Noname manuscript No.
(will be inserted by the editor)

Optimizing the Matrix Multiply Using Strassen and Winograd Algorithms with Limited Recursions on Many Core

Ayaz ul Hassan Khan · Mayez Al-Mouhamed · Allam Fatayer

Received: date / Accepted: date

Abstract Many core systems are basically designed for applications having large data parallelism. Strassen Matrix Multiply (MM) can be formulated as a depth first (DFS) traversal of a recursion tree where all cores work in parallel on computing each of the $N \times N$ sub-matrices, which are computed in sequence. This approach reduces the storage at the detriment of large data motion to gather and aggregate the results. Efficient Strassen implementation is challenging in many core. We propose Strassen and Winograd algorithms (S-MM and W-MM) based on three optimizations: (1) a small set of basic Algebra functions to reduce overhead, (2) invoking efficient library (CUBLAS 5.5) for basic functions, and (3) using parameter-tuning of parametric kernel to improve resource occupancy. Evaluation of S-MM and W-MM is carried out for a GPU and MIC (Xeon Phi). For GPUs, W-MM and S-MM with one recursion level outperform CUBLAS 5.5 Library with up to twice as faster for large arrays satisfying $N \geq 2048$ and $N \geq 3072$, respectively. Similar trends are observed for S-MM with reordering (R-S-MM), which is used to save storage. Compared to NVIDIA SDK library, S-MM and W-MM achieved a speedup between 20x to 80x for the above arrays. For MIC, two-recursion S-MM with reordering outperforms MKL library with between 14% to 26% for $N \geq 1024$. Similar encouraging results are obtained for 16-core Xeon-E5 server. We conclude that S-MM and W-MM implementations with a few recursion levels can be used as to further optimize the performance of basic algebra libraries.

Keywords Graphics Processing Unit (GPU) · CUDA Programming · Strassen · Fast Matrix Multiplication

Computer Engineering Department
King Fahd University of Petroleum and Minerals
Tel.: +966-13-8607633
Fax: +966-13-8602174
E-mail: ahkhan@kfupm.edu.sa, mayez@kfupm.edu.sa, g201003720@kfupm.edu.sa

1 Introduction

Modern Graphics Processing Units (GPUs) use multiple streaming multiprocessors (SMs) with potentially hundreds of cores, fast context switching, and high memory bandwidth to tolerate ever-increasing latencies to main memory by overlapping long-latency loads in stalled threads with useful computation in other threads [14]. Hiding the latency of main memory is based on an efficient zero-overhead switching mechanism, which is based on the use of a context register window. GPUs are primarily designed to run numerical computations having abundant data parallelism, i.e. having no recurrences or just marginal data dependencies. The hybrid CPU-GPU execution model allows data dependent computations to run on a host CPU, while the massive data parallel parts are efficiently executing on the GPU. The Compute Unified Device Architecture (CUDA) framework supports this combined approach of application execution. A Cooperative Heterogeneous Computing (CHC) framework [16] has also been proposed for explicitly processing CUDA applications in parallel on sets of heterogeneous processors including x86 based general-purpose multi-core processors and graphics processing units.

During the past few years the Many Core accelerators like the GPU (Nvidia) and Xeon Phi (Intel) [13,12] proved to be efficient in running parallel numerical applications. Orders of magnitude acceleration have been reported compared to traditional multi-core computers. Computational science and engineering aims at analyzing numerical algorithms to gain deeper insights in simulating a variety of problems from a wide range of application domains such as the problems that can be solved with Cellular Automata (CA) algorithms and fluid mechanics problems [8]. A large class of numerical techniques boil down to repeatedly solving a large system of linear equations with billions of unknowns. This very demanding computational task involves dense and sparse linear algebra solvers (LAS). Modern parallel computing is concerned with the analysis and development of efficient implementations of state-of-the-art numerical simulations on massively parallel computers (MPC). The efficient implementation of LAS on MPCs is obviously a challenging task which promises a significant performance gain in many core accelerators as compared to classical shared-memory multiprocessors (SMP) and distributed-memory systems [11,13]. The matrix operations in linear algebra iterative solvers are known to be communication bound that require new methods of solving large sparse linear systems with communication reduction approach specifically on a computing cluster with accelerators [6].

Matrix-Matrix multiplication (MM) is one fundamental component for linear Algebra solvers, combinatorial optimizations, and graph algorithms. MM is the basic linear algebra kernel that has $O(N^3)$ operations as per the standard MM algorithm, for $N \times N$ two dimensional arrays. Significant efforts has been made to obtain efficient MM implementations.

In 1969, Strassen proposed an $O(N^{log_2 7})$ algorithm for MM [25] that reduces the multiplication operations with some extra addition operations in the standard formulae. Several improvements have also been proposed on the orig-

inal Strassen's MM algorithm [27] and developed new optimized algorithms [22]. At present, the best upper bound kernel is $O(N^{2.37})$ [7]. And it is believed that an MM optimal algorithm will run in essentially $O(N^2)$ time [24]. Out of these algorithms, original Strassen's and its Winograd's variant have been considered for the implementations but still remain unnoticed due to its relative weaker numerical properties in comparison to the standard $O(N^3)$ algorithm.

Dumitrescu et. al [10] have implemented fast MM based on the Strassen [25] and Winograd [27] algorithms using the ring and torus topologies on MIMD distributed-memory multi-computers with the generalization of Hyper-Torus. The results show a good asymptotic behavior has been proved in terms of complexity and efficiency. The parallel implementations achieved speedup of 30x over an improved sequential code and a speedup of 75x over the standard sequential method. These results are consistent with those reported by Bailey [3] for the same matrix dimension on a Cray-2 supercomputer. Unlike the standard algorithm that is easily customizable, these parallel implementations are applicable only on the fixed number of processors. Authors emphasized the need to design fast matrix multiplication algorithms while recognizing the difficulties of their implementation on MIMD computers. Although fast matrix multiply algorithms are difficult to implement on MIMD distributed-memory systems, achieving scalable performance is one recognized challenge.

Matrix multiply kernels have been investigated extensively on GPUs and several optimizations have been proposed [20,9,18,15,28,26] including GEMM implementation in MAGMA and CUBLAS libraries. Out of these implementations, CUDA BLAS (CUBLAS) library includes a highly optimized matrix multiplication kernel [21] providing best performance on latest GPU architectures Fermi and Kepler.

Li et. al [17] presented both Strassen algorithm and its Winograd variant on NVIDIA C1060 GPU for integer and single-precision floating point data arrays. Strassen implementation obtained a speedup of 32 - 35 % while Winograd variant obtained a speedup of 33 - 36 % over CUBLAS 3.0 SGEMM implementations. While the maximum numerical error was about twice the magnitude of the SGEMM for single precision and zero for arrays of integers with N = 16384.

Communication cost of the Strassen's algorithm has been estimated [5] using graph expansion analysis and obtained the first lower bound on their communication costs. The calculated lower bounds show that not only does Strassen's algorithm reduce computation, but it also creates an opportunity for reducing communication. In addition, the lower bound becomes tighter as the amount of available memory grows, suggesting that using extra memory may also allow for faster algorithms. Using the above analysis, a new algorithm (CAPS) [4] based on Strassen's fast MM has been proposed. CAPS minimizes communication and attains theoretical lower bounds as identified in [5]. CAPS traverses the recursion tree of Strassen's sequential algorithm in two ways that are Breath-First-Step (BFS) and Depth-First-Step (DFS). BFS step requires more memory but reduces communication costs while a DFS step requires

little extra memory but is less communication-efficient. The implementation of the above algorithm on Cray XT4 obtains speedups of 24% - 184% for a fixed matrix dimension, where N = 94080. Lipshitz et. al [19] also evaluated the Strassen MM algorithm on Hopper (Cray XE6), Intrepid (IBM BG/P), and Franklin (Cray XT4) machines. Note that the above speedup, that has been achieved over the previous algorithms, is valid for both large matrices and small matrices when using a large number of processors. A few other approaches for optimizing MM on MIC have been reported. For example a hand-made library of operators [12] has been implemented on MIC, which allowed achieving approximately near performance as compared to the Math Kernel Library (MKL).

