (*e.g.*, a_{11} and m_1).

We need not restrict ourselves to the $\langle 2, 2, 2 \rangle$ case; there exists a tensor for matrix multiplication given any set of valid dimensions. When considering a base case of $M \times K$ by $K \times N$ matrix multiplication (denoted $\langle M, K, N \rangle$), the tensor has dimensions $MK \times KN \times MN$ and MKN non-zeros. In particular, $t_{ijk} = 1$ if the following three conditions hold: (1) $(i - 1) \mod K = \lfloor (j - 1)/N \rfloor$; (2) $(j - 1) \mod N = (k - 1) \mod N$; and (3) $\lfloor (i - 1)/K \rfloor = \lfloor (k - 1)/N \rfloor$. Otherwise, $t_{ijk} = 0$ (here we assume entries are 1-indexed).

[2] Here $(*)$ denotes element-wise vector multiplication.

2.2.3 Approximate Tensor Decompositions

The APA algorithms discussed in Section 1.1 arise from *approximate* tensor decompositions. With Bini's algorithm, for example, the factor matrices (*i.e.*, the corresponding $[\![\mathbf{U}, \mathbf{V}, \mathbf{W}]\!]$) have entries $1/\lambda$ and λ. As $\lambda \to 0$, the low-rank tensor approximation approaches the true tensor. However, as λ gets small, we suffer from loss of precision in the floating point calculations of the resulting fast algorithm. Setting $\lambda = \sqrt{\epsilon}$ minimizes the loss of accuracy for one step of Bini's algorithm, where ϵ is machine precision [6], but even in this case at least half the digits are lost with a single recursive step of the algorithm.

2.3 Finding Fast Algorithms

We conclude this section with a description of a method for searching for and discovering fast algorithms for matrix multiplication. Our search goal is to find low-rank decompositions of tensors corresponding to matrix multiplication of a particular set of dimensions, which will identify fast, recursive algorithms with reduced arithmetic complexity. That is, given a particular base case $\langle M, K, N\rangle$ and the associated tensor $\mathcal{T}$, we seek a rank R and matrices $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{W}$ that satisfy Equation (1). Table 1 summarizes the algorithms that we find and use for numerical experiments in Section 5.

The rank of the decomposition determines the number of active multiplications, or recursive calls, and therefore the exponent in the arithmetic cost of the algorithm. The number of other operations (additions and inactive multiplications) will affect only the constants in the arithmetic cost. For this reason, we want sparse $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{W}$ matrices with simple values (like ± 1), but that goal is of secondary importance compared to minimizing the rank R. Note that these constant values do affect performance of these algorithms for reasonable matrix dimensions in practice, though mainly because of how they affect the communication costs of the implementations rather than the arithmetic cost. We discuss this in more detail in Section 3.2.

2.3.1 Equivalent Algorithms

Given an algorithm $[\![\mathbf{U}, \mathbf{V}, \mathbf{W}]\!]$ for base case $\langle M, K, N\rangle$, we can transform it to an algorithm for any of the other 5 permutations of the base case dimensions with the same number of multiplications. This is a well known property [15]; here we state the two transformations that generate all permutations in our notation. We let $\mathbf{P}_{I\times J}$ be the permutation matrix that swaps row-order for column-order in the vectorization of an $I \times J$ matrix. In other words, if $\mathbf{A}$ is $I \times J$, $\mathbf{P}_{I\times J} \cdot \text{vec}(\mathbf{A}) = \text{vec}\left(\mathbf{A}^{\mathsf{T}}\right)$.

Proposition 2.1. *Given a fast algorithm $[\![\mathbf{U}, \mathbf{V}, \mathbf{W}]\!]$ for $\langle M, K, N\rangle$, $[\![\mathbf{P}_{K\times N}\mathbf{V}, \mathbf{P}_{M\times K}\mathbf{U}, \mathbf{P}_{M\times N}\mathbf{W}]\!]$ is a fast algorithm for $\langle N, K, M\rangle$.*

Proposition 2.2. *Given a fast algorithm $[\![\mathbf{U}, \mathbf{V}, \mathbf{W}]\!]$ for $\langle M, K, N\rangle$, $[\![\mathbf{P}_{M\times N}\mathbf{W}, \mathbf{U}, \mathbf{P}_{K\times N}\mathbf{V}]\!]$ is a fast algorithm for $\langle N, M, K\rangle$.*

Fast algorithms for a given base case also belong to equivalence classes. Two algorithm are equivalent if one can be generated from another based on the following transformations [10, 21].

Proposition 2.3. *If $[\![\mathbf{U}, \mathbf{V}, \mathbf{W}]\!]$ is a fast algorithm for $\langle M, K, N\rangle$, then the following are also fast algorithms for $\langle M, K, N\rangle$:*

$$[\![\mathbf{UP}, \mathbf{VP}, \mathbf{WP}]\!]$$

for any permutation matrix $\mathbf{P}$;

$$[\![\mathbf{UD}_x, \mathbf{VD}_y, \mathbf{WD}_z]\!]$$

for any diagonal matrices $\mathbf{D}_x$, $\mathbf{D}_y$, and $\mathbf{D}_z$ such that $\mathbf{D}_x\mathbf{D}_y\mathbf{D}_z = \mathbf{I}$;

$$[\![(\mathbf{Y}^{-\mathsf{T}} \otimes \mathbf{X})\mathbf{U}, (\mathbf{Z}^{-\mathsf{T}} \otimes \mathbf{Y})\mathbf{V}, (\mathbf{X} \otimes \mathbf{Z}^{-\mathsf{T}})\mathbf{W}]\!]$$

for any nonsingular matrices $\mathbf{X} \in \mathbb{R}^{M\times M}$, $\mathbf{Y} \in \mathbb{R}^{K\times K}$, $\mathbf{Z} \in \mathbb{R}^{N\times N}$.

Table 1. Summary of fast algorithms. Algorithms without citation were found by the authors using the ideas in Section 2.3. An asterisk denotes an approximation (APA) algorithm. The number of multiplications is equal to the rank R of the corresponding tensor decomposition. The multiplication speedup per recursive step is the expected speedup if matrix additions were free. This speedup does not determine the fastest algorithm because the maximum number of recursive steps depend on the size of the sub-problems created by the algorithm. By Propositions 2.1 and 2.2, we also have fast algorithms for all permutations of the base case $\langle M, K, N\rangle$.

Algorithm base case	Number of multiplies (fast)	Number of multiplies (classical)	Multiplication speedup per recursive step
$\langle 2, 2, 3\rangle$	11	12	9%
$\langle 2, 2, 5\rangle$	18	20	11%
$\langle 2, 2, 2\rangle$ [32]	7	8	14%
$\langle 2, 2, 4\rangle$	14	16	14%
$\langle 3, 3, 3\rangle$	23	26	17%
$\langle 2, 3, 3\rangle$	15	18	20%
$\langle 2, 3, 4\rangle$	20	24	20%
$\langle 2, 4, 4\rangle$	26	32	23%
$\langle 3, 3, 4\rangle$	29	36	24%
$\langle 3, 4, 4\rangle$	38	48	26%
$\langle 3, 3, 6\rangle$ [31]	40	54	35%
$\langle 2, 2, 3\rangle$* [5]	10	12	20%
$\langle 3, 3, 3\rangle$* [30]	21	27	29%

2.3.2 Numerical Search

Given a rank R for base case $\langle M, K, N\rangle$, Equation (1) defines $(MKN)^2$ polynomial equations of the form given in Equation (2). Because the polynomials are trilinear, alternating least squares (ALS) can be used to iteratively compute an approximate (numerical) solution to the equations. That is, if two of the three factor matrices are fixed, the optimal third factor matrix is the solution to a linear least squares problem. Thus, each outer iteration of ALS involves alternating among solving for $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{W}$, each of which can be done efficiently with a QR decomposition, for example. This approach was first proposed for fast matrix multiplication search by Brent [7], but ALS has been a popular method for general low-rank tensor approximation for as many years (see [24] and references therein).

