2011 IEEE 17th International Conference on Parallel and Distributed Systems

Strassen's Matrix Multiplication on GPUs

Junjie Li Sanjay Ranka Sartaj Sahni
Department of Computer and Information Science and Engineering
University of Florida
Gainesville, FL, USA
{*jl3, ranka, sahni*}*@cise.ufl.edu*

***Abstract*—We provide efficient single-precision and integer GPU implementations of Strassen's algorithm as well as of Winograd's variant. On an NVIDIA C1060 GPU, a speedup of 32% (35%) is obtained for Strassen's 4-level implementation and 33% (36%) for Winograd's variant relative to the $sgemm$ (integer version of $sgemm$) code in CUBLAS 3.0 when multiplying 16384×16384 matrices. The maximum numerical error for the single-precision implementations is about 2 orders of magnitude higher than those for $sgemm$ when $n = 16384$ and is zero for the integer implementations.**

***Keywords*-GPU; CUDA; matrix multiplication; Strassen's algorithm; Winograd's variant; accuracy;**

I. Introduction

Matrix multiplication is an integral component of the CUDA (Compute Unified Driver Architecture) BLAS library [1] and much effort has been expended in obtaining an efficient CUDA implementation. The current implementation in the CUDA BLAS library is based on an algorithm due to Volkov and Demmel [18]. A further 3% reduction (on the NVIDIA Tesla C1060) in run time is achieved by the algorithm $GPU8$ [12]. Li, Ranka, and Sahni [12] provide a step-by-step development of efficient GPU matrix multiplication algorithms beginning with the classical three-loop $O(n^3)$ single-core algorithm to multiply two $n \times n$ matrices. Although significant effort has been expended to obtain efficient GPU algorithms for matrix multiplication based on the classical $O(n^3)$ single-core algorithm, there appears to be no work toward obtaining efficient GPU implementations of any of the single-core matrix algorithms whose complexity is less than $O(n^3)$. Of these latter lower complexity algorithms, Strassen's original $O(n^{2.81})$ algorithm [17] and Winograd's variant [19] of this algorithm, whose asymptotic complexity is also $O(n^{2.81})$ are considered the most practical. Hence, we focus on these two algorithms in this paper. We note that the asymptotically fastest matrix multiplication algorithm at this time has a complexity $O(n^{2.38})$ [6] and it is believed that "*an optimal algorithm for matrix multiplication will run in essentially* $O(n^2)$ *time*" [14].

Both Strassen's algorithm and Winograd's variant compute the product C of two matrices A and B by first decomposing each matrix into 4 roughly equal sized blocks as in Figure 1. Strassen's algorithm [17] computes C by

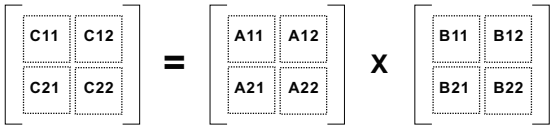

Figure 1. Block decomposition of A, B, and C

performing 7 matrix multiplications and 18 add/subtracts using the following equations:

$$
\begin{array}{ll}
M_1 = (A_{11} + A_{22})(B_{11} + B_{22}) & C_{11} = M_1 + M_4 - M_5 + M_7 \\
M_2 = (A_{21} + A_{22})B_{11} & C_{12} = M_3 + M5 \\
M_3 = A_{11}(B_{12} - B_{22}) & C_{21} = M_2 + M_4 \\
M_4 = A_{22}(B_{21} - B_{11}) & C_{22} = M_1 - M_2 + M_3 + M_6 \\
M_5 = (A_{11} + A_{12})B_{22} & \\
M_6 = (A_{21} - A_{11})(B_{11} + B_{12}) & \\
M_7 = (A_{12} - A_{22})(B_{21} + B_{22}) &
\end{array}
$$

When this block decomposition is applied recursively until the block dimensions reach (or fall below) a threshold value (say τ) the arithmetic complexity of Strassen's algorithm becomes $O(n^{2.81})$.

Winograd's variant of Strassen's method uses the following equations to compute C with 7 matrix multiplies and 15 add/subtracts [7]:

$$
\begin{array}{lll}
S_1 = A_{21} + A_{22} & M_1 = S_2 * S_6 & V_1 = M_1 + M_2 \\
S_2 = S_1 - A_{11} & M_2 = A_{11} * B_{11} & V_2 = V_1 + M_4 \\
S_3 = A_{11} - A_{21} & M_3 = A_{12} * B_{21} & C_{11} = M_2 + M_3 \\
S_4 = A_{12} - S_2 & M_4 = S_3 * S_7 & C_{12} = V_1 + M_5 + M_6 \\
S_5 = B_{12} - B_{11} & M_5 = S_1 * S_5 & C_{21} = V_2 - M_7 \\
S_6 = B_{22} - S_5 & M_6 = S_4 * B_{22} & C_{22} = V_2 + M_5 \\
S_7 = B_{22} - B_{12} & M_7 = A_{22} * S_8 & \\
S_8 = S_6 - B_{21} & &
\end{array}
$$

Although the recursive application of Winograd's variant also results in an asymptotic complexity of $O(n^{2.81})$, the reduction in number of matrix adds/subtracts from 18 to 15 manifests itself as a slightly smaller measured run time in practice.

Bailey, Lee, and Simon [4] describe an implementation of Strassen's algorithm for the CRAY-2 and CRAY Y-MP. This implementation uses three temporary (scratch) matrices at each level of the recursion. The total space required by these temporary matrices is at most n^2. However, the computation can be done using 2 temporaries and the space required by temporary matrices to at most $2n^2/3$. Karunadasa and Ranasinghe [11] decribe an implementation of Strassesn's algorithm on a 2 CPU GPU cluster in which one CPU has 4 cores and the other 2; each node has an NVIDIA GT 8800 GPU. In this implementation the work is shared by