In this paper we propose a method for optimizing the performance of basic Strassen MM algorithm using the DFS approach due to the limited global memory space on the GPU and MIC. Proposed approach is based on three optimization steps. First, to reduce function invocation overhead a small set of basic functions is used to provide matrix multiply, matrix addition, and matrix aggregation. Second, while earlier approaches used manually optimized basic kernels, our approach is based on invoking one of the most optimized libraries (CUBLAS 5.5) and embedding it into the code using static device functions which reduces the overhead of dynamic linking (loading functions into memory at first call) at runtime. Third, to improve GPU resource utilization we developed a parametric kernel and used parameter-tuning to find the most profitable GPU resource parameters. In the evaluation we show that 1-level of Strassen MM (S-MM) and other variants with CUBLAS as the basic library outperforms the native CUBLAS for large two dimensional arrays. Similar results have been achieved when running the proposed approach on MIC and invoking the MKL library. This suggests that the proposed approach can be used as an optimization technique to further enhance the performance of CUBLAS and MKL libraries for basic algebra operations.

The rest of the paper is organized as follows: Section 2 describes the strassen matrix multiplication method and its derivations. Section 3 presents the proposed Strassen GPU implementation. Section 4.1 presents GPU and MIC architecture and programming model that were used in the evaluation. Section 5 presents the performance evaluation and discuss the obtained results. In Section 6, we conclude our work.

2 Strassen Matrix Multiplication Method

In 1969, Volker Strassen developed a recursive matrix multiplication algorithm (S-MM) [25] based on a divide and conquer strategy.

The objective is the computation of the resultant product matrix C as follows:

$$C = AB \qquad A, B, C \quad \epsilon \quad R^{2^n \times 2^n} \tag{1}$$

where A and B are square matrices over a ring R with $N = 2^n$.

In case of non-square matrices, extra rows or columns will be filled by zeros to transform it into square matrices.

All three matrices will be sub-divided into equal size blocks of matrices in such a way that

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix}, \quad B = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}, \quad C = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix} \tag{2}$$

with

$$A_{i,j}, \quad B_{i,j}, \quad C_{i,j} \qquad \epsilon \quad R^{2^{n-1} \times 2^{n-1}} \tag{3}$$

then

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned} \tag{4}$$

In the above construction still 8 multiplications are needed to calculate $C_{i,j}$ matrices. In order to reduce the number of multiplications, the following new matrices have to be defined.

$$\begin{aligned} M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\ M_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\ M_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\ M_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\ M_5 &= (A_{1,1} + A_{1,2})B_{2,2} \\ M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\ M_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}) \end{aligned} \tag{5}$$

Now, using only the above 7 multiplications, $C_{i,j}$ can be express in terms M_k as follows:

$$\begin{aligned} C_{1,1} &= M_1 + M_4 - M_5 + M_7 \\ C_{1,2} &= M_3 + M_5 \\ C_{2,1} &= M_2 + M_4 \\ C_{2,2} &= M_1 - M_2 + M_3 + M_6 \end{aligned} \tag{6}$$

The sub-division process can be done recursively till the degeneration of sub-matrices into numbers. The complexity of S-MM algorithm in terms of additions and multiplications operations can be calculated as follows:

$$f(n) = 7f(n-1) + l4^n \tag{7}$$

where f(n) denotes the number of additions performed at each level l of the algorithm.

Algorithm	Multiplications	Additions/Subtraction
S-MM	7	18
W-MM	7	15

Table 1 Number of Multiplications and Additions/Subtractions of each approach

$$g(n) = (7 + O(1))^n \tag{8}$$

where g(n)denotes the number of multiplications performed at each level.

Thus, the asymptotic complexity for multiplying matrices with $N = 2^n$ is $O([7 + O(1)]^n) = O(N^{log_2 7 + O(1)}) \approx O(N^{2.8074})$. However, this reduction in multiplications has been achieved with some reduction in numerical stability of the algorithm and the requirement of significantly more memory in comparison to the naïve matrix multiplication algorithm of $O(N^3)$. The arithmetic complexity of the algorithm per iteration is:

$$t_m(n) = n^{log_2 7} \qquad t_a(n) = 6n^{log_2 7} - 6n^2 \tag{9}$$

Where $t_m(n)$ and $t_a(n)$ respectively denote the number of matrix multiplications and the number of matrix additions.

Winograd [27] proposed an alternative approach to S-MM, which is denoted as W-MM, that reduces the number of additions to 15 only which has been proved to be the minimum for any of the rank 7 algorithm. Table 1 shows the comparison of operations in both strassen and winograd approaches in each recursion. The following equations describe the ordered steps of the W-MM algorithm:

$$\begin{array}{lll}
S_1 = A_{2,1} + A_{2,2} & M_1 = S_2 \times S_6 & \\
S_2 = S_1 - A_{1,1} & M_2 = A_{1,1} \times B_{1,1} & V_1 = M_1 + M_2 \\
S_3 = A_{1,1} - A_{2,1} & M_3 = A_{1,2} \times B_{2,1} & V_2 = V_1 + M_4 \\
S_4 = A_{1,2} - S_2 & M_4 = S_3 \times S_7 & C_{1,1} = M_2 + M_3 \\
S_5 = B_{1,2} - B_{1,1} & M_5 = S_1 \times S_5 & C_{1,2} = V_1 + M_5 + M_6 \\
S_6 = B_{2,2} - S_5 & M_6 = S_4 \times B_{2,2} & C_{2,1} = V_2 - M_7 \\
S_7 = B_{2,2} - B_{1,2} & M_7 = A_{2,2} \times S_8 & C_{2,2} = V_2 + M_5 \\
S_8 = S_6 - B_{2,1} & &
\end{array} \tag{10}$$

The complexity of W-MM algorithm is as follows:

$$t_m(n) = n^{log_2 7} \qquad t_a(n) = 5n^{log_2 7} - 5n^2 \tag{11}$$

3 Strassen Matrix Multiplication Optimization Method

Previous implementation of S-MM and W-MM algorithms on GPUs [17], denoted as H-S-MM algorithm, which uses 10 different user-defined matrix algebra kernels that are manually optimized basic kernels for matrix multiply and matrix addition. H-S-MM is based on the following intermediate operations

with re-ordering of the original Strassen and Winograd operations for reduced memory allocations.

$$\begin{aligned} Z &= X + Y \\ Z &= X - Y \\ Z &= X \times Y \\ (Y+, Z+) &= W \times X \\ (Y+, Z-) &= W \times X \\ (Y, Z-) &= W \times X \\ (Y, Z+) &= W \times X \\ Z &= W \times X + Y \\ W &= U \times V; Y+ = W; Z+ = Y; Y+ = X \\ Y &= X - V \times W; Z+ = X \end{aligned} \tag{12}$$

These implementations were run on NVIDIA C1060 GPU and the performance of H-S-MM was compared with CUBLAS 3.0 SGEMM kernel.

To enhance the S-MM performance we propose the following optimization steps:

1. **A small set of basic Algebra operators**. A set of three basic functions is used to provide the needed functionality as the basic matrix algebra kernels which are the matrix multiply (MM), matrix addition (Madd/sub), and matrix aggregation (Magg). The objective is to reduce the overhead associated with the invocation of a lower level set of basic library. We also define two special matrix algebra kernels for matrix add composition (Maddcomp) and matrix composite addition (Mcompadd) to combine the repetitive addition of different matrices into the same matrix and to reduce overhead of kernel termination and invocation.
2. **Embedding of optimized library**. Earlier approaches used manually optimized basic kernels for matrix multiply and matrix addition. In our approach we use some of the most optimized library to date, i.e. the CUBLAS 5.5 which is optimized at the level of the CUDA assembly language. This library is invoked for each MM operation in the S-MM and W-MM algorithms. We used the available static device functions of CUBLAS instead of the external library to reduce the overhead of dynamic linking of library at runtime.
3. **Optimizing GPU occupancy**. The performance of CUDA programs depends to a large extent on the use of the GPU available resource and the number of active blocks a given GPU allow. To provide a systematic approach for optimizing the basic matrix multiply, we developed a matrix-aggregation (Magg) operator as parametric kernel and used parameter-tuning (section 3.2 explains the tuning algorithm) to find the number of blocks and number of threads in each block to maximize GPU resource oc-

cupancy [1]. Specifically, the above approach allows us to find an optimal tile size for Magg kernel to guarantee the best GPU resource utilization.