The main difficulties ALS faces for this problem include getting stuck at local minima, encountering ill-conditioned linear least-squares problems, and, even if ALS converges to machine-precision accuracy, computing dense $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{W}$ matrices with floating point entries. We follow the work of Johnson and McLoughlin [21] and Smirnov [31] in addressing these problems. We use multiple starting points to handle the problem of local minima, add regularization to help with the ill-conditioning, and encourage sparsity in order to recover exact factorizations (with integral or rational values) from the approximations.

The most useful techniques in our search have been (1) exploiting equivalence transformations [21, Eq. (6)] to encourage sparsity and obtain discrete values and (2) using and adjusting the regularization penalty term [31, Eq. (4-5)] throughout the iteration. As described in earlier efforts, algorithms for small base cases can be discovered nearly automatically. However, as the values M, N, and K grow, more hands-on tinkering using heuristics seems to be necessary to find discrete solutions.

3. Implementation and Practical Considerations

We now discuss our code generation method for fast algorithms and the major implementation issues. All experiments were conducted on a single compute node on NERSC's Edison. Each node has two 12-core Intel 2.4 GHz Ivy Bridge processors and 64 GB of memory.

3.1 Code Generation

Our code generator automatically implements a fast algorithm in C++ given the **U**, **V**, and **W** matrices representing the algorithm. The generator simultaneously produces both sequential and parallel implementations. We discuss the sequential code in this section and the parallel extensions in Section 4. For computing $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$, the following are the key ingredients of the generated code:

- Using the entries in the **U** and **V** matrices, form the temporary matrices $\mathbf{S}_r$ and $\mathbf{T}_r$, $1 \le r \le R$, via matrix additions and scalar multiplication. The $\mathbf{S}_r$ and $\mathbf{T}_r$ are linear combinations of sub-blocks of **A** and **B**, respectively. For each $\mathbf{S}_r$ and $\mathbf{T}_r$, the corresponding linear combination is a custom implementation. Scalar multiplication by ± 1 is replaced with native addition / subtraction operators. The code generator can produce three variants of matrix additions, which we describe in Section 3.2.

  When a column of **U** or **V** contains a single non-zero element, there is no matrix addition (only scalar multiplication). In order to save memory, the code generator does not form a temporary matrix in this case. The scalar multiplication is piped through to subsequent recursive calls and is eventually used in a base case call to `dgemm`.
- Recursive calls to the fast matrix multiplication routine compute $\mathbf{M}_r = \mathbf{S}_r \cdot \mathbf{T}_r$, $1 \le r \le R$.
- Using the entries of **W**, linear combinations of the $\mathbf{M}_r$ form the output **C**. Matrix additions and scalar multiplications are again handled carefully, as above.
- Common subexpression elimination detects redundant matrix additions, and the code generator can automatically implement algorithms with fewer additions. We discuss this process in more detail in Section 3.3.
- Dynamic peeling [33] accounts for matrices whose dimensions are not evenly divided by the base case of the fast algorithm. This method handles the boundaries of the matrix at each recursive level, and requires no additional memory. (Other methods, such as zero-padding, require additional memory). With dynamic peeling, the implementation can multiply matrices of any dimensions.

Figure 1 shows performance benchmarks of the code generator's implementation. In order to compare the performance of matrix multiplication algorithms with different computational costs, we use the *effective* GFLOPS metric for $P \times Q \times R$ matrix multiplication:

$$\text{effective GFLOPS} = \frac{2PQR - PR}{\text{time in seconds}} \cdot \text{1e-9}. \tag{3}$$

We note that effective GFLOPS is only the true GFLOPS for the classical algorithm (the fast algorithms perform fewer floating point operations). However, this metric lets us compare all of the algorithms on an inverse-time scale, normalized by problem size [13, 27].

We compare our code-generated Strassen implementation with MKL's `dgemm` and a tuned implementation of Strassen-Winograd from D'Alberto *et al.* [9] (recall that Strassen-Winograd performs the same number of multiplications but fewer matrix additions than Strassen's algorithm). The code generator's implementation outperforms MKL and is competitive with the tuned implementation.

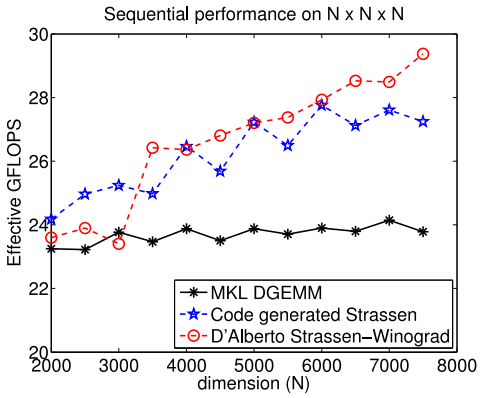

Figure 1. Effective performance (Equation (3)) of our code generator's implementation of Strassen's algorithm against MKL's `dgemm` and a tuned implementation of the Strassen-Winograd algorithm [9]. The problems sizes are square. The generated code easily outperforms MKL and is competitive with the tuned code.

Thus, we are confident that the general conclusions we draw with code-generated implementations of fast algorithms will also apply to hand-tuned implementations.

3.2 Handling Matrix Additions

While the matrix multiplications constitute the bulk of the running time, matrix additions are still an important performance optimization. We call the linear combinations used to form $\mathbf{S}_r$, $\mathbf{T}_r$, and $\mathbf{C}_{ij}$ *addition chains*. For example, $\mathbf{S}_1 = \mathbf{A}_{11} + \mathbf{A}_{22}$ is an addition chain in Strassen's algorithm. We consider three different implementations for the addition chains:

1. **Pairwise**: With r fixed, compute $\mathbf{S}_r$ and $\mathbf{T}_r$ using the `daxpy` BLAS routine for all matrices in the addition chain. This requires $\text{nnz}(\mathbf{u}_r)$ calls to `daxpy` to form $\mathbf{S}_r$ and $\text{nnz}(\mathbf{v}_r)$ calls to form $\mathbf{T}_r$. After the recursive computations of the $\mathbf{M}_r$, we follow the same strategy to form the output. The ith sub-block (row-wise) of **C** requires $\text{nnz}(w_{i,:})$ `daxpy` calls.[3]
2. **Write-once**: With r fixed, compute $\mathbf{S}_r$ and $\mathbf{T}_r$ with only one write for each entry (instead of, for example, $\text{nnz}(\mathbf{v}_r)$ writes for $\mathbf{S}_r$ with the pairwise method). In place of `daxpy`, stream through the necessary submatrices of **A** and **B** and combine the entries to form $\mathbf{S}_r$ and $\mathbf{T}_r$. This requires reading some submatrices of **A** and **B** several times, but writing to only one output stream at a time. Similarly, we write the output matrix **C** once and read the $\mathbf{M}_r$ several times.
3. **Streaming**: Read each input matrix once and write each temporary matrix $\mathbf{S}_r$ and $\mathbf{T}_r$ once. Stream through the entries of each sub-block of **A** and **B**, and update the corresponding entries in *all* temporary matrices $\mathbf{S}_r$ and $\mathbf{T}_r$. Similarly, stream through the entries of the $\mathbf{M}_r$ and update *all* submatrices of **C**.

