

Strassen’s Matrix Multiplication on GPUs *

Junjie Li Sanjay Ranka Sartaj Sahni
{jl3, ranka, sahani}@cise.ufl.edu

Department of Computer and Information Science and Engineering
University of Florida, Gainesville, FL 32611

Abstract

We provide efficient single- and double-precision GPU (Graphics Processing Unit) implementations of Strassen’s matrix multiplication algorithm as well as of Winograd’s variant of this algorithm. The single-precision implementations of these two algorithms are compared analytically using the arithmetic count, device-memory transactions, and device memory to multiprocessor data volume metrics. Our analysis indicates that, for 16384×16384 matrices, our single-precision implementation of Strassen’s algorithm limited to four levels of recursion reduces the number of arithmetics by 41.3%, the number of transactions by 33.7%, and the volume by 29.2% relative to the best known GPU implementation of the classical n^3 matrix multiplication algorithm. The corresponding reductions achieved by Winograd’s variant are 41.3%, 35.1%, and 31.5%. Experimental results obtained using an NVIDIA C1060 GPU indicate a speedup of 32% for Strassen’s 4-level implementation and 33% for Winograd’s variant relative to the *sgemm* code in CUBLAS 3.0 when multiplying 16384×16384 matrices. Our double-precision implementations of Strassen’s and Winograd’s algorithms, respectively, achieve a speedup of 20.2% and 21% relative to *dgemm* when the matrix size n is 8192. The maximum numerical error introduced by Strassen’s and Winograd’s algorithms are about 2 orders of magnitude higher than those for *sgemm* when $n = 16384$ and about 1 order of magnitude higher than for *dgemm* for $n = 8192$. The average numerical error introduced by Strassen’s and Winograd’s algorithms are, respectively, 2 and 3 orders of magnitude higher than those for *sgemm* when $n = 16384$ and about 1 order of magnitude higher than for *dgemm* for $n = 8192$.

Keywords: GPU, CUDA, matrix multiplication, Strassen’s algorithm, Winograd’s variant, accuracy

1 Introduction

Matrix multiplication is an integral component of the CUDA (Compute Unified Driver Architecture) BLAS library [2] 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

*This work was supported, in part, by the National Science Foundation under grants CNS0829916, CNS0905308, CCF0903430, NETS 0963812, and NETS 1115184.

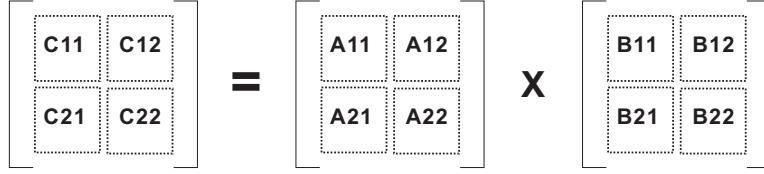


Figure 1: Block decomposition of A , B , and C

$O(n^{2.81})$ algorithm [17] and Winograd’s variant [20] 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 $O(n^{2.38})$ [5] 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 performing 7 matrix multiplications and 18 add/subtracts using the following equations:

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

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 [8]:

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

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.

While there appears to be no GPU implementation of either Strassen’s or Winograd’s variant, both variants have been implemented for other architectures. For example, Bailey, Lee, and Simon [1] 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 T_1 and

Step	Computation	Comment
1	$C_{12} = A_{21} - A_{11}$	
2	$C_{21} = B_{11} + B_{12}$	
3	$C_{22} = C_{12} * C_{21}$	M_6
4	$C_{12} = A_{12} - A_{22}$	
5	$C_{21} = B_{21} + B_{22}$	
6	$C_{11} = C_{12} * C_{21}$	M_7
7	$C_{12} = A_{11} + A_{22}$	
8	$C_{21} = B_{11} + B_{22}$	
9	$T_1 = C_{12} * C_{21}$	M_1
10	$C_{11} = T_1 + C_{11}$	$M_1 + M_7$
11	$C_{22} = T_1 + C_{22}$	$M_1 + M_6$
12	$T_2 = A_{21} + A_{22}$	
13	$C_{21} = T_2 * B_{11}$	M_2
14	$C_{22} = C_{22} - C_{21}$	$M_1 - M_2 + M_6$
15	$T_1 = B_{21} - B_{11}$	
16	$T_2 = A_{22} * T_1$	M_4
17	$C_{21} = C_{21} + T_2$	$M_2 + M_4$
18	$C_{11} = C_{11} + T_2$	$M_1 + M_4 + M_7$
19	$T_1 = B_{12} - B_{22}$	
20	$C_{12} = A_{11} * T_1$	M_3
21	$C_{22} = C_{22} + C_{12}$	$M_1 - M_2 + M_3 + M_6$
22	$T_2 = A_{11} + A_{12}$	
23	$T_1 = T_2 * B_{22}$	M_5
24	$C_{12} = C_{12} + T_1$	$M_3 + M_5$
25	$C_{11} = C_{11} - T_1$	$M_1 + M_4 - M_5 + M_7$

Figure 2: Steps in Strassen implementation

T_2 at each level using the steps given in Figure 2. The implementation of Figure 2 reduces the space required by temporary matrices to at most $2n^2/3$.

Douglas et al. [8] provide an implementation of Winograd’s variant that uses two temporary matrices at each level of the recursion. So, this implementation, which is given in Figure 3, uses at most $2n^2/3$ memory for temporary matrices. Douglas et al. [8] report on the performance of their implementation on a variety of serial and parallel computers.

Huss-Lederman et al. [10, 11] 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. [8] (Figure 3). 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. [3] 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.