3.1 Basic GPU Kernels

In the following we present the details of each Algebra operator.

- *Madd/sub:* This kernel performs the single precision matrix matrix addition/subtraction to compute the intermediate sub matrices for each of the multiplication proposed in strassen and winograd algorithms. To achieve the best possible performance, we used cublasSgeam function of CUBLAS 5.5. The function can be used to perform one of the following operations:

$$\begin{aligned} C &= \alpha A + \beta B && or \\ &= \alpha A^T + \beta B && or \\ &= \alpha A + \beta B^T && or \\ &= \alpha A^T + \beta B^T \end{aligned} \tag{13}$$

  Where α and β are scalars, and A, B, and C are matrices stored in column major format with dimensions $m \times n$. In our implementation, we have used this function for addition or subtraction of submatrices (Z = X + Y or Z = X - Y) by setting $\alpha = \beta = 1$ or $\alpha = 1, \beta = -1$ and also for copying the submatrices as required for the multiplications mentioned in section 2 by setting $\alpha = 1, \beta = 0$.
- *MM:* This kernel performs the single precision matrix matrix multiplication using the cublasSgemm function of CUBLAS 5.5. The function can be used to perform one of the following operations:

$$\begin{aligned} C &= \alpha AB + \beta C && or \\ &= \alpha A^T B + \beta C && or \\ &= \alpha AB^T + \beta C && or \\ &= \alpha A^T B^T + \beta C \end{aligned} \tag{14}$$

  Where α and β are scalars, and A, B, and C are matrices stored in column major format with dimensions A $m \times k$, B $k \times n$ and C $m \times n$. In our implementation, we have used this function with only non transpose case for seven multiplications ($M_1 - M_7$) mentioned in section 2 by setting $\alpha = 1, \beta = 0$.
- *Magg:* This is a set of four similar kernels to perform the final four operations (C11, C12, C21, C22) at the recursion termination mentioned in section 2. All of these kernels load each operand from the global memory to shared memory and then to register by applying the related operation (addition or subtraction). Then the result will be store back to the global memory. These kernels take two parameters that define the width

(TILE_X) and height (TILE_Y) of the tile to be loaded into shared memory. Code Listings 1 - 4 show the kernel functions for these operations. Based on the restructuring algorithm [1], we have used TILE_X = 32 and TILE_Y = 16 that gives best performance of these kernels by optimal resource utilization. The kernels are invoked with block dimension (TILE_X x TILE_Y) and grid dimension (N/TILE_X x N/TILE_Y) where N is the dimension of submatrices.

Listing 1 Compute C11 Kernel

```
__global__ void computeC11 (float *C, float *m1, float *m4,
    float *m5, float *m7, int width, int subWidth)
{
      int tx = threadIdx.x;
      int ty = threadIdx.y;
      int row = blockIdx.y * TILE_Y + ty;
      int column = blockIdx.x * TILE_X + tx;

      __shared__ float as[TILE_Y][TILE_X];

      float Csub;

      as[ty][tx]=m1[(row+i)*subWidth+column];
      Csub=as[ty][tx];
      as[ty][tx]=m4[(row+i)*subWidth+column];
      Csub+=as[ty][tx];
      as[ty][tx]=m5[(row+i)*subWidth+column];
      Csub-=as[ty][tx];
      as[ty][tx]=m7[(row+i)*subWidth+column];
      Csub+=as[ty][tx];

      C[(row+i)*width+column]=Csub;
}
```

Listing 2 Compute C12 Kernel

```
__global__ void computeC12 (float *C, float *m3, float *m5,
    int width, int subWidth)
{
      int tx = threadIdx.x;
      int ty = threadIdx.y;

      int row = blockIdx.y * TILE_Y + ty;
      int column = blockIdx.x * TILE_X + tx;

      __shared__ float as[TILE_Y][TILE_X];

      float Csub;

      as[ty][tx]=m3[(row+i)*subWidth+column];
      Csub=as[ty][tx];
      as[ty][tx]=m5[(row+i)*subWidth+column];
      Csub+=as[ty][tx];

      C[(row+i)*width+column]=Csub;
```

```
}
```

Listing 3 Compute C21 Kernel

```
__global__ void computeC21 (float *C, float *m2, float *m4,
    int width, int subWidth)
{
      int tx = threadIdx.x;
      int ty = threadIdx.y;

      int row = blockIdx.y * TILE_Y + ty;
      int column = blockIdx.x * TILE_X + tx;

      __shared__ float as[TILE_Y][TILE_X];

      float Csub;

      as[ty][tx]=m2[(row+i)*subWidth+column];
      Csub=as[ty][tx];
      as[ty][tx]=m4[(row+i)*subWidth+column];
      Csub+=as[ty][tx];

      C[(row+i)*width+column]=Csub;
}
```

Listing 4 Compute C22 Kernel

```
__global__ void computeC22 (float *C, float *m1, float *m2,
    float *m3, float *m6, int width,int subWidth)
{
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int row = blockIdx.y * TILE_Y + ty;
  int column = blockIdx.x * TILE_X + tx;

  __shared__ float as[TILE_Y][TILE_X];

  float Csub;

  as[ty][tx]=m1[(row+i)*subWidth+column];
  Csub=as[ty][tx];
  as[ty][tx]=m2[(row+i)*subWidth+column];
  Csub-=as[ty][tx];
  as[ty][tx]=m3[(row+i)*subWidth+column];
  Csub+=as[ty][tx];
  as[ty][tx]=m6[(row+i)*subWidth+column];
  Csub+=as[ty][tx];

  C[(row+i)*width+column]=Csub;
}
```

– *Maddcomp:* this kernel is defined to perform the operation $(\alpha X+, \beta Y+) = Z$ that adds matrix Z in one or both X and Y based on the values of α and β. Code Listing 5 shows the kernel function for this operation.

Listing 5 matrix add composition

```
__global__ void Maddcomp(float *X, float *Y, float *Z, int
    alpha, int beta, int width, int wX, int wY, int wZ)
{
  int i = blockIdx.y*blockDim.y+threadIdx.y;
  int j = blockIdx.x*blockDim.x+threadIdx.x;

  float z = Z[i* wZ + j];

  X[i * wX + j] += alpha * z;
  Y[i * wY + j] += beta * z;
}
```

– *Mcompadd:* this kernel is defined to perform the operation $Z = (\alpha X+, \beta Y+)$ that adds one or both matrices X and Y in matrix Z based on the values of α and β. Code Listing 6 shows the kernel function for this operation.

Listing 6 matrix composite addition

```
__global__ void Mcompadd(float *X, float *Y, float *Z, int
    alpha, int beta, int width, int wX, int wY, int wZ)
{
  int i = blockIdx.y*blockDim.y+threadIdx.y;
  int j = blockIdx.x*blockDim.x+threadIdx.x;

  Z[i*wZ+j]+=alpha*X[i*wX+j]+beta*Y[i*wY+j];
}
```

3.2 Parameters Tuning Algorithm

Algorithm 1 determines the optimal parameters (TILE_X and TILE_Y) for the generated parametric CUDA kernel. The kernel will be executed with TILE_X x TILE_Y block dimension and N/TILE_X x N/TILE_Y grid dimension.

The algorithm evaluates the generated parametric kernel with various possible combinations of TILE_X and TILE_Y. The pruning of the list of possible parameters is used at three levels to reduce the repeated compilation and execution of the kernel. The three levels of pruning are as follows:

1. **Array Block Level:** This will skip those values of TILE_X and TILE_Y which do not equally distribute the number of resultant elements among all threads (see step 3 and 7).
2. **Kernel Block Level:** This will skip those values of TILE_X and TILE_Y which does not distributed the number of resultant elements among all kernel blocks (see step 11).
3. **Active Block Level:** This will skip those combinations of parameters which requires more than the available resources such as number of registers, shared memory and number of threads per SM (see step 32).