Each `daxpy` call requires two matrix reads and one matrix write (except for the first call in an addition chain, which is a copy and requires one read and one write). Let $\text{nnz}(\mathbf{U}, \mathbf{V}, \mathbf{W}) = \text{nnz}(\mathbf{U}) + \text{nnz}(\mathbf{V}) + \text{nnz}(\mathbf{W})$. Then the pairwise additions perform $2 \cdot \text{nnz}(\mathbf{U}, \mathbf{V}, \mathbf{W}) - 2R - MN$ submatrix reads and $\text{nnz}(\mathbf{U}, \mathbf{V}, \mathbf{W})$ submatrix writes. However, the additions use an efficient vendor implementation.

[3] Because `daxpy` computes $y \leftarrow \alpha x + y$, we make a call for each addition in the chain as well as one call for an initial copy.

The write-once additions perform $\text{nnz}(\mathbf{U}, \mathbf{V}, \mathbf{W})$ submatrix reads and at most $2R + MN$ submatrix writes. We do not need to write any data for the columns of $\mathbf{U}$ and $\mathbf{V}$ with a single nonzero entry. These correspond to addition chains that are just a copy, for example, $\mathbf{T}_2 = \mathbf{B}_{11}$ in Strassen's algorithm. While we perform fewer reads and writes than the pairwise additions, the complexity of our code increases (we have to write our own additions), and we can no longer use a tuned `daxpy` routine. We do not worry about code complexity because we use code generation. Since the problem is bandwidth-bound and compilers can automatically vectorize `for` loops, we don't expect the latter concern to be an issue.

Finally, the streaming additions perform $MK+KN+R$ submatrix reads and at most $2R + MN$ submatrix writes. This is fewer reads than the write-once additions, but we have increased the complexity of the writes. Specifically, we alternate writes to different memory locations, whereas with the write-once algorithm, we write to a single (contiguous) output stream.

The three methods also have different memory footprints. With pairwise or write-once, $\mathbf{S}_r$ and $\mathbf{T}_r$ are formed just before computing $\mathbf{M}_r$. After $\mathbf{M}_r$ is computed, the memory becomes available. On the other hand, the streaming algorithm must compute all temporary matrices $\mathbf{S}_r$ and $\mathbf{T}_r$ simultaneously, and hence needs R times as much memory for the temporary matrices. We will explore the performance of the three methods at the end of Section 3.3.

3.3 Common Subexpression Elimination

The $\mathbf{S}_r$, $\mathbf{T}_r$, and $\mathbf{M}_r$ matrices often share subexpressions. For example, in our $\langle 4, 2, 4\rangle$ fast algorithm (see Table 1), $\mathbf{T}_{11}$ and $\mathbf{T}_{25}$ are:

$$\mathbf{T}_{11} = \mathbf{B}_{24} - \mathbf{B}_{12} - \mathbf{B}_{22} \qquad \mathbf{T}_{25} = \mathbf{B}_{23} + \mathbf{B}_{12} + \mathbf{B}_{22}$$

Both $\mathbf{T}_{11}$ and $\mathbf{T}_{25}$ share the subexpression $\mathbf{B}_{12} + \mathbf{B}_{22}$, up to scalar multiplication. Thus, there is opportunity to remove additions / subtractions:

$$\mathbf{Y}_1 = \mathbf{B}_{12} + \mathbf{B}_{22} \qquad \mathbf{T}_{11} = \mathbf{B}_{24} - \mathbf{Y}_1 \qquad \mathbf{T}_{25} = \mathbf{B}_{23} + \mathbf{Y}_1$$

At face value, eliminating additions would appear to improve the algorithm. However, there are two important considerations. First, using $\mathbf{Y}_1$ with the pairwise or write-once approaches requires additional memory (with the streaming approach it requires only additional local variables).

Second, we discussed in Section 3.2 that an important metric is the number of reads and writes. If we use the write-once algorithm, we have actually *increased* the number of reads and writes. Originally, forming $\mathbf{T}_{11}$ and $\mathbf{T}_{25}$ required six reads and two writes. By eliminating the common subexpression, we performed two fewer reads in forming $\mathbf{T}_{11}$ and $\mathbf{T}_{25}$ but needed an additional two reads and one write to form $\mathbf{Y}_1$. In other words, we have read the same amount of data and written *more* data. In general, eliminating the same length-two subexpression k times reduces the number of matrix reads and writes by $k - 3$. Thus, a length-two subexpression must appear at least four times for elimination to reduce the total number of reads and writes in the algorithm.

Figure 2 shows the performance all three matrix addition methods from Section 3.2, with and without common subexpression elimination (CSE). For CSE, we greedily eliminate length-two subexpressions. In general, the write-once algorithm without CSE performs the best on the rectangular matrix multiplication problem sizes. For these problems, CSE lowers performance of the write-once algorithm and has little to modest effect on the streaming and pairwise algorithms. For square matrix problems, the best variant is less clear, but write-once with no elimination often performs the highest. We use write-once without CSE for the rest of our performance experiments.

3.4 Recursion Cutoff Point

In practice, we take only a few steps of recursion before calling a vendor-tuned library classical routine as the base case (in our case, Intel MKL's `dgemm`). One method for determining the cutoff point is to benchmark each algorithm and measure where the implementation outperforms `dgemm`. While this is sustainable for the analysis of any individual algorithm, we are interested in a large class of fast algorithms. Furthermore, a simple set of cutoff points limits understanding of the performance and will have to be re-measured for different architectures. Instead, we provide a rule of thumb based on the performance of `dgemm`.

Figure 3 shows the performance of Intel MKL's sequential and parallel `dgemm` routines. We see that the routines exhibit a "ramp-up" phase and then flatten for sufficiently large problems. In both serial and parallel, multiplication of square matrices ($N \times N \times N$ computation) tends to level at a higher performance than the problem shapes with a fixed dimension ($N \times 800 \times N$ and $N \times 800 \times 800$). Our principle for recursion is to take a recursive step only if the sub-problems fall on the flat part of the curve. If the ratio of performance drop in the DGEMM curve is greater than the speedup per step (as listed in Table 1), then taking an additional recursive step cannot improve performance.[4] Finally, we note that some of our parallel algorithms call the sequential `dgemm` routine in the base case. Both curves will be important to our parallel fast matrix multiplication algorithms in Section 4.

4. Parallel Algorithms for Shared Memory

We present three algorithms for parallel fast matrix multiplication: depth-first search (DFS), breadth-first search (BFS), and a hybrid of the two (HYBRID). In this work, we target shared memory machines, although the same ideas generalize to distributed memory. For example, DFS and BFS ideas are used for a distributed memory implementation of Strassen's algorithm [27].

4.1 Depth-First Search

The DFS algorithm is straightforward: when recursion stops, the classical algorithm uses all threads on each sub-problem. In other words, we use parallel matrix multiplication on the leaf nodes of a depth-first traversal of the recursion tree. At a high-level, the code path is exactly the same as in the sequential case, and the main parallelism is in library calls. The advantages of DFS are that the memory footprint matches the sequential algorithm and the code is simpler—parallelism in multiplications is hidden inside library calls. Furthermore, matrix additions are trivially parallelized. The key disadvantage of DFS is that the recursion cutoff point is larger (Figure 3), limiting the number of recursive steps. On Edison's 24-core compute node, the recursion cutoff point is around $N = 5000$.