Step	Computation	Comment
1	$T_1 = A_{11} - A_{21}$	S_3
2	$T_2 = B_{22} - B_{12}$	S_7
3	$C_{21} = T_1 * T_2$	M_4
4	$T_1 = A_{21} + A_{22}$	S_1
5	$T_2 = B_{12} - B_{11}$	S_5
6	$C_{22} = T_1 * T_2$	M_5
7	$T_1 = T_1 - A_{11}$	S_2
8	$T_2 = B_{22} - T_2$	S_6
9	$C_{11} = T_1 * T_2$	M_1
10	$T_1 = A_{12} - T_1$	S_4
11	$C_{12} = T_1 * B_{22}$	M_6
12	$C_{12} = C_{22} + C_{12}$	$M_5 + M_6$
13	$T_1 = A_{11} * B_{11}$	M_2
14	$C_{11} = C_{11} + T_1$	V_1
15	$C_{12} = C_{11} + C_{12}$	$V_1 + M_5 + M_6$
16	$C_{11} = C_{11} + C_{21}$	V_2
17	$T_2 = T_2 - B_{21}$	S_8
18	$C_{21} = A_{22} * T_2$	M_7
19	$C_{21} = C_{11} - C_{21}$	$V_2 - M_7$
20	$C_{22} = C_{11} + C_{22}$	$V_2 + M_5$
21	$C_{11} = A_{12} * B_{21}$	M_3
22	$C_{11} = T_1 + C_{11}$	$M_2 + M_3$

Figure 3: Steps in Douglas et al.’s [8] implementation of Winograd variant

The remainder of this paper is organized as follows. In Section 2, we describe the architecture of the NVIDIA Tesla C1060 GPU. The CUDA programming model is described in Section 3 and the fastest $O(n^3)$ GPU matrix multiplication algorithm *GPU8* [12] is described in Section 4. Section 5 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 6 and the general multilevel recursive implementation is given in Section 7. Experimentation results for single- and double-precision implementations of Strassen’s and Winograd’s algorithms are presented in Section 8. We conclude in Section 9.

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, 11].

2 GPU Architecture

NVIDIA’s Tesla C1060 GPU, Figure 5, is an example of NVIDIA’s general purpose parallel computing architecture CUDA (Compute Unified Driver Architecture) [16]. Figure 5 is a simplified version of Figure 4 with $N = 30$ and $M = 8$. 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

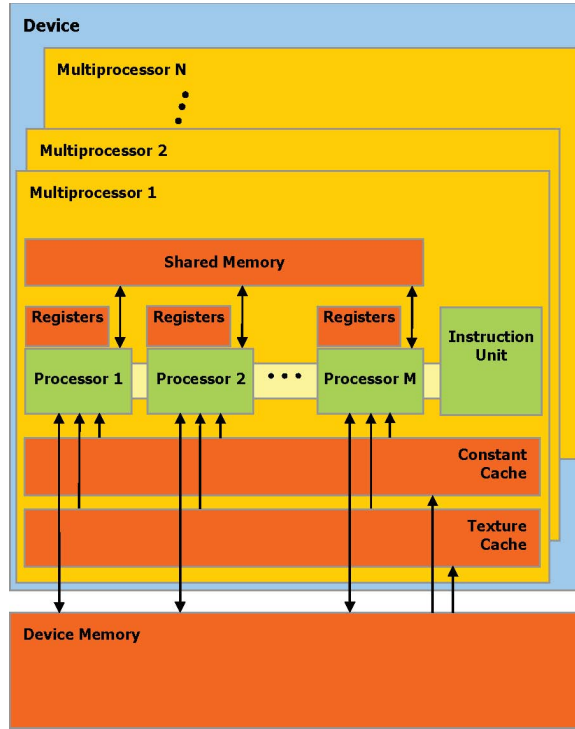


Figure 4: NVIDIA's GPU hardware model [7]

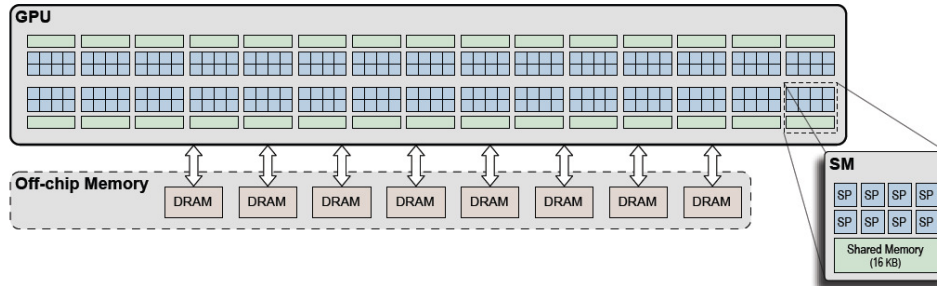


Figure 5: NVIDIA's Tesla C1060 GPU [16]

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 [7]. 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) [19]. The C1060 consumes 188W of power. The architecture of the NVIDIA Tesla C2050 (also known as Fermi) corresponds to Figure 4 with $N = 14$ and $M = 32$. So, a C2050 has 14 SMs and each SM has 32 SPs giving the C2050 a total of 448 SPs or cores. Although each SP of a C2050 has its own integer, single- and double-precision units, the 32 SPs of an SM share 4 single-precision transcendental function units. An SM has 64KB of on-chip memory that can be “*configured as 48KB of shared memory with 16KB of L1 cache (default setting) or as 16KB of shared memory with 48KB of L1 cache*” [7]. Additionally, there are 32K 32-bit registers

per SM and 3GB of off-chip device/global memory that is shared by all 14 SMs. The peak performance of a C2050 is 1,288 GFlops (or 1.288TFlops) of single-precision operations and 515GFlops of double-precision operations and the power consumption is 238W [4]. Once again, the peak of 1,288GFlops requires that MADDs and SF instructions be dual issued. When there are MADDs alone, the peak single-precision rate is 1.03GFlops. Notice that the ratio of power consumption to peak single-precision GFlop rate is 0.2W/GFlop for the C1060 and 0.18W/GFlop for the C2050. The corresponding ratio for double-precision operations is 2.4W/GFlops for the C1060 and 0.46W/GFlop for the C2050. In NVIDIA parlance, the C1060 has compute capability 1.3 while the compute capability of the C2050 is 2.0.

A Tesla GPU is packaged as a double-wide PCIe card (Figure 6) 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.



Figure 6: NVIDIA’s Tesla PCIe16 Card (www.nvidia.com)

3 Programming Model

At a high-level, a GPU uses the master-slave programming model [21] in which the GPU operates as a slave under the control of a master or host processor. In our experimental set up for single-precision matrix multiplication, the master or host is a 2.8GHz Xeon quad core processor and the GPU is the NVIDIA Tesla C1060. For double-precision matrix multiplication, the host is a XXX six core processor and the GPU is the NVIDIA Tesla 2050. We describe the programming model making explicit reference to the single-precision setup only. Programming in the master-slave model requires us to write a program that runs on the master processor (in our case the Xeon). This master program sends data to the slave(s) (in our case a single C1060 GPU), invokes a kernel or function that runs on the slave(s) and processes this sent data, and finally receives the results back from the slave. This process of sending data to the slave, executing a slave kernel, and receiving the computed results may be repeated several times by the master program. In CUDA, the host/master and GPU/slave codes may be written in C. CUDA provides extensions to C to allow for data transfer to/from device memory and for kernel/slave code to access registers, shared memory, and device memory.