The algorithm takes kernel source file, resultant matrix dimension (N) and GPU Compute Capability (CC) for the target GPU device. It first loads the

Algorithm 1 Parameters Tuning Algorithm

findOptimalParameters(N, CC)

Parameters:
N = Matrix Dimension
CC = Compute Capability of GPU Device
Constants and Keywords:
params = Structure of GPU Parameters
minTW = Minimum TILE_X, maxTW = Maximum TILE_X
minTH = Minimum TILE_Y, maxTH = Maximum TILE_Y
KB = Kernel Blocks
RPT = Registers Per Thread
ShM = Shared Memory Per Block
RPB = Registers Per Block
WPB = Warps Per Block
ABW = Active Blocks Limit based on WPB
ABShM = Active Blocks Limit based on ShM
ABR = Active Blocks Limit based on RPB
CompleteParamsList = Set of all Possible Kernel Parameters
CandidateParamsList = Set of Candidate Kernel Parameters
OptimalParams = Set of final Optimal Kernel Parameters

Algorithm:

```
Load params for compute capability of CC
for tw=minTW to maxTW Step *2 do
    if N mod tw ≠ 0 then
        continue
    end if
    for th=minTH to maxTH Step *2 do
        if N mod th ≠ 0 then
            continue
        end if
        KB = INT(N/tw) * INT(N/th)
        if KB = 0 then
            continue
        end if
        Compile kernel to determine the required RPT and ShM
        RPB=INT(RPTxbs,params.RegisterAllocationUnitSize)
        WPB=CEILING(bs/params.ThreadsPerWarp)
        ABW=FLOOR(params.WarpsPerSM/WPB)
        ABShM=FLOOR(params.MaxSharedMemory/ShM)
        ABR=FLOOR(params.RegisterFileSize/RPB)
        Add parameters into CompleteParamsList
    end for
end for
for all p in CompleteParamsList do
    if p.ABW>0 and p.ABShM>0 and p.ABR>0 then
        Add p into CandidateParamsList
    end if
end for
mintime = 0
for all p in CandidateParamsList do
    Execute the kernel with TILE_X=p.tw, TILE_Y=p.th
    determine execution time (ktime) of the kernel
    if ktime>0 and (mintime=0 or mintime>ktime) then
        mintime = ktime
        OptimalParams = p
    end if
end for
```

parameters for the given compute capability such as Register Allocation Unit Size, Threads Per Warp, Warps Per SM, Maximum Shared Memory Per Block, Register File Size, and etc (see step 1). It then loop over all possible combination of TILE_X and TILE_Y limiting to the range given by user with appropriate pruning (Array Block Level and Kernel Block Level) of the parameters as explained above. For each combination, it compiles the kernel with ptx information to determine the required number of Registers Per Thread (RPT) and Shared Memory (ShM) per block (see step 14). Then, calculate and store the restricted number of Active Blocks by Warp (ABW), Active Blocks by Shared Memory (ABShM), and Active Blocks by Registers (ABR) into a structured list (CompleteParamsList) (see steps 15 - 20). Then, it performs parameters pruning at Active Block Level and generate a list of possible optimal parameters (CandidateParamsList) (see steps 23 - 27). Finally, it execute the kernel for each combination of parameters in CandidateParamsList and determine the final optimal parameters (OptimalParams) that gives the minimum execution time (see steps 29 - 36).

3.3 Algorithm Implementations

3.3.1 Strassen Adaptation (S-MM)

Algorithm 2 shows the pseudo code of S-MM. Following the strassen block partitioning, the function starts by calculating the dimension of current matrix blocks that is dividing each dimension by 2 (step 1). Our current implementation works only for square matrices so the height and the width of 2D matrices will be equal. Then, in step 2, it performs global memory allocations to store the intermediate results of seven multiplications that are m1, m2, m3, m4, m5, m6, m7 and also for their operands that will be resulted by some addition or copy operation on matrix blocks that are m1a, m1b, m2a, m2b, m3a, m3b, m4a, m4b, m5a, m5b, m6a, m6b, m7a, m7b. Each of these sub-matrices are of the dimension subWidth x subWidth. These operands will be calculated in step 3 and 4. In Step 5 - 7, the threshold of current matrix dimension (BLOCK_THRESHOLD) is checked to decide whether to continue with recursion or terminate recursion. After recursion termination, synchronization among all threads is required to avoid data hazards (step 16) as it may be possible that elements of matrices m_1 to m_7 be read by some threads which may be calculated by some other threads in different thread blocks. In step 17, the resultant matrix C is computed and the global memory resources is de-allocated in step 18 which were allocated in step 2.

3.3.2 Strassen with Re-ordered Steps (R-S-MM)

The Original implementation of Strassen suffer from memory usage [17], and it is not practical due to the size of memory that it needs for huge matrices. We implemented a reorder algorithm of Strassen to reduce memory allocations

Algorithm 2 Our Strassen Implementation Pseudo code (S-MM)

$S\_MM(A, B, C, width)$

Parameters:
A=matrix A, B=matrix B, C=matrix C
width=dimension of the matrices (assuming square matrices)
subWidth=represent the dimension of the current block partitions
$BLOCK\_THRESHOLD$=Stop Recursion Condition
Note: *Madd/sub, MM, and Magg* are the basic kernels as explained in section 3.1

Execution Steps:
1: subWidth=width/2
2: Allocate space in global memory for each of the submatrices to store intermediate results of dimension subWidth
3: Calculate m1a=A11+A22,m1b=B11+B22,m2a=A21+A22,m2b=B11,m3a=A11,m4a=A22,m5a=A11+A12,m5b=B22,m6b=B11+B12, m7b=B21+B22 using Madd/sub
4: Calculate m3b=B12-B22,m4b=B21-B11,m6a=A21-A11,m7a= A12-A22 using Madd/-sub
5: **if** $width <= BLOCK\_THRESHOLD$ **then**
6: Calculate m1=m1a*m1b,m2=m2a*m2b,m3=m3a*m3b,m4=m4a*m4b,m5=m5a*m5b,m6=m6a*m6b,m7=m7a*m7b using MM
7: **else**
8: Call $S\_MM(m1a, m1b, m1, subWidth)$
9: Call $S\_MM(m2a, m2b, m2, subWidth)$
10: Call $S\_MM(m3a, m3b, m3, subWidth)$
11: Call $S\_MM(m4a, m4b, m4, subWidth)$
12: Call $S\_MM(m5a, m5b, m5, subWidth)$
13: Call $S\_MM(m6a, m6b, m6, subWidth)$
14: Call $S\_MM(m7a, m7b, m7, subWidth)$
15: **end if**
16: Synchronize all threads
17: Caculate C11=M1+M4-M5+M7, C12=M3+M5, C21=M2+M4, and C22=M1-M4+M3+M6 using Magg
18: Free all allocated global memory as created in step 2

by reusing the resultant matrix for storing intermediate results instead of allocating separate temporary sub-matrices. In each level of recursion they need only two matrices (T1,T2) of size (N/2L) as intermediate sub matrices. Where L is the level of recursion and N is dimension of the matrix.

Algorithm 3 shows the pseudo code of R-S-MM that is similar to the steps proposed by Junjie et. al. [17] that reduces memory allocations at each level of recursion.

3.3.3 Winograd Adaptation (W-MM)

Algorithm 4 shows the pseudo code of W-MM. The implementation of W-MM is similar to described for S-MM with the following exceptions:

- In step 2, it allocates only 15 submatrices (m_1 to m_7 and S_1 to S_8) instead of 21 submatrices.
- In step 3, it performs only 8 addition/subtraction operations instead of 14 addition/subtraction/copying operations.