4.2 Breadth-First Search

The BFS algorithm uses task-based parallelism. Each leaf node in the matrix multiplication recursion tree is an independent task. The recursion tree also serves as a dependency graph: we need to compute all $\mathbf{M}_r$, $1 \le r \le R$, (children) before forming the result (parent). The major advantage of BFS is we can take more recursive steps because the recursion cutoff point is based on the sequential `dgemm` curves. Matrix additions to form $\mathbf{S}_r$ and $\mathbf{T}_r$ are part of the task that computes $\mathbf{M}_r$. In the first level of recursion, matrix additions to form $\mathbf{C}_{ij}$ from the $\mathbf{M}_r$ are handled in the same way as DFS, since all threads are available.

The BFS approach has two distinct disadvantages. First, it is difficult to load balance the tasks because the number of threads

[4] Note that the inverse is not necessarily true, the speedup depends on the overhead of the additions.

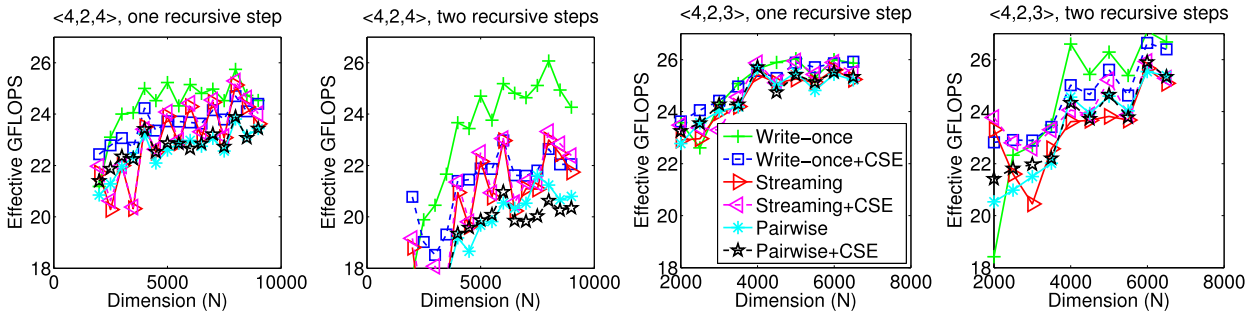

Figure 2. Effective performance (Equation (3)) comparison of common subexpression elimination (CSE) and the three matrix addition methods: write-once, streaming, and pairwise (see Section 3.2). The $\langle 4, 2, 4\rangle$ fast algorithm computed $N \times 1600 \times N$ ("outer product" shape) for varying N, and the $\langle 4, 2, 3\rangle$ fast algorithm computed $N \times N \times N$ (square multiplication). Write-once with no CSE tends to have the highest performance, especially for the $\langle 4, 2, 4\rangle$ algorithm. Pairwise is slower because it performs more reads and writes.

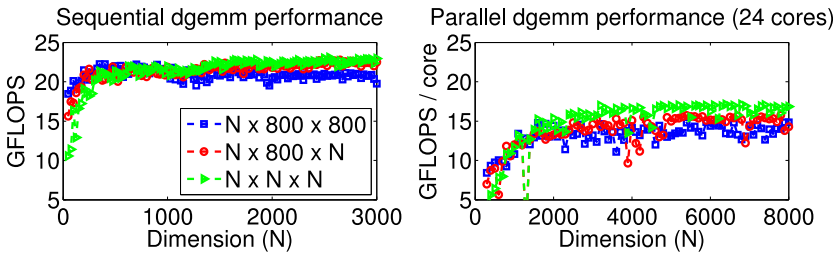

Figure 3. Performance curves of MKL's `dgemm` routine in serial (left) and in parallel (right) for three different problem shapes. The performance curves exhibit a "ramp-up" phase and then flatten for large enough problems. Performance levels near $N = 1500$ in serial and $N = 5000$ in parallel. For large problems in both serial and parallel, $N \times N \times N$ multiplication is faster than $N \times 800 \times N$, which is faster than $N \times 800 \times 800$. We note that sequential performance is faster than per-core parallel performance due to Intel Turbo Boost, which increases the clock speed from 2.4 to 3.2 GHz. With Turbo Boost, peak sequential performance is 25.6 GFLOPS. Peak parallel performance is 19.2 GFLOPS/core.

may not divide the number of tasks evenly. Also, with only one step of recursion, the number of tasks can be smaller than the number of threads. For example, one step of Strassen's algorithm produces only 7 tasks and one step of the fast $\langle 3, 2, 3\rangle$ algorithm produces only 15 tasks. Second, BFS requires additional memory since the tasks are executed independently. In a fast algorithm for $\langle M, K, N\rangle$ with R multiplies, each recursive step requires a factor $R/(MN)$ more memory than the output matrix $\mathbf{C}$ to store the $\mathbf{M}_r$. There are additional memory requirements for the $\mathbf{S}_r$ and $\mathbf{T}_r$ matrices, as discussed in Section 3.2.

4.3 Hybrid

Our novel hybrid algorithm compensates for the load imbalance in BFS by applying the DFS approach on a subset of the base case problems. With L levels of recursion and P threads, the hybrid algorithm applies task parallelism (BFS) to the first $R^L - (R^L \mod P)$ multiplications. The number of BFS sub-problems is a multiple of P, so this part of the algorithm is load balanced. All threads are used on each of the $R^L \mod P$ remaining multiplications (DFS).

An alternative approach uses another level of hybridization: evenly assign as many as possible of the remaining $R^L \mod P$ multiplications to disjoint subsets of $P' < P$ threads (where P' divides P), and then finish off the still-remaining multiplications with all P threads. This approach reduces the number of small multiplications assigned to all P threads where perfect scaling is harder to achieve. However, it leads to additional load balancing concerns in practice and requires a more complicated task scheduler.

4.4 Implementation

The code generation from Section 3.1 produces code that can compile to the DFS, BFS, or HYBRID parallel algorithms. We use OpenMP to implement each algorithm. The overview of the parallelization is:

- **DFS**: Each `dgemm` call uses all threads. Matrix additions are always fully parallelized.
- **BFS**: Each recursive matrix multiplication routine and the associated matrix additions are launched as an OpenMP task. At each recursive level, the `taskwait` barrier ensures that all $\mathbf{M}_r$ matrices are available to form the output matrix.
- **HYBRID**: Matrix multiplies are either launched as an OpenMP task (BFS), or the number of MKL threads is adjusted for a parallel `dgemm` (DFS). This is implemented with the `if` conditional clause of OpenMP tasks. Again, `taskwait` barriers ensure that the $\mathbf{M}_r$ matrices are computed before forming the output matrix. We use an explicit synchronization scheme with OpenMP locks to ensure that the DFS steps occur *after* the BFS tasks complete. This ensures that there is no oversubscription of threads.