At another level, GPUs use the SIMT (single instruction multiple thread) programming model in which the GPU accomplishes a computational task using thousands of light weight threads. The threads are grouped into blocks and the blocks are organized as a grid. While a block of threads may be 1-, 2-, or 3-dimensional, the grid of blocks may only be 1- or 2-dimensional. Kernel invocation requires the specification of the block and grid dimensions along with any parameters the kernel may have. This is

illustrated below for a matrix multiply kernel *MatrixMultiply* that has the parameters a , b , c , and n , where a , b , and c are pointers to the start of the row-major representation of $n \times n$ matrices and the kernel computes $c = a * b$.

```
MatrixMultiply<<<GridDimensions, BlockDimensions>>>(a,b,c,n)
```

A GPU has a block scheduler that dynamically assigns thread blocks to SMs. Since all the threads of a thread block are assigned to the same SM, the threads of a block may communicate with one another via the shared memory of an SM. Further, the resources needed by a block of threads (e.g., registers and shared memory) should be sufficiently small that a block can be run on an SM. The block scheduler assigns more than 1 block to run concurrently on an SM when the combined resources needed by the assigned blocks does not exceed the resources available to an SM. However, since CUDA provides no mechanism to specify a subset of blocks that are to be co-scheduled on an SM, threads of different blocks can communicate only via the device memory.

Once a block is assigned to an SM, its threads are scheduled to execute on the SM's SPs by the SM's warp scheduler. The warp scheduler divides the threads of the blocks assigned to an SM into warps of 32 consecutively indexed threads from the same block. Multidimensional thread indexes are serialized in row-major order for partitioning into warps. So, a block that has 128 threads is partitioned into 4 warps. Every thread currently assigned to an SM has its own instruction counter and set of registers. The warp scheduler selects a warp of ready threads for execution. If the instruction counters for all threads in the selected warp are the same, all 32 threads execute in 1 step. On a GPU with compute capability 2.0, each SM has 32 cores and so all 32 threads can perform their common instruction in parallel, provided, of course, this common instruction is an integer or floating point operation. On a GPU with compute capability 1.3, each SM has 8 SPs and so a warp can execute the common instruction for only 8 threads in parallel. Hence, when the compute capability is 1.3, the GPU takes 4 rounds of parallel execution to execute the common instruction for all 32 threads of a warp. When the instruction counters of the threads of a warp are not the same, the GPU executes the different instructions serially. Note that the instruction counters may become different as the result of “*data dependent conditional branches*” in a kernel [7]. When the compute capability is 1.3, the device-memory accesses of a half warp are coalesced into a single transaction when the data being accessed lie in the same 128-byte segment of device memory. The transaction size is reduced to 64 bytes in case the accessed data are in the same 64-byte segment and to 32 bytes when they are in the same 32-byte segment. The transaction size determines the volume of data moved.

An SM's warp scheduler is able to hide much of the 400 to 600 cycle latency of a device-memory access by executing warps that are ready to do arithmetics while other warps wait for device-memory accesses to complete. So, the performance of code that makes many accesses to device memory can often be improved by optimizing it to increase the number of warps scheduled on an SM. This optimization could involve increasing the number of threads per block and/or reducing the shared memory and register utilization of a block to enable the scheduling of a larger number of blocks on an SM.

4 The Matrix Multiplication Algorithm *GPU8*

The matrix multiplication kernel *GPU8* (Figures 8 and 9), 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,

```

__device__ void update2(float *a, float b, float *c)
{
    for (int i = 0; i < 16; i++)
        c[i] += a[i * 4] * b;
}

```

Figure 7: Updating c values when as read using `float2`

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 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 `float4s` rather than `floats` as would otherwise be the case. This reduces the time to write to shared memory.

The number of half warps is $n^2/256$. In each iteration of the `for i` loop, a half warp makes 4 128-byte transactions to read in a values and 64 64-byte transactions to read in b values. Thus, *GPU8* makes $n^3/4096$ 128-byte device-memory transactions on a and $n^3/256$ 64-byte transactions on b . Additionally, $n^2/16$ 64-byte transactions are made on c . Each transaction has 100% utilization. So, the the total number of transactions is $17n^3/4096 + n^2/16$ and the volume is $9n^3/32 + 4n^2$. By comparison, the number of transactions and volume for the *sgemm* code in CUBLAS 3.0 [2] are $5n^3/1024 + n^2/16$ and $5n^3/16 + 4n^2$, respectively.

5 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 10. 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 10.
3. *mul*(X, Y, Z) computes $Z = X * Y$ using the kernel code of Figures 8 and 9. 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


```

__global__ void GPU8 (float *a, float *b, float *c, int n)
{
  // thread code to compute one column of a 16 x 128 sub-matrix of c
  // use shared memory to hold the transpose of a
  // 16 x 64 sub-matrix of 1 x 4 sub-vectors of a
  __shared__ float as[16][65];

  // registers for column of c sub-matrix
  float cr[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

  int nDiv64 = n/64;
  int sRow = threadIdx.y;
  int sRow4 = sRow*4;
  int sCol = threadIdx.x;
  int tid = sRow*16+sCol.x;
  int aNext = (16*blockIdx.y+sRow)*n+sCol*4;
  int bNext = 128*blockIdx.x + tid;
  int cNext = 16*blockIdx.y*n + 128*blockIdx.x + tid;
  int nTimes2 = 2*n;
  int nTimes3 = 3*n;
  int nTimes4 = 4*n;

  a += aNext;
  b += bNext;
  c += cNext;

  float4 *a4 = (float4 *)a;

  for (int i = 0; i < nDiv64; i++)
  {
    *( (float4 *)&as[sCol][sRow4]) ) = a4[0];
    *( (float4 *)&as[sCol][sRow4+32]) ) = a4[nTimes2];
    __syncthreads(); // wait for read to complete
  }
}

```

Figure 8: *GPU8* (Part a)