1521-9097/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPADS.2011.130

the 2 CPUs and the 2 GPUs with the GPUs doing most of the submatrix multiplies and the CPUs doing the submatrix additions and subtractions and some of the multiplications. They do not, however, attempt to develop an efficient implementation in which all of the work is done by a GPU. Douglas et al. [7] provide an implementation of Winograd's variant that uses two temporary matrices at each level of the recursion. So, this implementation uses at most $2n^2/3$ memory for temporary matrices. Douglas et al. [7] report on the performance of their implementation on a variety of serial and parallel computers. Huss-Lederman et al. [10], [9] describe two implementations of Winograd's variant. The first uses two temporary matrices at each level of the recursion and is identical to the implementation of Douglas et al. [7]. The second implementation uses 3 temporaries at each level of the recursion. This second implementation, however, is recommended only for the case when we are using the Winograd variant to do a multiply-accumulate (i.e., $C = \alpha AB + \beta C$) and not when we are doing a straight multiply ($C = AB$) as in this paper. So, we do not consider this implementation further in this paper. Boyer et al. [5] show how to implement Winograd's variant using no temporary matrix. They provide two implementations. The first does not increase the number of arithmetic operations but overwrites the input matrices A and B. Since we do not permit overwriting of the input matrices, we do not consider this implementation. Although the second in-place implementation does not overwrite the input matrices, it increases the number of arithmetics by a constant factor. So, we do not consider this implementation either. There appears to be no GPU implementation of Winograd's algorithm.

The remainder of this paper is organized as follows. In Section II, we describe the architecture of the NVIDIA Tesla C1060 GPU. The fastest $O(n^3)$ GPU matrix multiplication algorithm $GPU8$ [12] is described in Section III. Section IV gives the basic GPU kernels used in our GPU adaptations of Strassen's algorithm and Winograd's variant and also analyzes these kernels for their device-memory transactions and volume complexity. A one-level GPU implementation of Strassen's algorithm and Winograd's variant (i.e., an implementation that does not apply Strassen's and Winograd's equations recursively) is given in Section V and the general multilevel recursive implementation is given in Section VI. Experimentation results for single-precision and integer implementations of Strassen's and Winograd's algorithms are presented in Section VII. We conclude in Section VIII.

Throughout this paper, we assume that n is a power of 2. Adaptations to other values of n may be done using methods such as padding and peeling [10], [9]. While our development is done explicitly for single-precision data, the integer version is obtained by simply changing the data type from `float` to `int`.

II. GPU Architecture

NVIDIA's Tesla C1060 GPU is an example of NVIDIA's general purpose parallel computing architecture CUDA (Compute Unified Driver Architecture) [16]. The C1060 comprises 30 streaming multiprocessors (SMs) and each SM comprises 8 scalar processors (SPs), 16KB of on-chip shared memory, and 16,384 32-bit registers. Each SP has its own integer and single-precision floating point units. Each SM has 1 double-precision floating-point unit and 2 single-precision transcendental function (special function, SF) units that are shared by the 8 SPs in the SM. The 240 SPs of a Tesla C1060 share 4GB of off-chip memory referred to as *device* or *global* memory [2]. A C1060 has a peak performance of 933 GFlops of single-precision floating-point operations and 78 GFlops of double-precision operations. The peak of 933GFlops is for the case when Multiply-Add (MADD) instructions are dual issued with special function (SF) instructions. In the absence of SF instructions, the peak is 622GFlops (MADDs only) [3]. The C1060 consumes 188W of power.

A Tesla GPU is packaged as a double-wide PCIe card and using an appropriate motherboard and a sufficiently large power supply, one can install up to 4 GPUs on the same motherboard. In this paper, we focus on single GPU computation.

III. The Matrix Multiplication Algorithm $GPU8$

The matrix multiplication kernel $GPU8$, which is due to Li, Ranka, and Sahni [12], assumes that the matrices A, B, and C are mapped to the device memory arrays a, b, and c using the row-major mapping [15]. The kernel is invoked by the host using $(16, 8)$ thread blocks. A thread block reads a 16×64 sub-matrix of a from device memory to shared memory. Each half warp reads the 64 a values in a row of the 16×64 sub-matrix, which lie in two adjacent 128-byte segments of device memory, using two 128-byte transactions. To accomplish this, each thread reads a 1×4 sub-matrix of a using the data type `float4`. The 16×64 a sub-matrix that is input from device memory may be viewed as a 16×16 matrix in which each element is a 1×4 vector. The transpose of this 16×16 matrix of vectors is stored in the shared-memory array $as[16][65]$ with each 1×4 vector using four adjacent elements of a row of as. This mapping ensures that the 16 elements in each column of the 16×64 sub-matrix of a that is input from device memory are stored in different banks of shared memory. So, the writes to shared memory done by a half warp of $GPU8$ are conflict free. Further, by storing the transpose of a 16×16 matrix of 1×4 vectors rather than the transpose of a 16×64 matrix of scalars, $GPU8$ is able to do the writes to shared memory using `float4`s rather than `float`s as would otherwise be the case. This reduces the time to write to shared memory.

The total number of transactions for $GPU8$ is $17n^3/4096 + n^2/16$ and the volume is $9n^3/32 + 4n^2$ [12].

```
__global__ void add (float *d_A, float *d_B, float *d_C,
                     int widthA, int widthB, int widthC)
{
    int startA = blockIdx.x*64 + threadIdx.x*2 +
                 (blockIdx.y*8 + threadIdx.y)*widthA;
    int startB = blockIdx.x*64 + threadIdx.x*2 +
                 (blockIdx.y*8 + threadIdx.y)*widthB;
    int startC = blockIdx.x*64 + threadIdx.x*2 +
                 (blockIdx.y*8 + threadIdx.y)*widthC;

    float2 tempA = *(float2 *)(d_A+startA);
    float2 tempB = *(float2 *)(d_B+startB);

    tempA.x += tempB.x;
    tempA.y += tempB.y;

    *(float2 *)(d_C+startC) = tempA;
}
```

Figure 2. Kernel to add two matrices

By comparison, the number of transactions and volume for the $sgemm$ code in CUBLAS 3.0 [1] are $5n^3/1024+n^2/16$ and $5n^3/16 + 4n^2$, respectively.

IV. Basic GPU Kernels

We use several basic GPU kernels to arrive at our efficient GPU adaptation of Strassen's algorithm and Winograd's variant. These kernels are described below.