4.5 Shared-Memory Bandwidth Limitations

The performance gains of the fast algorithms rely on the cost of matrix multiplications to be much larger than the cost of matrix additions. Since matrix multiplication is compute-bound and matrix addition is bandwidth-bound, these computations scale differently with the amount of parallelism. For large enough matrices, MKL's `dgemm` achieves near-peak performance of the node (Figure 3). On

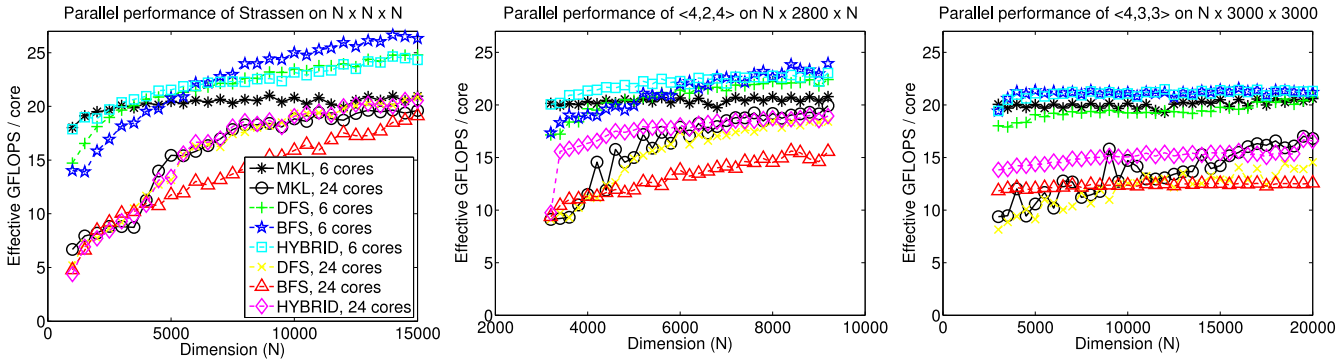

Figure 4. Effective performance (Equation (3)) comparison of the BFS, DFS, and HYBRID parallel implementations on representative fast algorithms and problem sizes. We use 6 and 24 cores to show the bandwidth limitations of the matrix additions. (Left): Strassen's algorithm on square problems. With 6 cores, we see significant speedups on large problems. (Middle): The $\langle 4, 2, 4\rangle$ fast algorithm (26 multiplies) on $N \times 2800 \times N$ problems. HYBRID performs the best in all cases. With 6 cores, the fast algorithm consistently outperforms MKL. With 24 cores, the fast algorithm can achieve significant speedups for small problem sizes. (Right): The $\langle 4, 3, 3\rangle$ fast algorithm (29 multiplies) on $N \times 3000 \times 3000$ problems. HYBRID again performs the best. With 6 cores, the fast algorithm gets modest speedups over MKL.

the other hand, the STREAM benchmark [28] shows that the node achieves around a five-fold speedup in bandwidth with 24 cores. In other words, in parallel, matrix multiplication is near 100% parallel efficiency and matrix addition is near 20% parallel efficiency. The bandwidth bottleneck makes it more difficult for parallel fast algorithms to be competitive with parallel MKL. To illuminate this issue, we will present performance results with both 6 and 24 cores. Using 6 cores avoids the bandwidth bottleneck and leads to much better performance per core.

4.6 Performance Comparisons

Figure 4 shows the performance of the BFS, DFS, and HYBRID parallel methods with both 6 and 24 cores for three representative algorithms. The left plot shows the performance of Strassen's algorithm on square problems. With 6 cores, HYBRID does the best for small problems. Since Strassen's algorithm uses 7 multiplies, BFS has poor performance with 6 cores when using one step of recursion. While all 6 cores can do 6 multiplies in parallel, the 7th multiply is done sequentially (with HYBRID, the 7th multiply uses all 6 cores). With two steps of recursion, BFS has better load balance but is forced to work on smaller sub-problems. As the problems get larger, BFS outperforms HYBRID due to synchronization overhead when HYBRID switches from BFS to DFS steps. When the matrix dimension is around 15,000, the fast algorithm achieves a 25% speedup over MKL. Using 24 cores, HYBRID and DFS are the fastest. With one step of recursion, BFS can achieve only sevenfold parallelism. With two steps, there are 49 sub-problems, so one core is assigned 3 sub-problems while all others are assigned 2. In general, we see that it is much more difficult to achieve speedups with 24 cores. However, Strassen's algorithm has a modest performance gain over MKL for large problem sizes ($\sim$ 5% faster).

The middle plot of Figure 4 shows the $\langle 4, 2, 4\rangle$ fast algorithm (26 multiplies) for $N \times 2800 \times N$ problems. With 6 cores, HYBRID is fastest for small problems and BFS becomes competitive for larger problems, where the performance is 15% better than MKL. In Section 5, we show that $\langle 4, 2, 4\rangle$ is also faster than Strassen's algorithm for these problems. With 24 cores, we see that HYBRID is drastically faster than MKL on small problems. For example, HYBRID is 75% faster on $3500 \times 2800 \times 3500$.[5] As the problem sizes get larger, we experience the bandwidth bottleneck and HYBRID achieves around the same performance as MKL. BFS uses one step of recursion and is consistently slower since it parallelizes 24 of 26 multiplies and uses only 2 cores on the last 2 multiplies. While multiple steps of recursion creates more load balance, the sub-problems are small enough that performance degrades even more. DFS follows a similar ramp-up curve as MKL, but the sub-problems are still too small to see a performance benefit.

The right plot of Figure 4 shows the $\langle 4, 3, 3\rangle$ fast algorithm (29 multiplies) for $N \times 3000 \times 3000$. We see similar trends as for the other problem sizes. With 6 cores, HYBRID does well for all problem sizes. Speedups are around $\sim$ 5% for large problems. With 24 cores, HYBRID is again drastically faster than MKL for small problem sizes and about the same as MKL for large problems.

5. Performance Experiments

We now present performance results for a variety of fast algorithms on several problem sizes. Based on the results of Section 4.5, we take the best of BFS and HYBRID when using 6 cores and the best of DFS and HYBRID when using 24 cores. For rectangular problem sizes in both sequential and parallel, we take the best of one or two steps of recursion. And for square problem sizes, we take the best of one, two, or three steps of recursion. Additional recursive steps do not improve the performance for the problem sizes we consider.

The square problem sizes for parallel benchmarks require the most memory—for some algorithms, three steps of recursion results in out-of-memory errors. In these cases, the original problem consumes 6% of the memory. For these algorithms, we only record the best of one or two steps of recursion in the performance plots. Finally, all timings are the median of five trials.

5.1 Sequential Performance

Figures 5 and 6 summarize the sequential performance of several fast algorithms. For $N \times N \times N$ problems (Figure 5), we test the algorithms in Table 1 and some of their permutations. For example, we test $\langle 4, 4, 2\rangle$ and $\langle 4, 2, 4\rangle$, which are permutations of $\langle 2, 4, 4\rangle$. In total, over 20 algorithms are tested for square matrices. Two of these algorithms, Bini's $\langle 3, 2, 2\rangle$ and Schönhage's $\langle 3, 3, 3\rangle$ are APA algorithms. We note that APA algorithms are of limited practical interest; even one step of recursion causes numerical errors in at least

[5] This result is an artifact of MKL's parallelization on these problem sizes and is not due to the speedups of the fast algorithm. We achieved similar speedups using our code generator and a classical, $\langle 2, 3, 4\rangle$ recursive algorithm (24 multiplies).

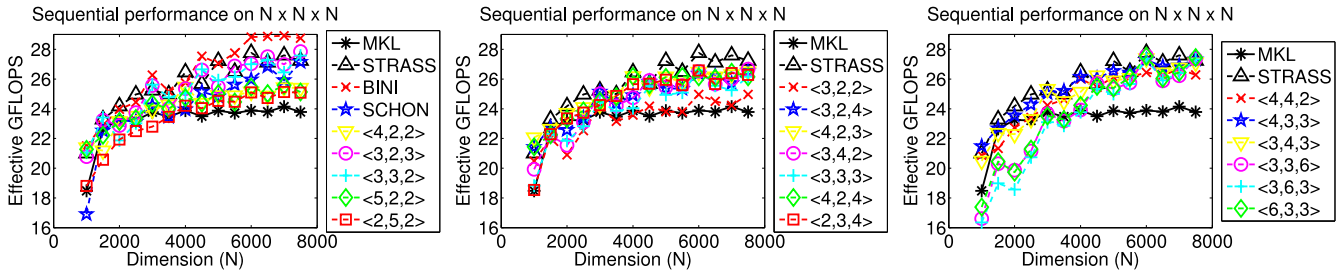

Figure 5. Effective sequential performance (Equation (3)) of a variety of fast algorithms on $N \times N \times N$ problem sizes distributed across three plots. Each data point is the best of one, two, or three steps of recursion—additional recursive steps did not improve performance. Bini and Schönhage are approximate algorithms, and all others are exact fast algorithms. MKL and Strassen's are repeated on all three plots for comparison. All of the fast algorithms outperform MKL for large enough problem sizes, and Strassen's algorithm usually performs the best.