Algorithm 3 Our Strassen with Re-Ordered Steps Implementation Pseudo code (R-S-MM)

$R\_S\_MM(A, B, C, width)$

Parameters:
A=matrix A, B=matrix B, C=matrix C
width=dimension of the matrices (assuming square matrices)
subWidth=represent the dimension of the current block partitions
$BLOCK\_THRESHOLD$=Stop Recursion Condition
Note: *Madd/sub, MM, Magg, Maddcomp, and Mcompadd* are the basic kernels as explained in section 3.1

Execution Steps:

1: **if** $width <= BLOCK\_THRESHOLD/2$ **then**
2: Calculate C=A*B using MM
3: **else**
4: subWidth=width/2
5: Allocate space in global memory for two submatrices (T1 and T2) to store intermediate results of dimension subWidth
6: Calculate T1=A11+A22,T2=B11+B22 using Madd/sub
7: Call R_S_MM(T1, T2, C21, subWidth) //M1 calculated
8: Calculate T1=A21-A11,T2=B11+B12 using Madd/sub
9: Call R_S_MM(T1, T2, C22, subWidth) //M6 calculated
10: Calculate T1=A12-A22,T2=B21+B22 using Madd/sub
11: Call R_S_MM(T1, T2, C11, subWidth) //M7 calculated
12: Calculate C11=C11+C21,C22=C22+C21 using Maddcomp
13: Calculate T1=A21+A22 using Madd/sub
14: Call R_S_MM(T1, B11, C21, subWidth) //M2 calculated
15: Calculate T2=B12-B22 using Madd/sub
16: Call R_S_MM(A11, T2, C12, subWidth) //M3 calculated
17: Calculate C22=C22-C21+C12 using Mcompadd //C22 calculated
18: Calculate T2=B21-B11 using Madd/sub
19: Call R_S_MM(A22, T2, T1, subWidth) //M4 calculated
20: Calculate C11=C11+T1,C21=C21+T1 using Maddcomp //C21 calculated
21: Calculate T1=A11+A12 using Madd/sub
22: Call R_S_MM(T1, B22, T2, subWidth) //M5 calculated
23: Calculate C11=C11-T2 and C12=C12+T2 using Maddcomp //C11 and C12 calculated
24: Free all allocated global memory created in step 5
25: **end if**

3.4 Tradeoffs between time-bound and storage-bound implementations

Proposed S-MM algorithm consists of traversing the recursion tree sequentially using the depth-first scheme DFS and executing each step using all available memory and processing cores. S-MM computes a linear combinations of pairwise matrix multiply and combines intermediate matrices as linear combinations of the above. S-MM is based on a traversal of the recursion tree where all the cores work in parallel on assembling each of the seven sub-matrices, which are computed in sequence. S-MM repeatedly applies the DFS steps, which reduces the required storage of intermediate results through assembling of one single matrix at a time. DFS requires allocation of only one sub-matrix because each of the intermediate results is computed in sequence. Thus there is

Algorithm 4 Our Winograd Implementation Pseudo code (W-MM)

$W\_MM(A, B, C, width)$

Parameters:
A=matrix A, B=matrix B, C=matrix C
width=dimension of the matrices (assuming square matrices)
subWidth=represent the dimension of the current block partitions
$BLOCK\_THRESHOLD$=Stop Recursion Condition
Note: *Madd/sub, MM, and Magg* are the basic kernels as explained in section 3.1

Execution Steps:
1: subWidth=width/2
2: Allocate space in global memory for each of the submatrices to store intermediate results of dimension subWidth
3: Calculate S1=A21+A22,S2=S1-A11,S3=A11+A21,S4=A12- S2,S5=B12-B11,S6=B22-S5,S7=B22-B12,S8=S6-B21 using Madd/sub
4: **if** $width <= BLOCK\_THRESHOLD$ **then**
5: Calculate m1=S2*S6,m2=A11*B11,m3=A12*B21,m4=S3*S7, m5=S1*S5,m6=S4*B22,m7=A22*S8 using MM
6: **else**
7: Call $W\_MM(S2, S6, m1, subWidth)$
8: Call $W\_MM(A11, B11, m2, subWidth)$
9: Call $W\_MM(A12, B21, m3, subWidth)$
10: Call $W\_MM(S3, S7, m4, subWidth)$
11: Call $W\_MM(S1, S5, m5, subWidth)$
12: Call $W\_MM(S4, B22, m6, subWidth)$
13: Call $W\_MM(A22, S8, m7, subWidth)$
14: **end if**
15: Synchronize all threads
16: Caculate C11=M2+M3,C12=M1+M2+M5+M6,C21=M1+M2+M4-M7, C22=M1+M2+M4+M5 using similar kernels as shown in Code Listing 1-4.
17: Free all allocated global memory as created in step 2

no redistribution of work and all cores participate in computing each resulting matrix. Each core computes the local additions and subtractions associated with the intermediate sub-matrices. In contrast, in the BFS the cores work on different subset of matrices in parallel, which requires extra memory but with reduced data motion among the cores, while DFS leads to large data motion among the cores to gather the resulting sub-matrices [19, 4].

A time-bound (TB) implementation aims at minimizing execution time by reducing the recursive function calls at base level (1-Level) of the algorithm, avoiding any unnecessary stack operations that might increase the execution time, disregarding of the required storage or by assuming no bound on storage. Thus TB can be used for S-MM and W-MM to highlight the implementation that produces the least execution time. Examples of TB implementations are S-MM and W-MM which are evaluated in the next section.

The storage-bound (SB) implementation aims at minimizing the used storage by re-ordering the original steps and use some of the resultant matrices to accumulate the intermediate results of multiplication operations and the final aggregation operations. However, this approach requires additional overhead of recursive function calls among other execution steps such as additions and

Implementation	Additions/Subtractions	Multiplications	Recursions	Memory Storage (KB)
S-MM	$7(\frac{N}{2^k})^3 + 18\sum_{i=1}^{k}(\frac{N}{2^k})^2$	$7(\frac{N}{2^k})^3$	$7(\sum_{i=1}^{k-1} 7^{k-i})$	$21(\sum_{i=1}^{k}(\frac{N}{2^i})^2) \times \frac{4}{1024}$
R-S-MM	$7(\frac{N}{2^k})^3 + 18\sum_{i=1}^{k}(\frac{N}{2^k})^2$		$7(\sum_{i=1}^{k} 7^{k-i})$	$2(\sum_{i=1}^{k}(\frac{N}{2^i})^2) \times \frac{4}{1024}$
W-MM	$7(\frac{N}{2^k})^3 + 15\sum_{i=1}^{k}(\frac{N}{2^k})^2$		$7(\sum_{i=1}^{k-1} 7^{k-i})$	$15(\sum_{i=1}^{k}(\frac{N}{2^i})^2) \times \frac{4}{1024}$

Table 2 Algorithm Complexity Analysis

subtractions. This option is useful if one may tolerate an increase in execution time provided that the minimum storage is being used which allows an implementation to use a larger problem size. Example of SB implementation is the R-S-MM.

At each computation level, S-MM (W-MM) repeatedly performs all the 18 (15) matrix additions before carrying out one in-depth recursion until reaching the deepest recursion level which is defined by a threshold on the smallest operable matrix size. When recursion pops up, S-MM (W-MM) performs one matrix multiplication. Recursion pops up until reaching the original matrix, where it proceeds on another depth-first path. The process is repeated until assembling the resulting matrix. On the other hand, R-S-MM repeatedly performs partial matrix additions in preparation of the single matrix multiplication before carrying out one in-depth recursion. This process of partial computing and recursion repeats until reaching the deepest recursion level. When recursion pops up, R-S-MM completes partial work for the matrix multiplication as well as all needed matrix additions. The pops up process continues until reaching back the original matrix at the top level. The process is re-iterated from the top level in a similar manner to S-MM.

The performance of each of the implementation depends on the number of additions/subtractions, multiplications and function recursive calls. Table 2 shows the number of additions/subtractions, multiplications, function recursive calls and additional memory allocations of each of the implementation based on the matrix dimension (N) and the level of recursion (k). Table 3 shows the number of arithmetic operations, function recursive calls and additional memory allocations for different values of N and k. Based on the analysis, W-MM shows the best performance among other implementations as it requires least number of arithmetic operations and function recursive calls. Whereas S-MM and R-S-MM has the equal number of arithmetic operations but R-S-MM has an overhead of stack operations due to more function recursive calls then the S-MM.