1) $add(X, Y, Z)$ computes $Z = X + Y$ using the kernel code of Figure 2. Each thread fetches two adjacent values of X and two adjacent values of B from device memory using the data type `float2`. Since the 16 pairs of X (Y) fetched from device memory lie in the same 128-byte segment, the fetches of a half warp are coalesced into a single 128-byte memory transaction. The fetched pairs of X and Y are added and the sums written to device memory. This write also requires one memory transaction per half warp. So, two $m \times m$ matrices are added using a total of $3m^2/32$ 128-byte transactions that result in a total volume of $12m^2$ bytes.
2) $sub(X, Y, Z)$ computes $Z = X - Y$ using a kernel code similar to that of Figure 2.
3) $mul(X, Y, Z)$ computes $Z = X*Y$ using $GPU8$. Let T and V, respectively, denote the number of memory transactions and volume for this code when multiplying two $m \times m$ matrices ($T = 17m^3/4096 + m^2/16$ and $V = 9m^3/32 + 4m^2$).
4) $mulIncInc(W, X, Y, Z)$ computes $(Y+, Z+) = W * X$ (i.e., Y and Z are both incremented by $W * X$). This is done by modifying the matrix multiply kernel so that it does not write out the elements of $W * X$ as each is computed. Instead, after an element of $W * X$ has been computed, the corresponding elements of Y and Z are read from device memory, incremented by the computed element of $W * X$, and the incremented values written back to device memory. Note that each element of $W * X$ is computed exactly once. The modified kernel makes $T - m^2/16$ transactions to multiply W and X as it does not write out $W * X$. Additional transactions are made to fetch Y and Z and write the incremented values. A half warp reads/writes Y and Z using coalesced 64-byte transactions. The total number of these transactions is $m^2/4$ ($m^2/16$ transactions are made to read or write each of Y and Z). So, the total number of transactions is $T+3m^2/16$ and the volume is $V + 12m^2$.
5) $mulIncDec(W, X, Y, Z)$ computes $(Y+, Z-) = W * X$. This is similar to $mulIncInc$ and has the same transaction count and volume.
6) $mulStoreDec(W, X, Y, Z)$ computes $(Y, Z-) = W * X$. Again, this is one by modifying the matrix multiply kernel so that after it stores a computed element of $W * X$ to the appropriate device memory location for Y, it reads the corresponding element of Z from device memory, decrements this element of Z by the value of he just computed element of Y and stores the decremented element of Z in device memory. In addition to the transactions (T) made to compute and store $W * X$, the modified kernel fetches and writes Z using $m^2/8$ 64-byte transactions. So, the modified kernel makes a total of $T + m^2/8$ transactions and generates a volume of $V + 8m^2$.
7) $mulStoreInc(W, X, Y, Z)$ computes $(Y, Z+) = W * X$ using a suitable modification of the matrix multiply kernel. This kernel is similar to that for $mulStoreDec$ and has the same number of transactions and volume.
8) $mulAdd(W, X, Y, Z)$ computes $Z = W*X+Y$. This kernel, in addition to doing all the work done by the matrix multiply kernel, needs to fetch Y from device memory. This fetching is done using $m^2/16$ 64-byte transactions. So, the total number of transactions is $T + m^2/16$ and the volume is $V + 4m^2$.
9) $mulIncIncInc(U, V, W, X, Y, Z)$ computes $W = U * V; Y$ += $W; Z$ += $Y; Y$ += X (in this order). The modification to the matrix multiply kernel requires that when an element of $U*V$ is computed, it is written to device memory as an element of W; the corresponding element of Y is fetched from device memory and incremented (but not written back to device memory); next the corresponding element of Z is fetched from device memory, incremented by the just computed Y value and written to device memory; finally this element of Y is incremented again by fetching the corresponding element of X from device memory and the incremented value written to device memory. $mulIncIncInc$ makes $m^2/16$ transactions to fetch each of X, Y, and Z and to write each of Y and Z (in addition to those made by the matrix multiply kernel). So, an extra $5m^2/16$ 64-byte transactions are made. The total number of transactions is $T+5m^2/16$ and the volume is $V + 20m^2$.
10) $mulSubInc(V, W, X, Y, Z)$ computes $Y = X - V * W; Z$ += X using a modification of the matrix

Kernel	Transactions	Volume
add	$3m^2/32$	$12m^2$
sub	$3m^2/32$	$12m^2$
mul	$T = 17m^3/4096 + m^2/16$	$V = 9m^3/32 + 4m^2$
$mulIncInc$	$T + 3m^2/16$	$V + 12m^2$
$mulIncDec$	$T + 3m^2/16$	$V + 12m^2$
$mulStoreDec$	$T + m^2/8$	$V + 8m^2$
$mulStoreInc$	$T + m^2/8$	$V + 8m^2$
$mulAdd$	$T + m^2/16$	$V + 4m^2$
$mulIncIncInc$	$T + 5m^2/16$	$V + 20m^2$
$mulSubInc$	$T + 3m^2/16$	$V + 12m^2$

Figure 3. Device-memory transaction statistics for $m \times m$ matrices

Step	Computation	GPU Kernel
1	$C_{12} = A_{21} - A_{11}$	$sub(A_{21}, A_{11}, C_{12})$
2	$C_{21} = B_{11} + B_{12}$	$add(B_{11}, B_{12}, C_{21})$
3	$C_{22} = C_{12} * C_{21}$	$mul(C_{12}, C_{21}, C_{22})$
4	$C_{12} = A_{12} - A_{22}$	$sub(A_{12}, A_{22}, C_{12})$
5	$C_{21} = B_{21} + B_{22}$	$add(B_{21}, B_{22}, C_{21})$
6	$C_{11} = C_{12} * C_{21}$	$mul(C_{12}, C_{21}, C_{11})$
7	$C_{12} = A_{11} + A_{22}$	$add(A_{11}, A_{22}, C_{12})$
8	$C_{21} = B_{11} + B_{22}$	$add(B_{11}, B_{22}, C_{21})$
9	$T_1 = C_{12} * C_{21}$	
10	$C_{11} = T_1 + C_{11}$	
11	$C_{22} = T_1 + C_{22}$	$mulIncInc(C_{12}, C_{21}, C_{11}, C_{22})$
12	$T_2 = A_{21} + A_{22}$	$add(A_{21}, A_{22}, T_2)$
13	$C_{21} = T_2 * B_{11}$	
14	$C_{22} = C_{22} - C_{21}$	$mulStoreDec(T_2, B_{11}, C_{21}, C_{22})$
15	$T_1 = B_{21} - B_{11}$	$sub(B_{21}, B_{11}, T_1)$
16	$T_2 = A_{22} * T_1$	
17	$C_{21} = C_{21} + T_2$	
18	$C_{11} = C_{11} + T_2$	$mulIncInc(A_{22}, T_1, C_{21}, C_{11})$
19	$T_1 = B_{12} - B_{22}$	$sub(B_{12}, B_{22}, T_1)$
20	$C_{12} = A_{11} * T_1$	
21	$C_{22} = C_{22} + C_{12}$	$mulStoreInc(A_{11}, T_1, C_{12}, C_{22})$
22	$T_2 = A_{11} + A_{12}$	$add(A_{11}, A_{12}, T_2)$
23	$T_1 = T_2 * B_{22}$	
24	$C_{12} = C_{12} + T_1$	
25	$C_{11} = C_{11} - T_1$	$mulIncDec(T_2, B_{22}, C_{12}, C_{11})$

Figure 4. GPU kernels in Strassen implementation

multiply kernel. The total number of transactions is $T + 3m^2/16$ and the volume is $V + 12m^2$.