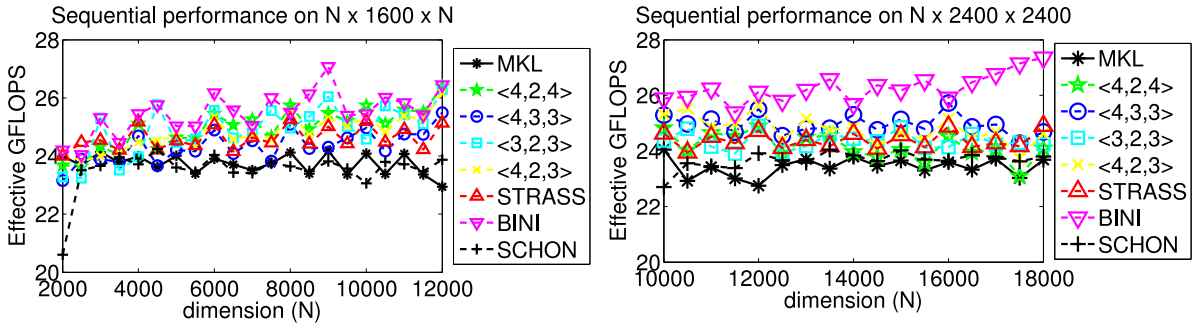

Figure 6. Effective sequential performance (Equation (3)) of fast matrix multiplication algorithms on rectangular problem sizes. (Left): Performance on an "outer product" shape, $N \times 1600 \times N$. Exact fast algorithms that have a similar outer product shape (*e.g.*, $\langle 4, 2, 4 \rangle$) tend to have the highest performance. (Right): Performance of multiplication of tall-and-skinny matrix by a small square matrix, $N \times 2400 \times 2400$. Again, fast algorithms that have this shape (*e.g.*, $\langle 4, 3, 3 \rangle$) tend to have the highest performance.

half the digits (a better speedup with the same or better numerical accuracy can be obtained by switching to single precision). For the problem sizes $N \times 1600 \times N$ and $N \times 2400 \times 2400$ (Figure 6), we evaluate the algorithms that are comparable to, or outperform, Strassen's algorithm. The results are summarized as follows:

1. All of the fast algorithms outperform MKL for large enough problem sizes. These algorithms are implemented with our code generator and use only the high-level optimizations described in Section 3.1. Since the fast algorithms perform less computation and communication, we expect this to happen.
2. For square matrices, Strassen's algorithm often performs the best. This is mostly due to its relatively small number of matrix additions in comparison to other fast algorithms. On large problem sizes, Strassen's algorithm provides around a 20% speedup over MKL's `dgemm`. The right plot of Figure 5 shows that some algorithms are competitive with Strassen's algorithm on large problems. These algorithms have large speedups per recursive step (see Table 1). While Strassen's algorithm can take more recursive steps, memory constraints and the cost of additions with additional recursive steps cause Strassen's algorithm to be on par with these other algorithms.
3. Although Strassen's algorithm has the highest performance for square matrices, other fast algorithms have higher performance for $N \times 1600 \times N$ and $N \times 2400 \times 2400$ problem sizes (Figure 6). The reason is that the fixed dimension constrains the number of recursive steps. With multiple recursive steps, the matrix sub-blocks become small enough so that `dgemm` does not achieve good performance on the sub-problem. Thus, algorithms that get a better speedup per recursive step have higher performance for these problem sizes.
4. For rectangular matrices, algorithms that "match the shape" of the problem tend to perform the best. For example, $\langle 4, 2, 4 \rangle$ and $\langle 3, 2, 3 \rangle$ both have the "outer product" shape of the $N \times 1600 \times N$ problem sizes and have the highest performance. Similarly, $\langle 4, 2, 3 \rangle$ and $\langle 4, 3, 3 \rangle$ have the highest performance of the exact algorithms for $N \times 2400 \times 2400$ problem sizes. The $\langle 4, 2, 4 \rangle$ and $\langle 4, 3, 3 \rangle$ algorithms provide around a 5% performance improvement over Strassen's algorithm and a 10% performance improvement over MKL on $N \times 1600 \times N$ and $N \times 2400 \times 2400$, respectively. The reason follows from the performance explanation from Result 3. Only one or two steps of recursion improve performance. Thus, algorithms that match the problem shape and have high speedups per step perform the best.
5. Bini's $\langle 3, 2, 2 \rangle$ APA algorithm typically has the highest performance on rectangular problem sizes. However, we remind the reader that the approximation used by this algorithm results in severe numerical errors.

5.2 Parallel Performance

Figure 7 shows the parallel performance for multiplying square matrices and Figure 8 shows the parallel performance for $N \times 2800 \times N$ and $N \times 3000 \times 3000$ problem sizes. We include performance on both 6 and 24 cores in order to illustrate the bandwidth issues discussed in Section 4.5. We observe the following patterns in the parallel performance data:

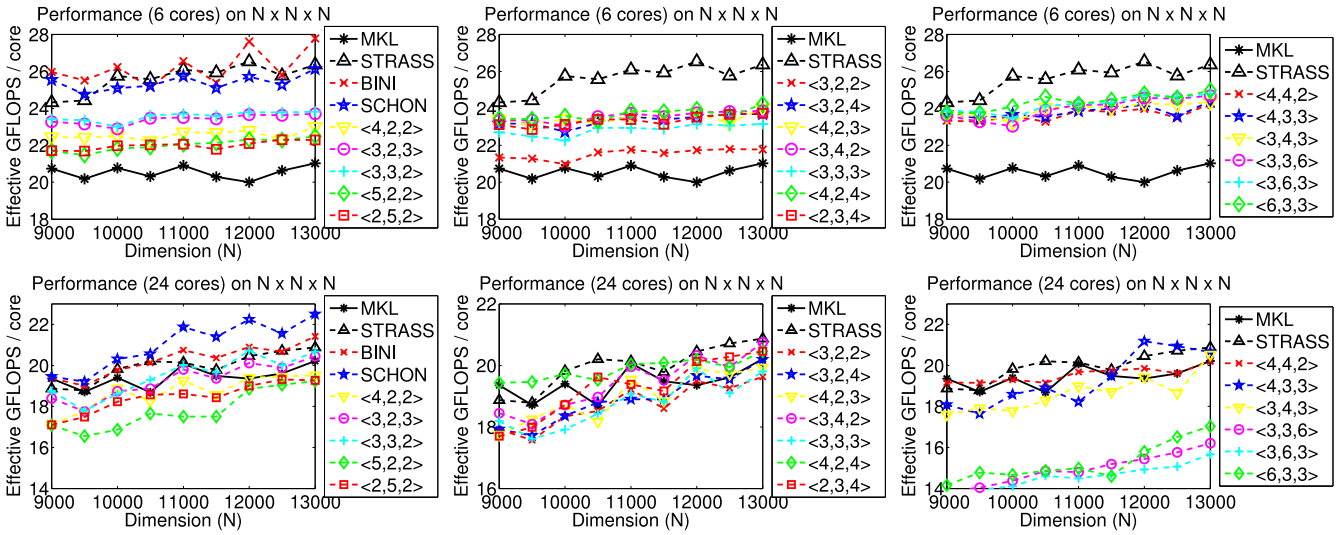

Figure 7. Effective parallel performance (Equation (3)) of fast algorithms on square problems using only 6 cores (top row) and all 24 cores (bottom row). With 6 cores, bandwidth is not a bottleneck and we see similar trends to the sequential algorithms. With 24 cores, speedups over MKL are less dramatic, but Strassen's (bottom left), $\langle 3,3,2\rangle$ (bottom left), and $\langle 4,3,3\rangle$ (bottom right) all outperform MKL and have similar performance. Bini and Schöhage have high performance, but they are APA algorithms and suffer from severe numerical problems.