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

```

float br0 = b[0];
float br1 = b[n];
float br2 = b[nTimes2];
float br3 = b[nTimes3];

b += nTimes4;

#pragma unroll
for (int k = 0; k < 15; k++)
{
    update2 (&as[k][0], br0, cr); br0 = b[0];
    update2 (&as[k][1], br1, cr); br1 = b[n];
    update2 (&as[k][2], br2, cr); br2 = b[nTimes2];
    update2 (&as[k][3], br3, cr); br3 = b[nTimes3];

    b+= nTimes4;
}

update2 (&as[15][0], br0, cr);
update2 (&as[15][1], br1, cr);
update2 (&as[15][2], br2, cr);
update2 (&as[15][3], br3, cr);

a4 += 16;
__syncthreads(); // wait for computation to complete
}

for (int j = 0; j < 16; j++)
{
    c[0] = cr[j];
    c += n;
}
}

```

Figure 9: GPU8 (Part b)

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 $V + 8m^2$.

```

__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 10: Kernel to add two matrices

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

6 One-Level Adaptation

6.1 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

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

Figure 11: Device-memory transaction statistics for $m \times m$ matrices

are not applied recursively). Figure 12 gives the sequence of kernel calls in a 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 11), 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 $63n^3/256 + 50n^2$.

6.2 One-Level Winograd

Our one-level GPU implementation of Winograd’s variant is given in Figure 13. 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 14 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 15 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*.

7 Multilevel Recursive GPU Adaptation

7.1 Multilevel Strassen

Figure 16 gives the recursive code for our implementation of Strassen’s method for n a power of 2. Adaptations to other values of n may be done using methods such as padding and peeling [10, 11]. 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 12) is used. When $n > \tau_2$, Strassen’s method (Figure 2) is applied recursively. In Figure 16, the notation $(X+, Y+) = Z$ refers to a single kernel that increments X and Y by Z . Such a kernel would read X ,

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 12: GPU kernels in Strassen implementation

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 16 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 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,

$$A(k, n) = 7A(k-1, n/2) + 18(n^2/4)$$

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 13: GPU kernels in Douglas et al.'s [8] implementation of Winograd variant

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

Figure 14: Transactions and volume for one-level multiplication of $n \times n$ matrices

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

Figure 15: 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 12;
  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 16: Strassen’s GPU matrix multiply

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

Figure 17: Transactions and volume for two-level multiplication of $n \times n$ matrices

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

Figure 18: Transactions and volume for three-level multiplication of $n \times n$ matrices

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

where $A(1, n)$, $T(1, n)$, and $V(1, n)$ are for a 1-level execution and are given in Figure 14.

Figures 17 through 19 give the values of A , T , and V for $k = 2, 3$, and 4. Figures 20 through 22 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).

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

Figure 19: Transactions and volume for four-level multiplication of $n \times n$ matrices

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

Figure 20: Percent reduction relative to *GPU8* for two-level multiplication

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

Figure 21: Percent reduction relative to *GPU8* for three-level multiplication

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

Figure 22: Percent reduction relative to *GPU8* for four-level multiplication

7.2 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 23. This code does 10 stand alone 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


```

Winograd(A, B, C, n)
{
  if (n <= τ1) compute C = A * B using GPU8;
  else if (n <= τ2) compute C = A * B using Figure 13;
  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 23: Winograd’s GPU matrix multiply

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 15.

Figures 17 through 19 and Figures 20 through 22, 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*.

8 Experimental Results

8.1 Single Precision Experiments

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

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

Figure 24: Run time (seconds) on the Tesla C1060

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 $\tau_1 = \tau_2/2$.

8.1.1 Run Time

Figures 24 and 25 give the time take by our various matrix multiplication codes. *Strassen* and *Winograd* had best performance when $\tau_2 = 4096$ while *tStrassen* and *tWinograd* performed best when $\tau_2 = 2048$. Figures 24 and 25 show 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 26 through 29 plot the percent reduction in run time, number of arithmetics, number of transactions, and volume achieved by *Strassen* and *Winograd* relative to *GPU8*. As can be seen, the speedup (reduction in run time) most closely tracks the reduction in volume.

8.1.2 Accuracy

The primary reason Strassen’s algorithm has not found wide application is that it is less numerically stable than the classical $O(n^3)$ algorithm [9]. We assess the numerical accuracy of Strassen’s algorithm and Winograd’s variant using the test matrix used in [13]:

$$A = I + uv^T \quad B = I - \frac{1}{1 + v^T u} uv^T \quad C = I$$

where I is the Kronecker delta matrix or simply the identity matrix (i.e., a matrix with 1s on the diagonal and 0 elsewhere), and the vectors u and v are as below:

$$u_i = \frac{1}{N + 1 - i} \quad v_i = \sqrt{i} \quad i = 1, \dots, N$$

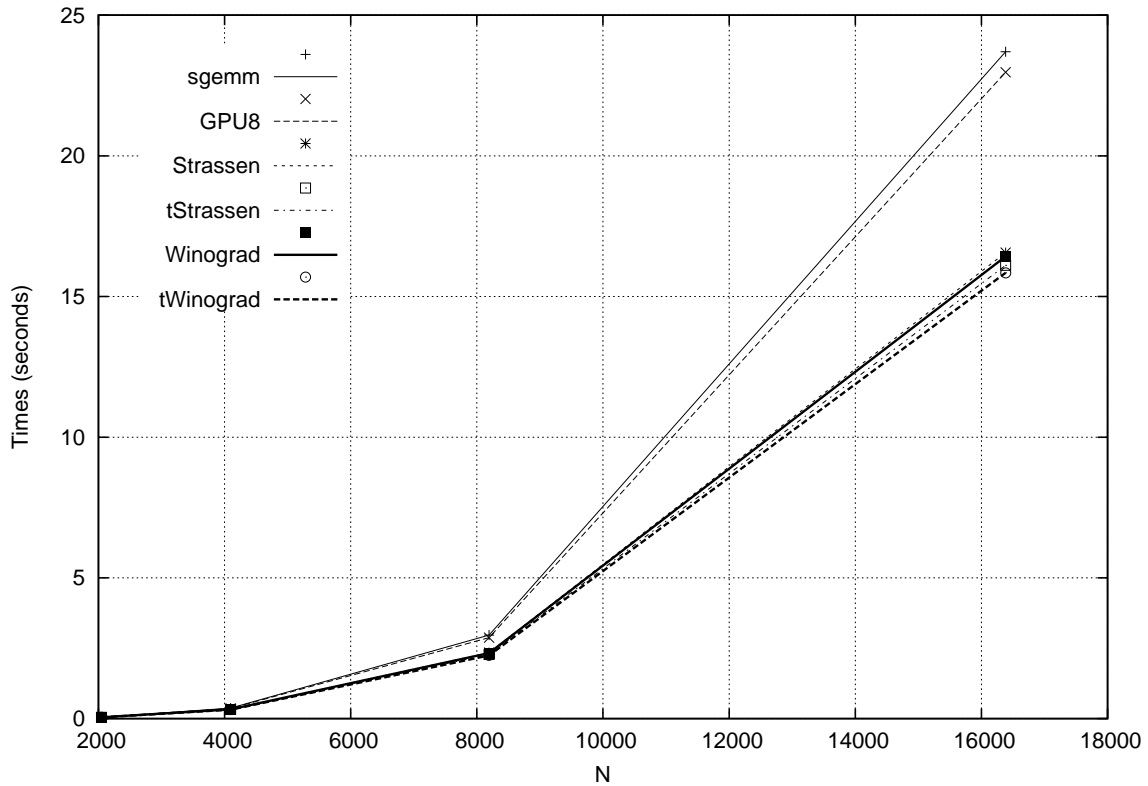


Figure 25: Plot of run time (seconds) on the Tesla C1060

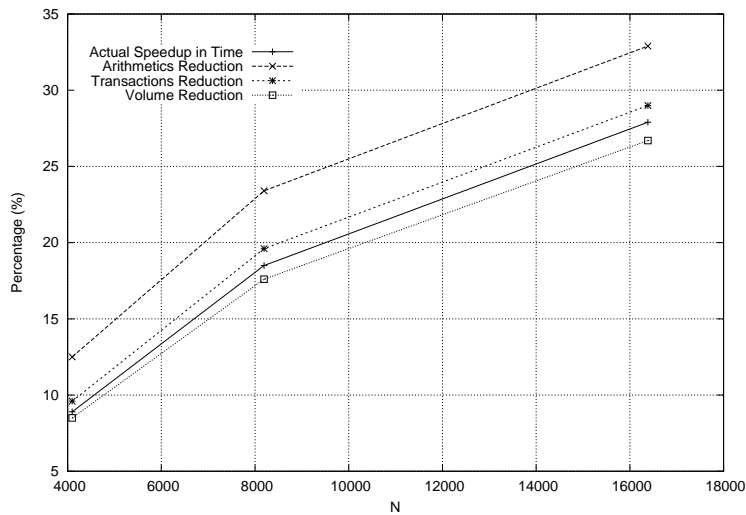


Figure 26: *Strassen* speedup when $\tau_2 = 4096$

Notice that although the product of A and B (i.e. I) is (theoretically) independent of u_i and v_i , there is some dependence, in practice, because of numerical errors introduced in the initialization of A and B on the host CPU during the computation of uv^T and doing floating-point divisions. Figure 30 gives the maximum absolute difference between an element of C as computed by each of our algorithms and the

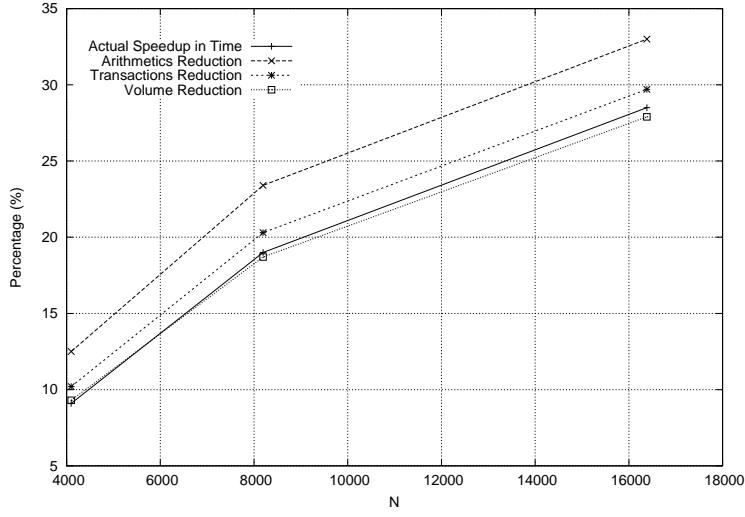


Figure 27: *Winograd* speedup when $\tau_2 = 4096$

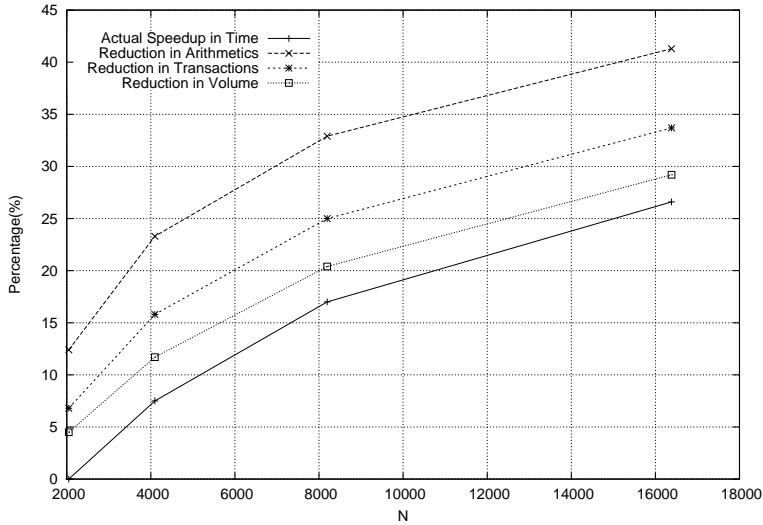


Figure 28: *Strassen* speedup when $\tau_2 = 2048$

ground truth I and Figure 31 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 . Since the use of texture memory does not impact accuracy, Figures 30 and 31 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