V. One-Level Adaptation

A. One-Level $Strassen$

In a one-level implementation of Strassen's algorithm and Winograd's variant, the 7 matrix products M_1 through M_7 are computed by a direct application of $GPU8$ (i.e., Strassen's and Winograd's equations are not applied recursively). Figure 4 gives the sequence of kernel calls in our one-level implementation of Strassen's method. We refer to the resulting program as one-level $Strassen$. The one-level GPU implementation of Strassen's method invokes the add and sub kernels 10 times, the mul and $mulIncInc$ kernels twice each, and the $mulStoreDec$, $mulStoreInc$, and $mulIncDec$ kernels once each. Using the transaction and volume data for each kernel (Figure 3), we determine the total transaction count to be $7T + 7m^2/4$ and the total volume to be $7V + 172m^2$, where $T = 17m^3/4096 + m^2/16$ and $V = 9m^3/32 + 4m^2$. When multiplying $n \times n$ matrices, the kernels are invoked with $m = n/2$. So, the total number of transactions is $119n^3/32768 + 35n^2/64$ and the volume is $63n^3/256 + 50n^2$.

Step	Computation	GPU Kernel
1	$T_1 = A_{11} - A_{21}$	$sub(A_{11}, A_{21}, T_1)$
2	$T_2 = B_{22} - B_{12}$	$sub(B_{22}, B_{12}, T_2)$
3	$C_{21} = T_1 * T_2$	$mul(T_1, T_2, C_{21})$
4	$T_1 = A_{21} + A_{22}$	$add(A_{21}, A_{22}, T_1)$
5	$T_2 = B_{12} - B_{11}$	$sub(B_{12}, B_{11}, T_2)$
6	$C_{22} = T_1 * T_2$	$mul(T_1, T_2, C_{22})$
7	$T_1 = T_1 - A_{11}$	$sub(T_1, A_{11}, T_1)$
8	$T_2 = B_{22} - T_2$	$sub(B_{22}, T_2, T_2)$
9	$C_{11} = T_1 * T_2$	$mul(T_1, T_2, C_{11})$
10	$T_1 = A_{12} - T_1$	$sub(A_{12}, T_1, T_1)$
11	$C_{12} = T_1 * B_{22}$	
12	$C_{12} = C_{22} + C_{12}$	$mulAdd(T_1, B_{22}, C_{22}, C_{12})$
13	$T_1 = A_{11} * B_{11}$	
14	$C_{11} = C_{11} + T_1$	
15	$C_{12} = C_{11} + C_{12}$	
16	$C_{11} = C_{11} + C_{21}$	$mulIncIncInc(A_{11}, B_{11}, , T_1, C_{21}, C_{11}, C_{12})$
17	$T_2 = T_2 - B_{21}$	$sub(T_2, B_{21}, T_2)$
18	$C_{21} = A_{22} * T_2$	
19	$C_{21} = C_{11} - C_{21}$	
20	$C_{22} = C_{11} + C_{22}$	$mulSubInc(A_{22}, T_2, C_{11}, C_{21}, C_{22})$
21	$C_{11} = A_{12} * B_{21}$	
22	$C_{11} = T_1 + C_{11}$	$mulAdd(A_{12}, B_{21}, T_1, C_{11})$

Figure 5. GPU kernels in Douglas et al.'s [7] implementation of Winograd variant

Method	Arithmetics	Transactions	Volume
$GPU8$	$2n^3 - n^2$	$17n^3/4096 + n^2/16$	$9n^3/32 + 4n^2$
$Strassen$	$7n^3/4 + 11n^2/4$	$119n^3/32768 + 35n^2/64$	$63n^3/256 + 50n^2$
$Winograd$	$7n^3/4 + 2n^2$	$119n^3/32768 + 29n^2/64$	$63n^3/256 + 41n^2$

Figure 6. Transactions and volume for one-level multiplication of $n \times n$ matrices

B. One-Level $Winograd$

Our one-level GPU implementation of Winograd's variant is given in Figure 5. This is based on the 22-step implementation of Douglas et al.'s [7]. We refer to this implementation as one-level $Winograd$. This implementation invokes the add and sub kernels 8 times, the mul kernel 3 times, the $mulAdd$ kernel twice, and the $mulIncIncInc$ and $mulSubInc$ kernels once each. When the kernels are invoked using $m \times m$ matrices, a total of $7T + 11m^2/8$ transactions are made and the total volume is $7V + 136m^2$. In a one-level implementation, $m = n/2$ and the total number of transactions becomes $119n^3/32768 + 29n^2/64$ and the volume is $63n^3/256 + 41n^2$. Figure 6 summarizes the number of arithmetic operations and transactions done by $GPU8$, one-level $Strassen$, and one-level $Winograd$ as well as the volume of data transfer done by each. Figure 7 gives the percent reduction in these quantities relative to $GPU8$. Based on this analysis, we expect the one-level methods to be about 12% faster than $GPU8$.