1. With 6 cores, bandwidth scaling is not a problem, and we find many of the same trends as in the sequential case. All fast algorithms outperform MKL. Apart from the APA algorithms, Strassen's algorithm is typically fastest for square matrices. The $\langle 3,2,3\rangle$ fast algorithm has the highest performance for the $N \times 2800 \times N$ problem sizes, while $\langle 4,3,3\rangle$ and $\langle 4,2,3\rangle$ have the highest performance for the $N \times 3000 \times 3000$. These algorithms match the shape of the problem.
2. With 24 cores, MKL's `dgemm` is typically the highest performing algorithm for rectangular problem sizes (bottom row of Figure 8). In these problems, the ratio of time spent in additions to time spent in multiplications is too large, and bandwidth limitations prevent the fast algorithms from outperforming MKL.
3. With 24 cores and square problem sizes (bottom row of Figure 7), several algorithms outperform MKL. Strassen's algorithm provides a modest speedup over MKL (around 5%) and is one of highest performing exact algorithms. The $\langle 4,3,3\rangle$ and $\langle 4,2,4\rangle$ fast algorithms outperform MKL and are competitive with Strassen's algorithm. The square problem sizes spend a large fraction of time in matrix multiplication, so the bandwidth costs for the matrix additions have less impact on performance.
4. Again, the APA algorithms (Bini's and Schönhage's) have high performance on rectangular problem sizes. It is still an open question if there exists a fast algorithm with the same complexity as Schönhage's algorithm. Our results show that a significant performance gain is possible with such an algorithm.

We also benchmarked the asymptotically fastest implementation of square matrix multiplication. The algorithm consists of composing $\langle 3,3,6\rangle$, $\langle 3,6,3\rangle$, $\langle 6,3,3\rangle$ base cases. At the first recursive level, we use $\langle 3,3,6\rangle$; at the second level $\langle 3,6,3\rangle$; and at the third, $\langle 6,3,3\rangle$. The composed fast algorithm is for $\langle 3\cdot 3\cdot 6, 3\cdot 6\cdot 3, 6\cdot 3\cdot 3\rangle = \langle 54,54,54\rangle$. Each step of the composed algorithm computes $40^3 = 64000$ matrix multiplications. The asymptotic complexity of this algorithm is $\Theta(N^{\omega_0})$, with $\omega_0 = 3\log_{54}(40) \approx 2.775$.

Although this algorithm is *asymptotically* the fastest, it does not perform well for the problem sizes considered in our experiments. For example, with 6 cores and BFS parallelism, the algorithm achieved only 8.4 effective GFLOPS/core multiplying square with dimension $N = 13000$. This is far below MKL's performance (Figure 7). We conclude that while the algorithm may be of theoretical interest, it does not perform well on the modest problem sizes of interest on shared memory machines.

6. Discussion

Our code generation framework lets us benchmark a large number of existing and new fast algorithms and test a variety of implementation details, such as how to handle matrix additions and how to implement the parallelism. However, we performed only high-level optimizations; we believe more detailed tuning of fast algorithms can provide performance gains. Based on the performance results we obtain in this work, we can draw several conclusions in bridging the gap between the theory and practice of fast algorithms.

First, in the case of multiplying square matrices, Strassen's algorithm consistently dominates the performance of exact algorithms (in sequential and parallel). Even though Smirnov's exact algorithm and Schönhage's APA algorithm are asymptotically faster in theory, they never outperform Strassen's for reasonable matrix dimensions in practice (sequential or parallel). This sheds some doubt on the prospect of finding a fast algorithm that will outperform Strassen's on square matrices; it will likely need to have a small base case and still offer a significant reduction in multiplications.

On the other hand, another conclusion from our performance results is that for multiplying rectangular matrices (which occurs more frequently than square in practice), there is a rich space for improvements. In particular, fast algorithms with base cases that match the shape of the matrices tend to have the highest performance. There are many promising algorithms, and we suspect that algorithm-specific optimizations will prove fruitful.

Third, in the search for new fast algorithms, our results confirm the importance of the (secondary) metric of sparsity of the $[\![\mathbf{U},\mathbf{V},\mathbf{W}]\!]$ factor matrices. Although the arithmetic cost associated with the sparsity is negligible in practice, the communication cost associated with each nonzero can be performance limiting. We note that the communication costs of the streaming additions algorithm

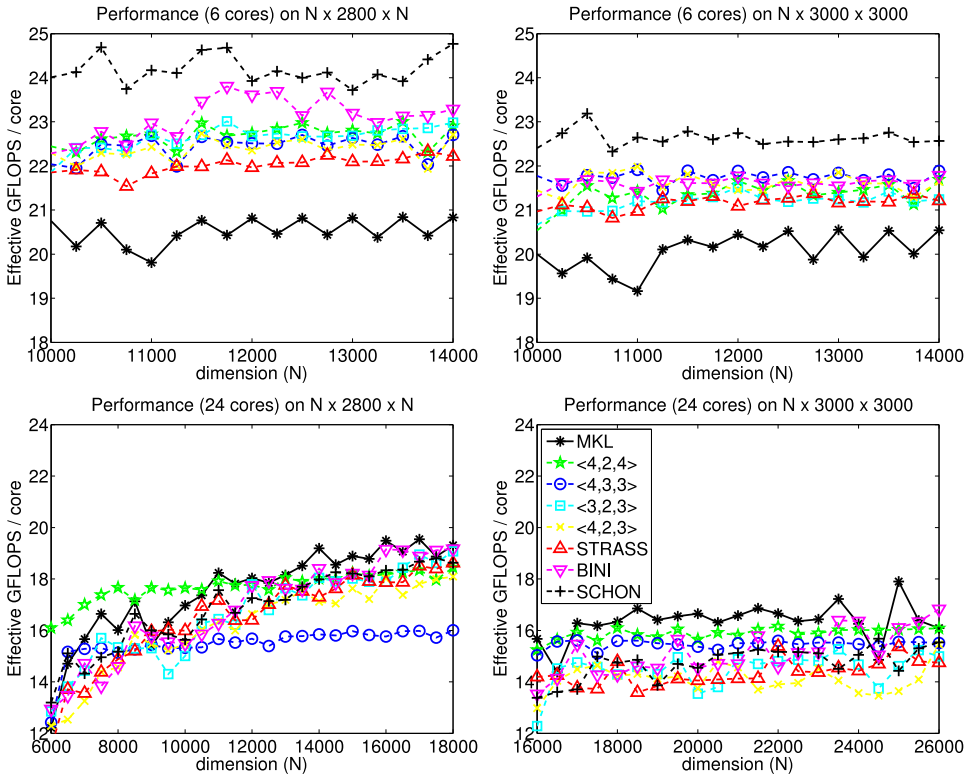

Figure 8. Effective parallel performance (Equation (3)) of fast algorithms on rectangular problems using only 6 cores (top row) and all 24 cores (bottom row). Problem sizes are an "outer product" shape, $N \times 2800 \times N$ and multiplication of tall-and-skinny matrix by a small square matrix, $N \times 3000 \times 3000$. With six cores, all fast algorithms outperform MKL, and new fast algorithms achieve about a 5% performance gain over Strassen's algorithm. With 24 cores, bandwidth is a bottleneck and MKL outperforms fast algorithms.

is independent of the sparsity, but the highest-performing additions algorithm in practice is the write-once algorithm, which is sensitive to the number of nonzeros.