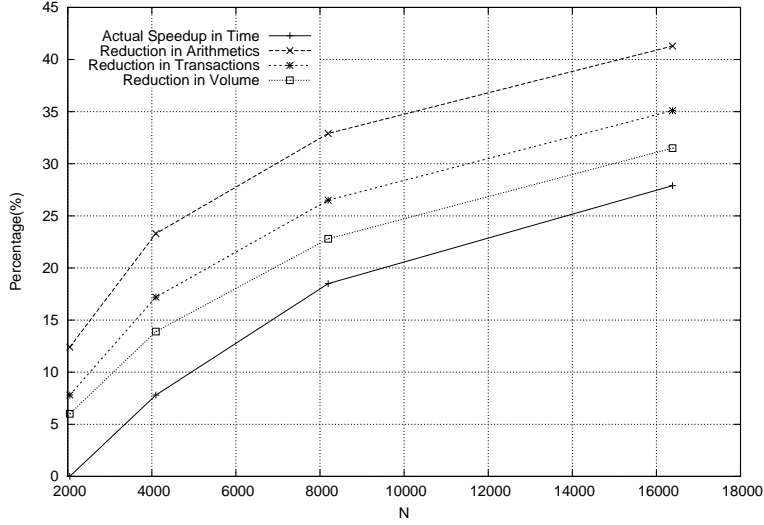


Figure 29: *Winograd* speedup when $\tau_2 = 2048$

Algorithm	τ_2	2048	4096	8192	16384
$O(n^3)$ on CPU	-	7.9e-5	1.6e-4	2.3e-4	3.9e-4
<i>sgemm</i>	-	8.1e-5	1.6e-4	2.4e-4	3.9e-4
<i>GPU8</i>	-	8.1e-5	1.6e-4	2.4e-4	3.9e-4
<i>Strassen</i>	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
<i>Winograd</i>	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 30: 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
<i>sgemm</i>	-	6.6e-8	5.6e-8	4.7e-8	3.3e-8
<i>GPU8</i>	-	6.6e-8	5.6e-8	4.7e-8	3.3e-8
<i>Strassen</i>	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
<i>Winograd</i>	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 31: Average errors

12 times that for *Strassen*.

8.1.3 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, $\tau_1 < n \leq \tau_2$

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

Figure 32: Maximum errors when $n = 16384$

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 32 through 35 give the maximum and average errors as a function of the level of the execution for the case $n = 16384$ and Figures 36 through 39 give the run time and 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 13 fold increase in the maximum error and an almost 17 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 425 fold increase in the average error.

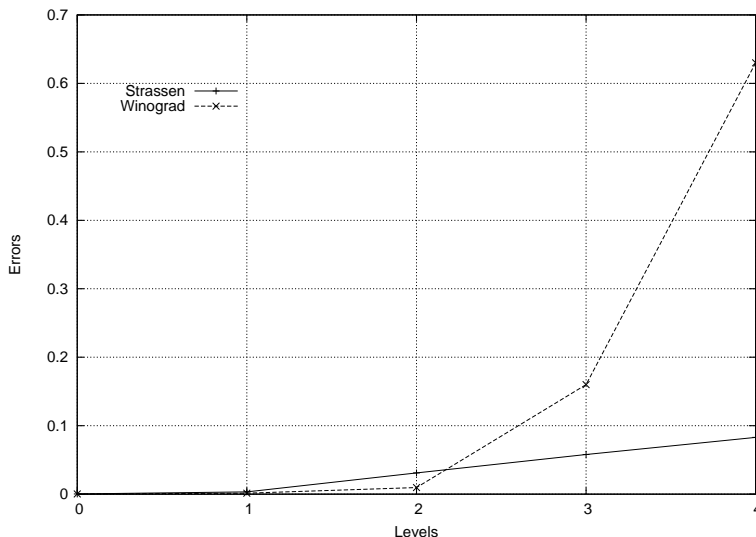


Figure 33: Plot of maximum errors when $n = 16384$

8.2 Double Precision Experiments

Double precision versions of *Strassen* and *Winograd* were developed for the Tesla C2050 (we used the C2050 as this GPU has one double-precision unit per processor core while in the C1060 each group of 8 processor cores shares a double-precision unit) and benchmarked against *dgemm*, the double-precision matrix multiply kernel included in SDK 3.2 for the C2050. Our double-precision adaptations, *dStrassen*

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

Figure 34: Average errors when $N=16384$

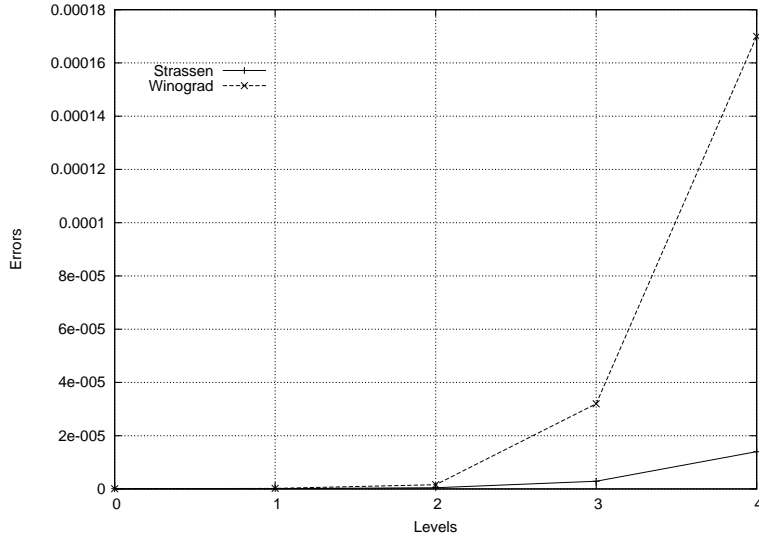


Figure 35: Plot of average errors when $n = 16384$

Algorithm	0-level	1-level	2-level	3-level	4-level
<i>Strassen</i>	22.971	20.223	18.025	16.561	16.855
<i>tStrassen</i>	22.971	20.167	17.967	16.374	16.107
<i>Winograd</i>	22.971	20.208	17.970	16.425	16.555
<i>tWinograd</i>	22.971	20.152	17.902	16.232	15.846

Figure 36: Time when $n = 16384$

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

Figure 37: Speedup(%) over *sgemm*/*GPU8* when $n = 16384$

and *dWinograd*, simply replaced the invocation of *GPU8* by an invocation of *dgemm* in Figures 16 and 23 and used double-precision versions of the kernels of Figure 11. We note that a version of *GPU8* optimized for the C2050 is not available and so we did not experiment with *GPU8*. Since *dgemm* makes effective use of texture memory and *dgemm* is invoked by *Strassen* and *Winograd* when $n \leq \tau_1$, the analogs of *tStrassen* and *tWinograd* for double-precision computations are identical to *dStrassen* and *dWinograd*, respectively.