VI. Multilevel Recursive GPU Adaptation

A. Multilevel $Strassen$

Figure 8 gives the recursive code for our implementation of Strassen's method for n a power of 2. Adaptations to other

n	Method	Arithmetics	Transactions	Volume
4096	$Strassen$	12.5	9.6	8.5
	$Winograd$	12.5	10.2	9.3
8192	$Strassen$	12.5	11.1	10.5
	$Winograd$	12.5	11.3	10.9
16384	$Strassen$	12.5	11.8	11.5
	$Winograd$	12.5	11.9	11.7

Figure 7. Percent reduction relative to $GPU8$ for one-level multiplication

```
Strassen(A, B, C, n) {
    if (n <= τ1) compute C = A * B using GPU8;
    else if (n <= τ2) compute C = A * B using Figure 4;
    else {
            C12 = A21 − A11; C21 = B11 + B12;
            Strassen(C12, C21, C22, n/2); // M6
            C12 = A12 − A22; C21 = B21 + B22;
            Strassen(C12, C21, C11, n/2); // M7
            C12 = A11 + A22; C21 = B11 + B22;
            Strassen(C12, C21, T1, n/2); // M1
            (C11+, C22+) = T1; T2 = A21 + A22;
            Strassen(T2, B11, C21, n/2); // M2
            C22 −= C21; T1 = B21 − B11;
            Strassen(A22, T1, T2, n/2); // M4
            (C11+, C21+) = T2; T1 = B12 − B22;
            Strassen(A11, T1, C12, n/2); // M3
            C22 += C12; T2 = A11 + A12;
            Strassen(T2, B22, T1, n/2); // M5
            (C11−, C12+) = T1;}
}
```

Figure 8. Strassen's GPU matrix multiply

values of n may be done using methods such as padding and peeling [10], [9]. The code uses 2 threshold values τ_1 and τ_2. When $n \leq \tau_1$ the matrices are multiplied using $GPU8$ and when $\tau_1 < n \leq \tau_2$ a one-level Strassen multiplication (defined by the kernel sequence given in Figure 4) is used. When $n > \tau_2$, Strassen's method is applied recursively. In Figure 8, the notation $(X+, Y+) = Z$ refers to a single kernel that increments X and Y by Z. Such a kernel would read X, Y and Z from device memory, increment X and Y, and then write the incremented X and Y to device memory. Hence the kernel would read Z only once while incrementing X and Y using the two steps $X+ = Z$ and $Y+ = Z$ would read Z twice. When, $\tau_2 < n \leq 2 * \tau_2$ the execution of Figure 8 is referred to as a two-level Strassen multiplication. The number of arithmetics, $A(2, n)$, in a two-level multiplication is $7A(1, n/2) + 18ADD(n/2)$, where $A(1, n/2)$ is the number of arithmetics needed in a one-level multiplication of $n/2 \times n/2$ matrices and $ADD(n/2)$ is the number of arithmetics needed to add two $n/2 \times n/2$ matrices. So, $A(2, n) = 7(7(2(n/4)^3 - (n/4)^2) + 18ADD(n/4)) + 18ADD(n/2) = 49n^3/32 + 149n^2/16$. For the number of transactions, $T(2, n)$, we see that two-level multiplication does 12 adds/subtracts/increments/decrements of $n/2 \times n/2$ matrices with each requiring $3(n/2)^2/32 = 3n^2/128$ transactions. The $++$ and $-+$ operations each make $5n^2/128$ transactions (this is a reduction of $n^2/128$ over doing two $+ =$ or one $+ =$ and one $- =$ operation). Each of the 7 multiply operations multiplies two $n/2 \times n/2$ matrices using a one-level multiply that does $119(n/2)^3/32768 + 35(n/2)^2/64$ transactions. So, $T(2, n) = 833n^3/262144 + 347n^2/256$. Using a similar analysis, we know that the volume, $V(2, n)$, is $441n^3/2048 + 277n^2/2$.

When $2^{k-2}\tau_2 < n \leq 2^{k-1}\tau_2$, a k=level execution of $strassen$ occurs. For this k-level execution,

$$\begin{aligned}
A(k,n) &= 7A(k-1,n/2) + 18(n^2/4) \\
&= 7A(k-1,n/2) + 9n^2/2 \\
T(k,n) &= 7T(k-1,n/2) + 12 * 3 * n^2/128 + 3 * 5 * n^2/128 \\
&= 7T(k-1,n/2) + 51n^2/128 \\
V(k,n) &= 7V(k-1,n/2) + 51n^2
\end{aligned}$$

n	Method	Arithmetics	Transactions	Volume
4096	$Strassen$	23.3	15.8	11.7
	$Winograd$	23.3	17.2	13.9
8192	$Strassen$	23.4	19.6	17.6
	$Winograd$	23.4	20.3	18.7
16384	$Strassen$	23.4	21.5	20.5
	$Winograd$	23.4	21.9	21.1

Figure 12. Percent reduction relative to $GPU8$ for two-level multiplication

n	Method	Arithmetics	Transactions	Volume
4096	$Strassen$	32.7	17.0	7.9
	$Winograd$	32.8	20.0	12.6
8192	$Strassen$	32.9	25.0	20.4
	$Winograd$	32.9	26.5	22.8
16384	$Strassen$	32.9	29.0	26.7
	$Winograd$	33.0	29.7	27.9

Figure 13. Percent reduction relative to $GPU8$ for three-level multiplication

where $A(1, n)$, $T(1, n)$, and $V(1, n)$ are for a 1-level execution and are given in Figure 6.

Figures 9 through 11 give the values of A, T, and V for k = 2, 3, and 4. Figures 12 through 14 give the percent reduction in arithmetics, transactions, and volume relative to $GPU8$ for k = 2, 3, and 4. Based on these numbers, we expect the two-level Strassen algorithm to run about 20% faster than $GPU8$ when $n = 16384$ (this would correspond to $\tau_2 = 8192$); we expect the three-level Strassen algorithm run 26% to 33% faster than $GPU8$; and the 4-level version to run 29% to 41% faster (depending on whether arithmetics, transactions, or volume dominates run time).