Fourth, we have identified a parallel scaling impediment for fast algorithms on shared-memory architectures. Because the memory bandwidth often does not scale with the number of cores, and because the additions and multiplications are separate computations in our framework, the overhead of the additions compared to the multiplications worsens in the parallel case. This hardware bottleneck is unavoidable on most shared-memory architectures, though we note that it does not occur in distributed memory where aggregate memory bandwidth scales with the number of nodes.

We would like to extend our framework to the distributed-memory case, in part because of the better prospects for parallel scaling. A larger fraction of the time is spent in communication for the classical algorithm on this architecture, and fast algorithms can reduce the communication cost in addition to the computational cost in this case [3]. Similar code generation techniques will be helpful in exploring performance in this case.

As matrix multiplication is the main computational kernel in linear algebra libraries, we also want to incorporate these fast algorithms into frameworks like BLIS [35] and PLASMA [26] to see how they affect a broader class of numerical algorithms.

Finally, we have not explored the numerical stability of the exact algorithms in order to compare their results. While theoretical bounds can be derived from each algorithm's $[\![\mathbf{U}, \mathbf{V}, \mathbf{W}]\!]$ representation, it is an open question which algorithmic properties are most influential in practice; our framework will allow for rapid empirical testing. As numerical stability is an obstacle to widespread use of fast algorithms, extensive testing can help alleviate (or confirm) common concerns.

Acknowledgments

This research was supported in part by an appointment to the Sandia National Laboratories Truman Fellowship in National Security Science and Engineering, sponsored by Sandia Corporation (a wholly owned subsidiary of Lockheed Martin Corporation) as Operator of Sandia National Laboratories under its U.S. Department of Energy Contract No. DE-AC04-94AL85000. Austin R. Benson is also supported by an Office of Technology Licensing Stanford Graduate Fellowship.

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] AMD. AMD core math library user guide, 2014. Version 6.0.

[2] D. H. Bailey. Extra high speed matrix multiplication on the Cray-2. *SIAM Journal on Scientific and Statistical Computing*, 9(3):603–607, 1988.

[3] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. Communication-optimal parallel algorithm for Strassen's matrix multiplication. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 193–204. ACM, 2012.

[4] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Graph expansion and communication costs of fast matrix multiplication. *Journal of the*

ACM, 59(6):32, 2012.

[5] D. Bini, M. Capovani, F. Romani, and G. Lotti. $O(n^{2.7799})$ complexity for $n \times n$ approximate matrix multiplication. *Information Processing Letters*, 8(5):234 – 235, 1979.

[6] D. Bini, G. Lotti, and F. Romani. Approximate solutions for the bilinear form computational problem. *SIAM Journal on Computing*, 9 (4):692–697, 1980.

[7] R. P. Brent. Algorithms for matrix multiplication. Technical report, Stanford University, Stanford, CA, USA, 1970.

[8] Cray. Cray application developer's environment user's guide, 2012. Release 3.1.

[9] P. D'Alberto, M. Bodrato, and A. Nicolau. Exploiting parallelism in matrix-computation kernels for symmetric multiprocessor systems. *ACM Transactions on Mathematical Software*, 38(1):2, 2011.

[10] H. F. de Groote. On varieties of optimal algorithms for the computation of bilinear mappings I. the isotropy group of a bilinear mapping. *Theoretical Computer Science*, 7(1):1 – 24, 1978.

[11] C. C. Douglas, M. Heroux, G. Slishman, and R. M. Smith. GEMMW: a portable level 3 BLAS Winograd variant of Strassen's matrix-matrix multiply algorithm. *Journal of Computational Physics*, 110(1):1–10, 1994.

[12] F. L. Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, pages 296–303, 2014.

[13] B. Grayson and R. Van De Geijn. A high performance parallel Strassen implementation. *Parallel Processing Letters*, 6(01):3–12, 1996.

[14] N. J. Higham. *Accuracy and stability of numerical algorithms*. SIAM, 2002.

[15] J. Hopcroft and J. Musinski. Duality applied to the complexity of matrix multiplication and other bilinear forms. *SIAM Journal on Computing*, 2(3):159–173, 1973.

[16] J. E. Hopcroft and L. R. Kerr. On minimizing the number of multiplications necessary for matrix multiplication. *SIAM Journal on Applied Mathematics*, 20(1):30–36, 1971.

[17] S. Huss-Lederman, E. M. Jacobson, J. R. Johnson, A. Tsao, and T. Turnbull. Strassen's algorithm for matrix multiplication: Modeling, analysis, and implementation. In *In Proceedings of Supercomputing '96*, pages 9–6, 1996.

[18] IBM. Engineering and scientific software library guide and reference, 2014. Version 5, Release 3.

[19] Intel. Math kernel library reference manual, 2014. Version 11.2.

[20] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026, 2004.

[21] R. W. Johnson and A. M. McLoughlin. Noncommutative bilinear algorithms for 3 x 3 matrix multiplication. *SIAM Journal on Computing*, 15(2):595–603, 1986.

[22] I. Kaporin. The aggregation and cancellation techniques as a practical tool for faster matrix multiplication. *Theoretical Computer Science*, 315(2):469–510, 2004.

[23] D. E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1981. ISBN 0-201-03822-6.

[24] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009.

[25] B. Kumar, C.-H. Huang, P. Sadayappan, and R. W. Johnson. A tensor product formulation of Strassen's matrix multiplication algorithm with memory reduction. *Scientific Programming*, 4(4):275–289, 1995.

[26] J. Kurzak, P. Luszczek, A. YarKhan, M. Faverge, J. Langou, H. Bouwmeester, J. Dongarra, J. J. Dongarra, M. Faverge, T. Herault, et al. Multithreading in the PLASMA library. *Multicore Computing: Algorithms, Architectures, and Applications*, page 119, 2013.

[27] B. Lipshitz, G. Ballard, J. Demmel, and O. Schwartz. Communication-avoiding parallel Strassen: Implementation and performance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 101, 2012.

[28] J. D. McCalpin. A survey of memory bandwidth and machine balance in current high performance computers. *IEEE TCCA Newsletter*, pages 19–25, 1995.

[29] V. Y. Pan. Strassen's algorithm is not optimal: Trilinear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 166–176, 1978.

[30] A. Schönhage. Partial and total matrix multiplication. *SIAM Journal on Computing*, 10(3):434–455, 1981.

[31] A. Smirnov. The bilinear complexity and practical algorithms for matrix multiplication. *Computational Mathematics and Mathematical Physics*, 53(12):1781–1795, 2013.

[32] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[33] M. Thottethodi, S. Chatterjee, and A. R. Lebeck. Tuning Strassen's matrix multiplication for memory efficiency. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, pages 1–14, 1998.

[34] R. A. van de Geijn and J. Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency-Practice and Experience*, 9(4): 255–274, 1997.

[35] F. G. Van Zee and R. A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software*, 2014. To appear.

[36] V. V. Williams. Multiplying matrices faster than Coppersmith-Winograd. In *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing*, pages 887–898. ACM, 2012.

[37] S. Winograd. On multiplication of 2×2 matrices. *Linear Algebra and its Applications*, 4(4):381–388, 1971.