4 An Overview of GPU and MIC

4.1 GPU And CUDA

GPU is organized into an array of highly threaded Streaming Multiprocessors (SMs). Each SM has a number of Streaming Processors (SPs) that share control logic and instruction cache. Each GPU currently comes with up to 4.8 GB

		S-MM			R-S-MM			W-MM		
N	k	Operations	Recursions	Storage	Operations	Recursions	Storage	Operations	Recursions	Storage
512	1	236060672	0	5376	236060672	7	512	235864064	0	3840
	2	30834688	49	6720	30834688	56	640	30588928	49	4800
	3	5218304	392	7056	5218304	399	672	4960256	392	5040
1024	1	1883766784	0	21504	1883766784	7	2048	1882980352	0	15360
	2	240779264	49	26880	240779264	56	2560	239796224	49	19200
	3	35553280	392	28224	35553280	399	2688	34521088	392	20160
2048	1	15051259904	0	86016	15051259904	7	8192	15048114176	0	61440
	2	1902641152	49	107520	1902641152	56	10240	1898708992	49	76800
	3	259653632	392	112896	259653632	399	10752	255524864	392	80640

Table 3 Operations and Recursions Comparison

of graphics double data rate (GDDR) DRAM referred to as global memory (GM) that is visible to all threads in all blocks. Each SM has a shared memory (ShM) which is on-chip, readable and writable, and visible to all threads running within SM and as fast as register access. However, ShM is very small in size compared to GM.

A CUDA program is a unified source code encompassing both the host and the device code. The device code is written using ANSI C extended with keywords for labeling data-parallel functions, called kernels, and their associated data structures [14]. Each kernel initiates a set of blocks defined by the programmer as grid dimension with number of threads to be executed within each block while invoking the device kernel function. Now, the block scheduler dynamically schedules each thread block to one SM based on the availability of resources within SM while individual threads will be distributed among multiple SPs within the SM. An SM can handle at most 16 blocks at a time. Also, the possible number of concurrent blocks per SM depends on the number of warps per block, number of registers per block, and the shared memory usage per block.

Each SM schedules one warp at a time with zero overhead warp scheduling. The warp is the unit of thread scheduling in SMs. Each warp consists of 32 threads of consecutive thread ids. In the case of higher dimensional kernels, warps will be retrieved from blocks according to the row major numbering. As warps executes in SIMD fashion, if there is a high latency exception such as loading data from GM or storing results to GM then the whole warp must be suspended and its context if preserved. A DMA operation is initiated by the SM whenever it finds one or more threads within a warp to perform such a long latency memory transfer operations (accessing global memory) and schedule another warp (ready to execute) to the SPs [14].

4.2 Xeon Phi and Programming Model

Intel Many Integrated Core (MIC) Architecture is based on 64-core cache-coherent SMP, where each core features hybrid 4-way hyperthreading with wide 512-bit vectorization(VPU) to exploit data parallelism [13, 12, 23]. All coherence notifications, control, address, and data are transferred on a set of

Sub matrix	Row index	Column Index
$Sub_{1,1}$	newsize	newsize
$Sub_{1,2}$	newsize	newsize+scolx
$Sub_{2,1}$	newsize+srowx	newsize
$Sub_{2,2}$	newsize+srowx	newsize+scolx

Table 4 Submatrix Indices

10 fast ring network. Cores issue instructions in-order with a dual issue and uses two levels L1 and L2 of cache memory. Each core can access all other L2 cache via the ring network which makes a collective L2 cache size up to 32MB. VPU has 32 SIMD registers each is 512 bit. VPU supports Fused Multiply-Add (FMA) operations. The sustainable peak performance using Stream Triad benchmark2would is about 20 GFLOPS. There are 8 memory controllers supporting up to 16 GDDR5 expected to deliver up to 5.5 GT/s.

MIC operates in three modes which are (1) offload mode in which the host by transferring part of its computing to a device, (2) symmetric mode in which MIC uses MPI to communicate with other devices, and native in which the application is locally run on MIC. There are three ways for thread assignments: scattering, which evenly distribute the threads across all cores, compact, which uses minimum number of cores for assigning four consecutive threads to each core, and balance, which equally scatters threads across all cores such that successive threads are assigned to the same core. In our work we used OpenMp programming paradigm. It is a fork-join model. However, the "Native" model where the program runs in the coprocessor only and all the work done on the coprocessor.

Our experiments, shows that the original time bound S-MM implementation is unpractical on MIC with 5.6 Gbyte of memory. With the above memory, we can use a matrix size of up to 3072 with 5 level of recursions. But in Strassen reorder implementation, we may use larger matrix sizes of up to 10240. For this we focus on using the reorder implementation on MIC which uses the previously defined R-S-MM algorithm with its operation reordering. For MIC, the proposed algorithm is denoted as Str_mkl which is the same as R-S-MM algorithm but with the invocation of the MKL function CBLAS_DGEMM for single precision FP and CBLAS_DGEAM for double precision FP.

The algorithm recursively calls L levels of recursion depending on the threshold value for the smallest matrix size. For each of the operation, the new size of the matrix and the position (indices) of sub matrices are passed as illustrated on Table 4.

5 Performance Evaluation

In this section we evaluate performance of proposed algorithm implementations S-MM, R-S-MM, and W-MM on following many cores: (1)an NVIDIA GPU (sub-section 5.1), and (2) an Intel Xeon Phi (sub-section 5.2). The GPU used is a Kepler architecture codename Tesla K20c with 4.8 GB of global

memory and 48 KB of shared memory with 13 Streaming Multiprocessors. The Xeon Phi model is 7110 with 60 cores, each is running at 1.3 GHz, where each core has a 32KB L1 cache (IC and DC), 512KB L2 cache, and 5.6 GB main memory. To measure performance, we executed each implementation with 100+1 iterations and calculate execution time as average of 100 iterations excluding the first iteration to avoid the startup cost of library calling. In the following sub-sections we present performance of proposed approaches.

5.1 Evaluation on GPU

Proposed algorithms S-MM, R-S-MM, and W-MM were implemented with optimized libraries CUBLAS SGEMM (version 5.5) and NVIDIA SDK MM (version 5.5). The above algorithms were run with different BLOCK_THRESHOLD (BT) values to see the effects of recursion on each implementation.

Using one recursion level, Fig 1 shows the speedup achieved by the proposed implementations S-MM, R-S-MM, and W-MM over native CUBLAS SGEMM versus problem size. Figure 2 and 3 presents a comparison of the execution times and GFLOPS respectively of S-MM, R-S-MM, W-MM, and CUBLAS SGEMM where W-MM execution times are the least for the studied range of data. CUBLAS is faster than S-MM only for relatively small $N \times N$ matrices where $N \leq 3072$ (for S-MM) and $N \leq 2048$ (for W-MM). For larger matrices where $N > 2048$ W-MM becomes faster than CUBLAS with up to twice as faster. S-MM becomes faster (speedup of 1 to 1.9) than CUBLAS for $N > 3072$. In addition, the re-ordering algorithm R-S-MM outperforms CUBLAS when $N > 6144$. For $N \geq 2048$, the three implementations satisfy the following condition in terms of execution time $T_{W-MM} \leq T_{S-MM} \leq T_{R-S-MM}$, where T_X is the execution time of implementation X. The proposed algorithms scales well with increase in the problem size of up to the largest size that the above GPU can handle. R-S-MM was ranked last because of it's SB implementation, which is due to additional overhead of recursive function calls, as compared to S-MM and W-MM which are based on TB as explained in section 3.4.

Figures 4 shows the speedup achieved by S-MM, R-S-MM, W-MM, and CUBLAS SGEMM over the NVIDIA SDK library. Notice that S-MM, W-MM, CUBLAS, and R-S-MM are 80x, 60x, 41x, and 38x faster than the NVIDIA SDK library when $N \geq 4096$, respectively. Additionally, by inspecting Figures 1 and 4 we note that the performance of our proposed implementations scales much better than CUBLAS versus an increase in array size.