B. Multilevel $Winograd$

The recursive code for a matrix multiply using Winograd's variant is similar to the multilevel code for Strassen's algorithm and is given in Figure 15. This code does 10 standalone adds/subtracts/increments/decrements of $n/2 \times n/2$ matrices at the outermost level with each reading 2 $n/2 \times n/2$ matrices and writing 1; the assignment to (C_{12}, C_{11}) reads 4 $n/2 \times n/2$ matrices and writes two; and the assignment to (C_{21}, C_{22}) reads 3 $n/2 \times n/2$ matrices and writes 2. The total number of reads/writes at the outermost level is therefore 41. So, for this code, we see that:

$$\begin{aligned}
A(k,n) &= 7A(k-1,n/2) + 15n^2/4 \\
T(k,n) &= 7T(k-1,n/2) + 41 * n^2/128 \\
V(k,n) &= 7V(k-1,n/2) + 41n^2
\end{aligned}$$

for $k > 1$ and $A(1, n)$, $T(1, n)$, and $V(1, n)$ are as in Figure 7. Figures 9 through 11 and Figures 12 through 14, respectively, give the values of A, T, and V and the percent reduction in these quantities relative to $GPU8$ for k = 2, 3, and 4. The expected speedup of $Winograd$ relative to $GPU8$ is slightly higher than for $Strassen$.

n	Method	Arithmetics	Transactions	Volume
4096	$Strassen$	40.9	10.8	-7.2
	$Winograd$	41.0	16.5	2.0
8192	$Strassen$	41.1	26.1	17.0
	$Winograd$	41.2	28.9	21.6
16384	$Strassen$	41.3	33.7	29.2
	$Winograd$	41.3	35.1	31.5

Figure 14. Percent reduction relative to $GPU8$ for four-level multiplication

Method	Arithmetics	Transactions	Volume
$GPU8$	$2n^3 - n^2$	$17n^3/4096 + n^2/16$	$9n^3/32 + 4n^2$
$Strassen$	$49n^3/32 + 149n^2/16$	$833n^3/262144 + 347n^2/256$	$441n^3/2048 + 277n^2/2$
$Winograd$	$49n^3/32 + 29n^2/4$	$833n^3/262144 + 285n^2/256$	$441n^3/2048 + 451n^2/4$

Figure 9. Transactions and volume for two-level multiplication of $n \times n$ matrices

Method	Arithmetics	Transactions	Volume
$GPU8$	$2n^3 - n^2$	$17n^3/4096 + n^2/16$	$9n^3/32 + 4n^2$
$Strassen$	$343n^3/256 + 1331n^2/64$	$5831n^3/2097152 + 2837n^2/1024$	$3087n^3/16384 + 2347n^2/8$
$Winograd$	$343n^3/256 + 263n^2/16$	$5831n^3/2097152 + 2323n^2/1024$	$3087n^3/16384 + 3813n^2/16$

Figure 10. Transactions and volume for three-level multiplication of $n \times n$ matrices

Method	Arithmetics	Transactions	Volume
$GPU8$	$2n^3 - n^2$	$17n^3/4096 + n^2/16$	$9n^3/32 + 4n^2$
$Strassen$	$2401n^3/2048 + 10469n^2/256$	$40817n^3/16777216 + 21491n^2/4096$	$21609n^3/131072 + 18061n^2/32$
$Winograd$	$2401n^3/2048 + 2081n^2/64$	$40817n^3/16777216 + 17573n^2/4096$	$21609n^3/131072 + 29315n^2/64$

Figure 11. Transactions and volume for four-level multiplication of $n \times n$ matrices

```
Winograd(A, B, C, n)
{
    if (n <= τ1 ) compute C = A * B using GPU8;
    else if (n <= τ2) compute C = A * B using Figure 5;
    else {
            T1 = A11 − A21; T2 = B22 − B12;
            Winograd(T1, T2, C21, n/2); //M4
            T1 = A21 + A22; T2 = B12 − B11;
            Winograd(T1, T2, C22, n/2); //M5
            T1 −= A11; T2 = B22 − T2;
            Winograd(T1, T2, C11, n/2); //M1
            T1 = A12 − T1;
            Winograd(T1, B22, C12, n/2); //M6
            C12 += C22;
            Winograd(A11, B11, T1, n/2); //M2
            (C12, C11) = (C11 + C12 + T1, C11 + C21 + T1);
            T2 −= B21;
            Winograd(A22, T2, C21, n/2);
            (C21, C22) = (C11 − C21, C11 + C22);
            Winograd(A12, B21, C11, n/2); //M3
            C11 += T1; }
}
```

Figure 15. Winograd's GPU matrix multiply

VII. Experimental Results

A. Single Precision Matrix Multiply

We programmed several versions of $GPU8$, $Strassen$, $Winograd$, and $sgemm$ using CUDA and measured their run time as well as accuracy on a Tesla C1060 GPU. The different versions of each algorithm varied in their use of texture memory for the input matrices A and B. Because of the limited availability of texture memory, $GPU8$ and $sgemm$ can be readily adapted to use texture memory only when $n < 16384$. For larger values of n, it is necessary to write a blocked version of these algorithms invoking the blocked version using texture memory for the smaller sized A and B blocks to be multiplied. Our experiments with the blocked version of $sgemm$, for example, resulted in a very small reduction in run time from the use of texture memory. For example, when $n = 16384$, our texture memory versions of $sgemm$ yielded best performance using blocks of size 8192×8192 and designating only the blocks of A as texture blocks. The measured reduction in time was about 0.6% relative to the non-blocked $sgemm$ code. Because of this very marginal reduction in run time even for the largest matrix size we used in our experiments, we do not report further on the blocked texture memory versions of $GPU8$ and $sgemm$. $Strassen$ and $Winograd$, on the other hand are well suited for texture memory as they recursively and naturally decompose matrices to smaller size submatrices until the threshold value τ_2 is reached. So long as $\tau_2 \leq 16384$, the pairs of matrices to be multiplied by $GPU8$ using the one-level kernels at the lowest level of recursion may be designated as texture matrices. Again, our experiments showed best performance when only the first matrix in the pair to be multiplied was designated as texture. Hence, in the following, $tStrassen$ and $tWinograd$ refer to versions of $Strassen$ and $Winograd$ in which when $GPU8$ is invoked by the one-level code used when the matrix size drops to τ_2, the first matrix of each pair to be multiplied by $GPU8$ is designated as texture (the syntax of the $GPU8$ code is correspondingly modified to work with its first matrix being texture). For our experiments, we set $\tau_1 = \tau_2/2$.