Since the Tesla C2050 has only 3GB of device memory and since double precision matrices need twice the memory needed by single precision matrices, the largest matrix we could experiment with had $n = 8192$. Both *dStrassen* and *dWinograd* exhibited best run-time when $\tau_2 = 4096$. As in the single-precision experiments, we set $\tau_1 = \tau_2/2$.

Figures 40 through 42 give the run time for *dgemm*, *dStrassen*, and *dWinograd* as well as the speedup attained by *dStrassen* and *dWinograd* over *dgemm* for $n = 4096$ and 8192. When $n = 8192$

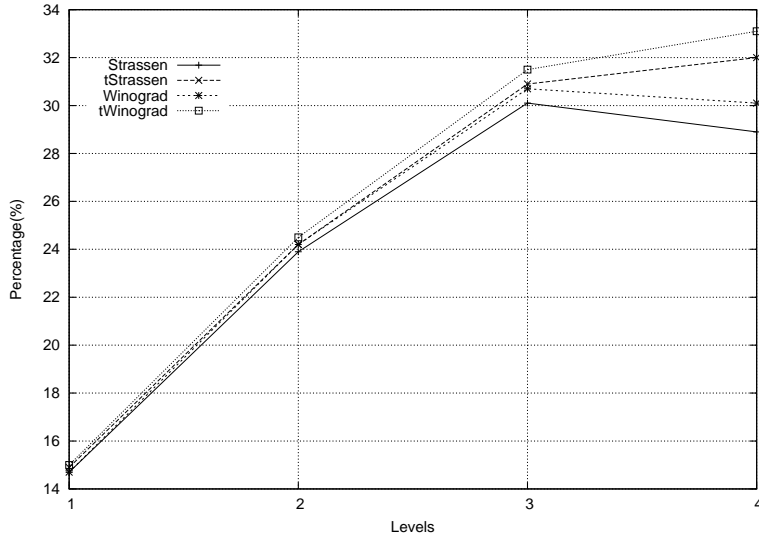


Figure 38: Speedup relative to *sgemm* when $n = 16384$

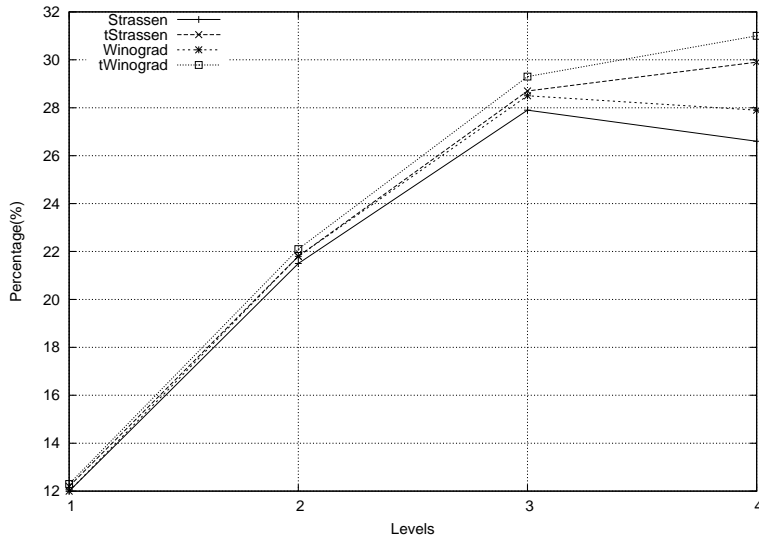


Figure 39: Speedup relative to *GPU8* when $n = 16384$

Algorithm	Time		Speedup over <i>dgemm</i>	
	4096	8192	4096	8192
<i>dgemm</i>	0.456	3.634	-	-
<i>dStrassen</i>	0.404	2.900	11.4%	20.2%
<i>dWinograd</i>	0.402	2.870	11.8%	21.0%

Figure 40: Double precision time and speedup on C2050, $\tau_2 = 4096$

the speedup attained by *dStrassen* was 20.2% and that attained by *dWinograd* was 21%. These compare with speedups of 21% and 21.5% attained by *Strassen* and *Winograd* relative to *sgemm* when $n = 8192$.

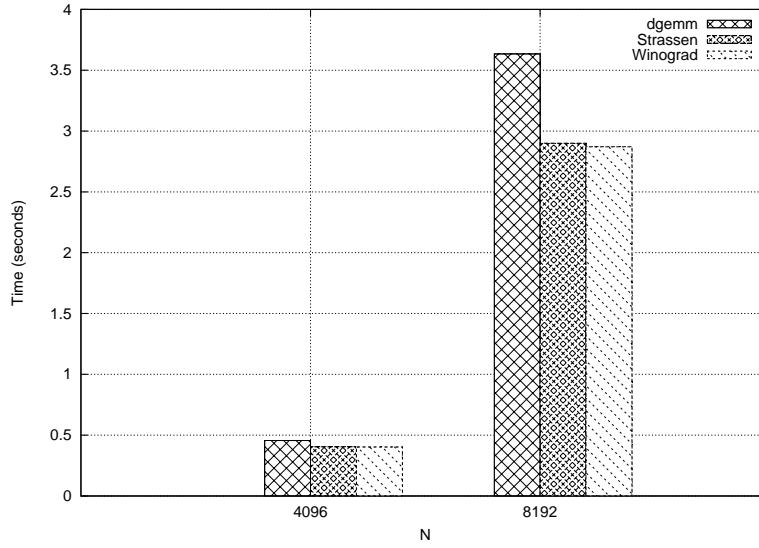


Figure 41: Double precision run time

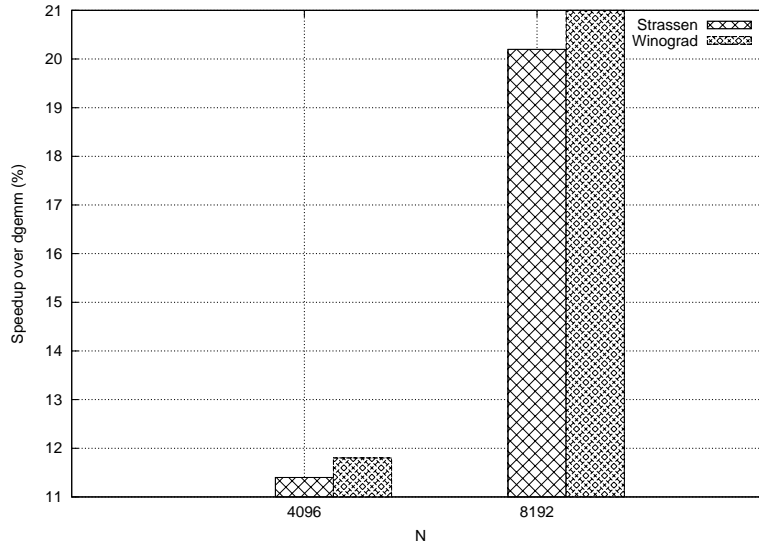


Figure 42: Double precision speedup relative to *dgemm*

To assess the accuracy of computation in double precision mode, we used the same test matrix as used in Section 8.1.2. Figure 43 gives the maximum and average errors in the computed product matrix when $n = 8192$. While a double-precision computation using the classical matrix multiplication algorithm on the CPU has the same error characteristics as *dgemm*, *dStrassen* and *dWinograd* have errors that are an order of magnitude higher; the errors using *dStrassen* are about half those using *dWinograd*.