However, as we increase the level of recursion the execution time is also increased due to the overhead of additional operations in both Strassen and Winograd implementations. The reason is that the reduction in one matrix multiplication using Strassen and Winograd is not large enough to (1) offset the overhead of the matrix additions and (2) the stack operation overheads due to recursive invocations when a two recursion levels or more are executed within S-MM, R-S-MM, and W-MM. Also note that CUBLAS_Sgemm itself is highly optimized at a very low level of programming.

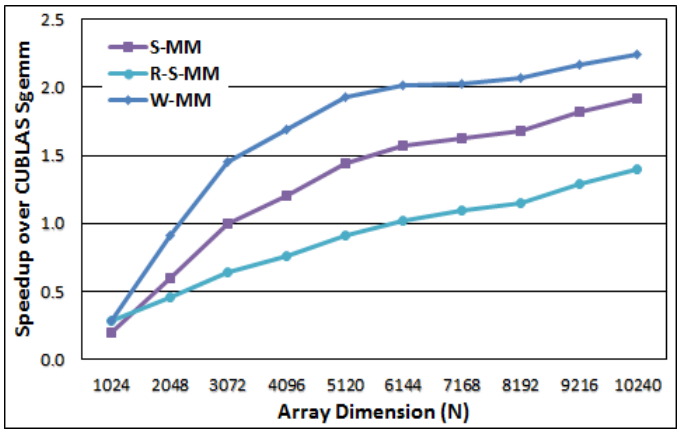

Fig. 1 Speedup of our implementations with 1-Level Recursion over CUBLAS Sgemm

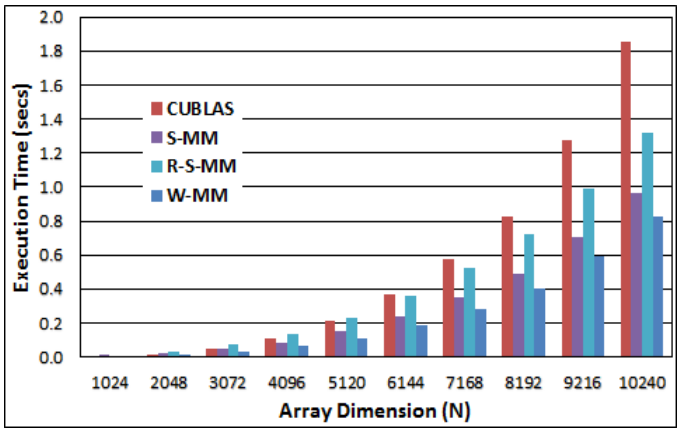

Fig. 2 Execution Time Comparison of our implementations with 1-Level Recursion and CUBLAS

Figure 5 shows the percentage increase of the execution time when two recursion levels are used over one recursion level implementations. The results show that our implementations can be more profitable with more levels of recursion with large matrices. Note that the excess in execution time decreases with increase in problem size for all proposed implementations. Also, W-MM shows a zero balance between matrix addition overhead and the saving in the matrix multiply operations. Unfortunately we are bound by N=10240 due to limitations of device memory on the Kepler K20c GPU.

5.2 Xeon Phi Results

In this section we compare the performance of proposed Strassen MM algorithm to that of MKL on MIC for two dimensional arrays. As stated before

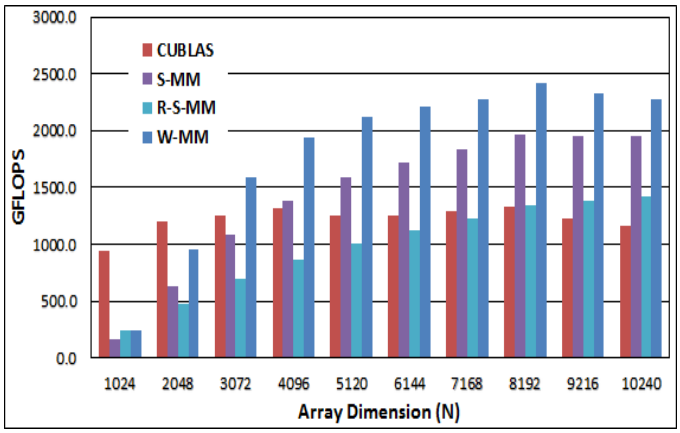

Fig. 3 GFLOPS Comparison of our implementations with 1-Level Recursion and CUBLAS

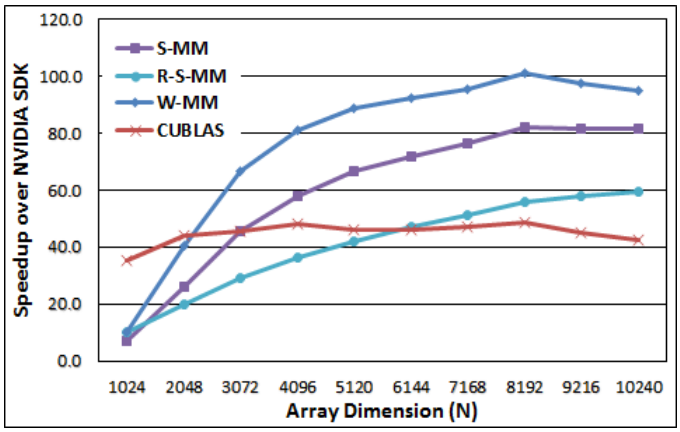

Fig. 4 Comparing Speedup over NVIDIA SDK by CUBLAS and our implementations with 1-Level Recursion

only the reorder MM algorithm denoted by Str_mkl (R-S-MM) has been implemented using C++ with OpenMp on MIC. Recall that Str_mkl is similar to R-S-MM with the difference that it invokes the MKL function CBLAS_DGEMM for matrix-matrix multiplication. On MIC, the memory allocation (malloc) for two dimensional array does not allocate contiguous memory when two dimensional indexing is used, which causes some performance degradation due to address translation overheads in the case of large array sizes. For this, we used explicit one dimensional array indexing instead of the standard two dimensional array indexing to reduce the above mentioned overheads. All the experiments were done by disabling the factorization unit and using the O2 level of compiler optimization. Also, the affinity was set ta "compact" to increase the data locality and sharing among the threads. In the case of compact affinity, a minimum number of cores is used for assigning four consecutive

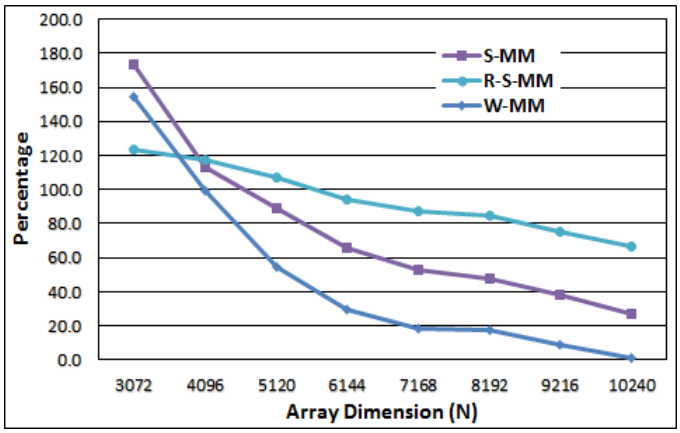

Fig. 5 Percentage increase in execution time with 2^{nd} level of recursion

threads to each core. In all the experiments, each core is running 4 OpenMp threads which binds to the hardware threads depending on the OS scheduling criteria. Finally, it is noticed that OpenMp incurs a relatively small overhead for scheduling, managing, and synchronizing threads under linux.

To assess the Str_mkl, we ran the experiments with different matrix sizes and different number of threads or cores. We experience the Str_mkl algorithm with the first five level of recursions and compare execution time to that achieved by directly invoking MKL for standard matrix multiplication. Figures 7,8,9,10 show the execution time of Str_mkl, for 1-5 recursion levels, and MKL for 8, 16, 32, and 60 MIC cores, respectively. In these Figures, the execution time of proposed Str-mkl, which invokes Strassen using CBLAS_DGEMM for Matrix-matrix multiplication, with different recursion levels are plotted with the execution time of native MKL CBLAS_DGEMM.