1) Run Time: Figure 16 gives the time take by our various matrix multiplication codes. $Strassen$ and $Winograd$ had best performance when $\tau_2 = 4096$ while $tStrasen$ and $tWinograd$ performed best when $\tau_2 = 2048$. Figure 16 shows the run time only for these best values of τ_2. Note that the $n = 2048$ times for $Strassen$ and $Winograd$ are the same as for $GPU8$ because $n = \tau_2/2 = \tau_1$. When $n = 16384$, the optimal τ_2 (4096) for $Strassen$ and $Winograd$ results in 3 levels of recursion while the optimal τ_2 (2048) for $tStrassen$ and $tWinograd$ results in 4 levels of recursion. As far as run time goes, when $n = 16384$, $tStrassen$ takes 2.7% less time than does $Strassen$ and the use of texture reduces the run time of $Winograd$ by 3.5%. Further, $Strassen$ is 30.1% faster than $sgemm$ and 27.9% faster than $GPU8$. Our fastest code, $tWinograd$, takes 33.1% less time to multiply two 16384×16384 than $sgemm$ and 31.0% less than taken by $GPU8$. Figures 17 and 18 plot the percent reduction in run time, number of arithmetics, number of transactions, and volume achieved by $Strassen$

Algorithm	τ_2	2048	4096	8192	16384
$sgemm$	-	0.048	0.373	2.966	23.699
$GPU8$	-	0.046	0.361	2.875	22.971
$Strassen$	4096	0.046	0.329	2.344	16.561
$tStrassen$	2048	0.044	0.320	2.276	16.107
$Winograd$	4096	0.046	0.328	2.329	16.425
$tWinograd$	2048	0.044	0.318	2.243	15.846

Figure 16. Run time (seconds) on the Tesla C1060

and $Winograd$ relative to $GPU8$ when $\tau_2 = 4096$. As can be seen, the speedup (reduction in run time) most closely tracks the reduction in volume.

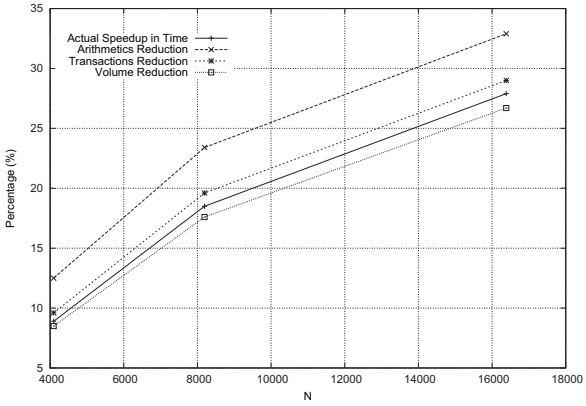

Figure 17. $Strassen$ speedup when $\tau_2 = 4096$

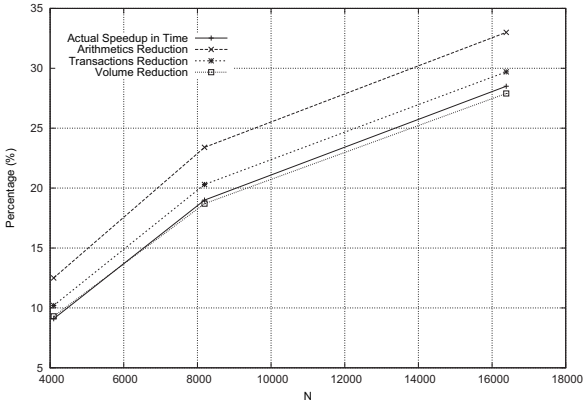

Figure 18. $Winograd$ speedup when $\tau_2 = 4096$

2) Accuracy: Although Strassen's algorithm produces accurate results over exact domain such as integers, it is known to be numerically inaccurate over the reals. [8] We assess the numerical accuracy of Strassen's algorithm and Winograd's variant using the single-precision test matrix used in [13]. Figure 19 gives the maximum absolute difference between an element of the product matrix as computed by each of our algorithms and the exact product and Figure 20 gives the average of the absolute differences. For comparison purposes, we include also the errors in the results obtained using the classical $O(n^3)$ matrix multiplication algorithm on the host CPU. For the reported errors, we used $\tau_2 = 2048$ and 4096.

Algorithm	τ_2	2048	4096	8192	16384
$O(n^3)$ on CPU	-	7.9e-5	1.6e-4	2.3e-4	3.9e-4
$sgemm$	-	8.1e-5	1.6e-4	2.4e-4	3.9e-4
$GPU8$	-	8.1e-5	1.6e-4	2.4e-4	3.9e-4
$Strassen$	2048	2.4e-4	6.7e-4	8.8e-3	8.3e-2
	4096	8.1e-5	3.4e-4	1.5e-3	5.8e-2
$Winograd$	2048	2.5e-4	1.9e-3	2.9e-2	6.3e-1
	4096	8.1e-5	5.0e-4	3.6e-3	1.6e-1

Figure 19. Maximum errors

Algorithm	τ_2	2048	4096	8192	16384
$O(n^3)$ on CPU	-	6.6e-8	5.5e-8	4.7e-8	3.3e-8
$sgemm$	-	6.6e-8	5.6e-8	4.7e-8	3.3e-8
$GPU8$	-	6.6e-8	5.6e-8	4.7e-8	3.3e-8
$Strassen$	2048	2.1e-7	4.6e-7	1.4e-6	1.4e-5
	4096	6.6e-8	1.7e-7	3.9e-7	2.9e-6
$Winograd$	2048	2.8e-7	1.3e-6	1.2e-5	1.7e-4
	4096	6.6e-8	2.8e-7	1.2e-6	3.2e-5

Figure 20. Average errors

Since the use of texture memory does not impact accuracy, Figures 19 and 20 do not explicitly show error measurements for $tStrassen$ and $tWinograd$ (the errors, respectively, are the same as for $Strassen$ and $Winograd$). The maximum and average errors for the classical CPU algorithm, $sgemm$, and $GPU8$ algorithms are almost the same. However, the errors for $Strassen$ and $Winograd$ are substantially larger than those for the classical algorithm, $sgemm$ and $GPU8$ when $n > \tau_1 = \tau_2/2$ (when $n \leq \tau_1 = \tau_2/2$, $Strassen$ and $Winograd$ reduce to $GPU8$). In fact, when $n = 16384$ and $\tau_2 = 2048$, the maximum error for $Strassen$ is 200 times that for the classical algorithm, $sgemm$ and $GPU8$ while the average error is 424 times as much. The corresponding ratios for $Winograd$ are 1615 and 5151. We note also that when $n = 16384$ and $\tau_2 = 2048$, the maximum error for $Winograd$ is about 7.6 times that for $Strassen$ and the average error is about 12 times that for $Strassen$.