We conducted an additional experiment to gauge the accuracy of the double precision algorithms. In this experiment, we generated 10 different 8192×8192 matrices with elements randomly selected from the range $[-1, 1]$. The maximum and average errors for each computation were computed relative to the results obtained by the classical matrix multiply algorithm on the CPU and then normalized by the average of the absolute values of the elements computed by the classical CPU algorithm. Figure 44 gives

	$O(n^3)$ on CPU	<i>dgemm</i>	<i>dStrassen</i>	<i>dWinograd</i>
Maximum	6.5e-13	6.5e-13	4.3e-12	7.5e-12
Average	1.2e-16	1.2e-16	1.3e-15	3.1e-15

Figure 43: Errors for test matrix of Section 8.1.2 when $n = 8192$ and $\tau_2 = 4096$

	<i>dgemm</i>	<i>dStrassen</i>	<i>dWinograd</i>
Maximum	8.9e-16	3.5e-14	1.4e-14
Average	5.3e-17	3.0e-15	1.8e-15

Figure 44: Normalized maximum and average errors for ten random matrices, $n = 8192$ and $\tau_2 = 4096$

the maximum of the normalized maximum errors and the average of the normalized average errors.

9 Conclusion

We have developed efficient GPU implementations of Strassen’s and Winograd’s matrix multiplication algorithms. Our experiments indicate that for single-precision arithmetic a speedup of 32% is achieved by Strassen’s algorithm while Winograd’s variant achieves a speedup of 33% relative to the *sgemm* code in CUBLAS when multiplying 16384×16384 matrices. Our double-precision implementations of Strassen’s and Winograd’s algorithms, respectively, achieve a speedup of 20.2% and 21% relative to *dgemm* when the matrix size n is 8192. These speedup, however, comes 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 $n = 16384$ and about 1 order of magnitude higher than for *dgemm* for $n = 8192$. The average numerical error introduced by Strassen’s and Winograd’s algorithms are, respectively, 2 and 3 orders of magnitude higher than those for *sgemm* when $n = 16384$ and about 1 order of magnitude higher than for *dgemm* for $n = 8192$. Whether the loss in accuracy is acceptable or not will depend on the application. 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.

References

- [1] D. Bailey, K. Lee, and H. Simon, Using Strassen’s algorithm to accelerate the solution of linear systems, *Jr. of Supercomputing*, 4, 357-371, 1990.
- [2] <http://icl.cs.utk.edu/magma/>
- [3] B. Boyer, C. Pernet, and W. Zhou, Memory efficient scheduling of Strassen-Winograd’s matrix multiplication algorithm, *ACM ISSAC*, 2009.
- [4] http://www.nvidia.com/object/product_tesla_C2050_C2070_us.html
- [5] D. Coppersmith and S. Winograd, ”Matrix multiplication via arithmetic progressions,” *Jr. of Symbolic Computations*, 9, 3, 251-280, 1990.
- [6] http://developer.download.nvidia.com/compute/cuda/3.0/toolkit/docs/CUBLAS_Library_3.0.pdf

- [7] NVIDIA CUDA Programming Guide, Version 3.0, 2010, <http://developer.nvidia.com/object/gpucomputing.html>
- [8] C. Douglas, M. Heroux, G. Sliselman, and R. Smith, GEMMW: A portable level 3 BLAS Winograd variant of Strassen's matrix-multiply algorithm, *Jr. of Computational Physics*, 110, 1-10, 1994.
- [9] N. Higham, "Exploring fast matrix multiplication within the level 3 BLAS," *ACM Trans. Math. Soft.*, 16(4), 352-368, 1990.
- [10] S. Huss-Lederman, E. Jacobson, J. Johnson, A. Tsao, and T. Turnbull, Implementation of Strassen's algorithm for matrix multiplication, *Supercomputing '96*, 1996.
- [11] S. Huss-Lederman, E. Jacobson, J. Johnson, A. Tsao, and T. Turnbull, Strassen's algorithm for matrix multiplication: Modeling, analysis, and implementation, CCS-TR-96-17, Center for Computing Sciences, 1996.
- [12] J. Li, S. Ranka and S. Sahni, "GPU matrix Multiplication," chapter in *Handbook on Multicore Computing (Editor: S. Rajasekaran)*, Chapman Hall, 2011, to appear.
- [13] I. Kaporin, "A practical algorithm for faster matrix multiplication," *Numerical Linear Algebra with Applications*, 6: 687-700, 1999.
- [14] S. Robinson, "Toward an optimal algorithm for matrix multiplication," *SIAM News*, 38, 9, 2005.
- [15] Sahni, S., *Data Structures, Algorithms, and Applications in C++*, Second Edition, Silicon Press, NJ, 2005.
- [16] Satish, N., Harris, M. and Garland, M., Designing Efficient Sorting Algorithms for Manycore GPUs, *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2009.
- [17] V. Strassen, Gaussian elimination is not optimal, *Numerische Mathematik*, 13, 354-356, 1969.
- [18] V. Volkov and J. Demmel, Benchmarking GPUs to Tune Dense Linear Algebra, *Supercomputing*, 2008.
- [19] http://wapedia.mobi/en/NVIDIA_Tesla
- [20] S. Winograd, On multiplication of 2×2 matrices, *Linear Algebra and Applications*, 4, 381-388, 1971.
- [21] Won, Y. and Sahni, S., Hypercube-to-host sorting, *Jr. of Supercomputing*, 3, 41-61, 1989.