3) Performance by Number of Levels: Because of the large numerical errors resulting from $Strassen$ and $Winograd$, we decided to determine how the error varied with the number of levels of recursion. Note that in a 1-level execution, $\tau_1 < n \leq \tau_2$ and in a 2-level execution, $\tau_2 < n \leq 2\tau_2$. A 0-level execution occurs when $n \leq \tau_1 = \tau_2/2$. Figures 21 and 22 give the maximum and average errors as a function of the level of the execution for the case $n = 16384$ and Figure 23 gives the reduction in run time relative to $sgemm$ and $GPU8$. As expected, the errors and speedup (reduction in run time) increase with the number of levels. For example, the 1-level version of $Strassen$ achieves almost a 15% speedup relative to $sgemm$ at the expense of an almost 8 fold increase in the maximum error and an almost 2 fold increase in the average error while the 4-level version achieves a speedup of almost 29% at a cost of an almost 213 fold increase in the maximum error and an almost 212 fold increase in the average error.

B. Integer Matrix Multiply

As indicated earlier, the integer implementations of our algorithms were obtained by simply changing the data

Algorithm	0-level	1-level	2-level	3-level	4-level
$Strassen$	3.9e-4	3.3e-3	3.1e-2	5.8e-2	8.3e-2
$Winograd$	3.9e-4	1.4e-3	9.7e-3	1.6e-1	6.3e-1

Figure 21. Maximum errors when $n = 16384$

Algorithm	0-level	1-level	2-level	3-level	4-level
$Strassen$	6.6e-8	1.1e-7	4.4e-7	2.9e-6	1.4e-5
$Winograd$	6.6e-8	1.9e-7	1.6e-6	3.2e-5	1.7e-4

Figure 22. Average errors when $n = 16384$

type `float` in the single-precision implementation to `int`. Experiments similar to those reported in section VII-A were conducted and for $n = 16,384$, a speedup, relative to the integer version of $sgemm$, of 32% was observed for $Strassen$ and 35% for $tStrassen$ using $\tau_2 = 2048$. The speedup for $Winograd$ and $tWinograd$ was 35% and 36%, respectively, using $\tau_2 = 2048$. As expected, the numerical error was zero.

VIII. Conclusion

We have developed efficient GPU implementations of Strassen's and Winograd's matrix multiplication algorithms. Our experiments indicate that for single-precision (integer) arithmetic a speedup of 32% (35%) is achieved by Strassen's algorithm while Winograd's variant achieves a speedup of 33% (36%) relative to the $sgemm$ (integer version of $sgemm$) code in CUBLAS 3.0 when multiplying 16384×16384 matrices. For single-precision arithmetics, these speedups, however, come at significant cost in the accuracy of the computed result. The maximum numerical error introduced by Strassen's and Winograd's algorithms are about 2 orders of magnitude higher than those for $sgemm$ when $n = 16384$. Whether the loss in accuracy is acceptable or not will depend on the application. There is no loss in accuracy when computing over exact domain such as the integers. We have analyzed the arithmetic, transaction and volume complexity of the various matrix multiplication algorithms considered in this paper (single-precision versions). Our experiments indicate that speedup most closely follows volume.

Acknowledgment

This work was supported, in part, by the National Science Foundation under grants CNS0829916, CNS0905308, and CCF0903430.

References

[1] Cublas 3.0. http://developer.nvidia.com/object/cuda_3_0_downloads.html.

Algorithm	1-level	2-level	3-level	4-level
$Strassen$	14.7/12.0	23.9/21.5	30.1/27.9	28.9/26.6
$tStrassen$	14.9/12.2	24.2/21.8	30.9/28.7	32.0/29.9
$Winograd$	14.7/12.0	24.2/21.8	30.7/28.5	30.1/27.9
$tWinograd$	15.0/12.3	24.5/22.1	31.5/29.3	33.1/31.0

Figure 23. Speedup(%) over $sgemm/GPU8$ when $n = 16384$

[2] Nvidia cuda programming guide. http://developer.nvidia.com/object/cuda_3_0_downloads.html.
[3] Tesla. http://wapedia.mobi/en/NVIDIA_Tesla.
[4] D. H. Bailey, K. Lee, and H. D. Simon. Using strassen's algorithm to accelerate the solution of linear systems. *The Journal of Supercomputing*, 4:357–371, 1991. 10.1007/BF00129836.
[5] B. Boyer, J.-G. Dumas, C. Pernet, and W. Zhou. Memory efficient scheduling of strassen-winograd's matrix multiplication algorithm. In *Proceedings of the 2009 international symposium on Symbolic and algebraic computation*, ISSAC '09, pages 55–62, New York, NY, USA, 2009. ACM.
[6] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251 – 280, 1990. Computational algebraic complexity editorial.
[7] 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*, 110:1–10, Jan. 1994.
[8] N. J. Higham. Exploiting fast matrix multiplication within the level 3 blas. *ACM Trans. Math. Softw.*, 16:352–368, December 1990.
[9] 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.
[10] S. Huss-Lederman, E. M. Jacobson, A. Tsao, T. Turnbull, and J. R. Johnson. Implementation of strassen's algorithm for matrix multiplication. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '96, Washington, DC, USA, 1996. IEEE Computer Society.
[11] N.P. Karunadasa, D.N. Ranasinghe, On the Comparative Performance of Parallel Algorithms on Small GPU/CUDA Clusters. In *High Performance Computing (HiPC), Student Research Symposium*, http://www.hipc.org/hipc2009/studentsymposium.php, 2009.
[12] J. Li, S. Ranka and S. Sahni. *GPU matrix Multiplication, chapter in* Handbook on Multicore Computing. Chapman Hall, 2011. to appear.
[13] I. Kaporin. A practical algorithm for faster matrix multiplication. *Numerical Linear Algebra with Applications*, 6(8):687–700, 1999.
[14] S. Robinson. Toward an optimal algorithm for matrix multiplication. *SIAM News*, 2005.
[15] S. Sahni. *Data Structures, Algorithms, And Applications In C++*. Silicon Press, Summit, NJ, USA, 2004.
[16] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.
[17] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969. 10.1007/BF02165411.
[18] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press.
[19] S. Winograd. On multiplication of 2 x 2 matrices. *Linear Algebra and its Applications*, 4(4):381 – 388, 1971.
[20] Y. Won and S. Sahni. Host-to hypercube sorting. *Comput. Syst. Sci. Eng.*, 4:161–168, July 1989.