For one recursion level, Str_mkl is faster than MKL by a smaller factor which is between 4 to 8 % for all the experimented number of cores. The reason is that the MIC memory is not fully utilized to take advantage of using deeper recursion levels with Str_mkl. For two recursion levels, Str_mkl is faster than MKL by 14 to 26 % when the array size $N \geq 1024$ for all the experimented number of cores. In this case, there is a matching between the MIC available memory and the storage of intermediate arrays required by the Str_mkl depth-first recursive kernels. Vtune run-time profiler was used to validate the efficient use of the memory in the case of one or two recursion levels. The number of L1 and L2 misses is dramatically lower than that noticed in the case of deeper recursion levels. For three recursion levels, Str_mkl achieves shorter execution times only for large matrix size (N>=16000) and for all number of cores. Specifically, Str_mkl is 6 to 12 % faster than MKL. For four or more recursion levels the overhead of matrix additions seem to offset the benefit of saving on matrix multiply, where MKL times are shorter than those of proposed Str_mkl for all problem sizes and all the experimented number of cores.

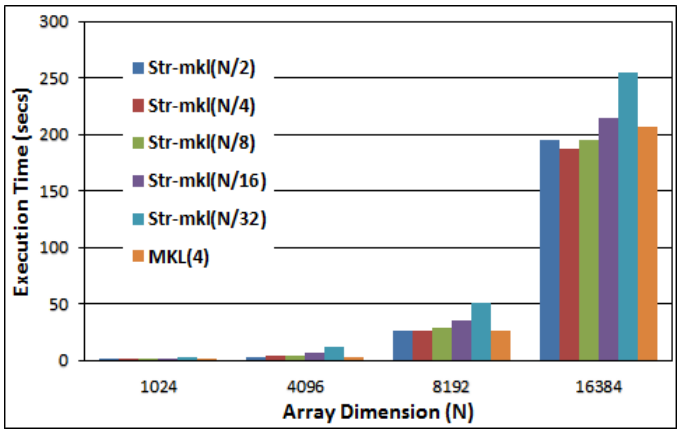

Fig. 6 Execution time of Strassen with MKL using 4 cores

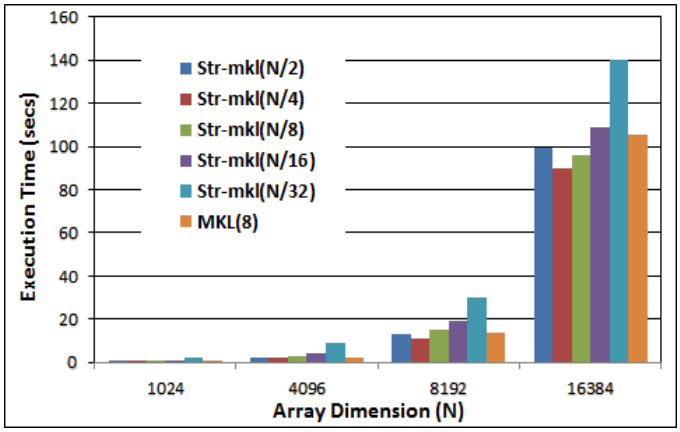

Fig. 7 Execution time of Strassen with MKL using 8 cores

Proposed algorithm Str-mkl is based on a traversal of the recursion tree where all the MIC cores work in parallel on computing each of the sub-matrices, which are computed in sequence. DFS reduces the required storage of intermediate results through assembling of one single sub-matrix at a time because it allocates one sub-matrix and each of the intermediate results is computed in sequence. Thus DFS leads to aggregate data motion among all the cores to gather the resulting matrix. In MIC the inter-core communication is carried out using eight fast rings which seems to be efficiently aggregating the core sub-matrices by implicitly (read misses) involving MIC coherence protocol. With four threads assigned to each core, threads exposed to a high latency exception, like a remote data fetching, are switched off using a fast context switching, which enables another ready thread to start execution. For the first two recursion of Str-mkl, it is clear that the above MIC latency hiding techniques seems to be profitable in trading one matrix multiplication at the

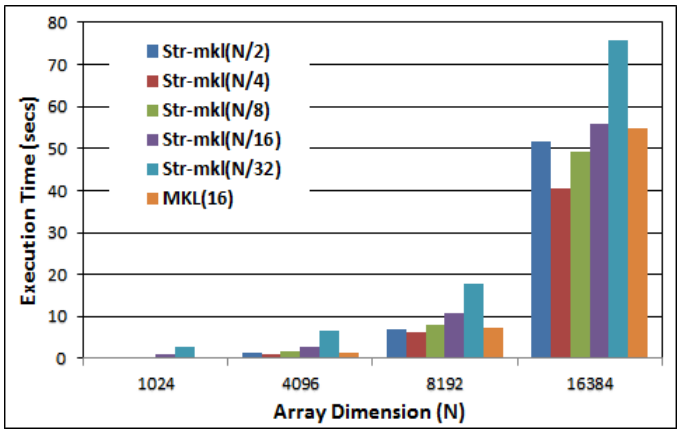

Fig. 8 Execution time of Strassen with MKL using 16 cores

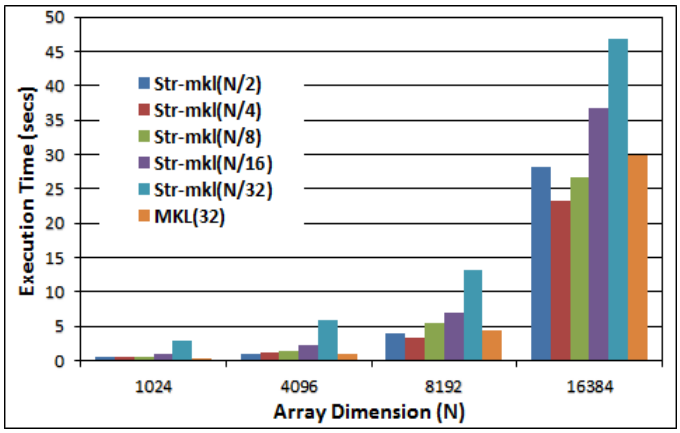

Fig. 9 Execution time of Strassen with MKL using 32 cores

cost of 18 extra matrix additions. However, the proposed Str-mkl with with three or more recursion levels is no more profitable as compared to MKL due to the following overheads:

1. The reduction in one matrix multiplication seems not sufficient to offset the overhead of additional smaller matrix additions,
2. The large data transfer (implicit by the coherence protocol) when aggregating the intermediate array data for assembling the current working matrix.
3. The CBLAS_DGEMM library time increases when the size of the matrix is smaller than 2048 specially when using large number of cores. Figure 11 shows how the CBLAS_DGEMM library performs with smaller size of matrices and larger number of threads.
4. The stack operation overhead, which is due to recursive Strassen invocations and the less efficient cache utilization when DFS works with small arrays,

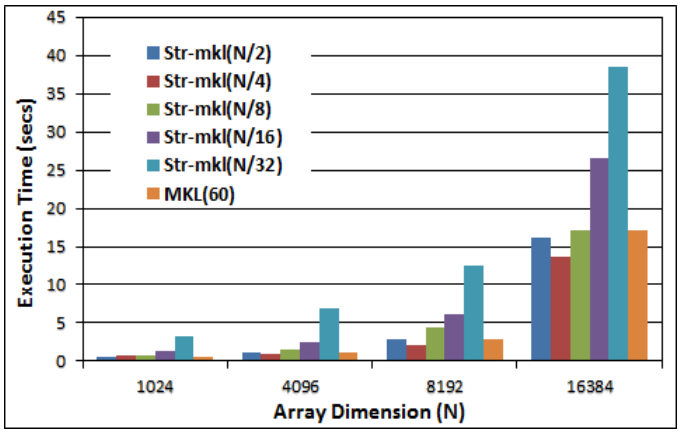

Fig. 10 Execution time of Strassen with MKL using 60 cores

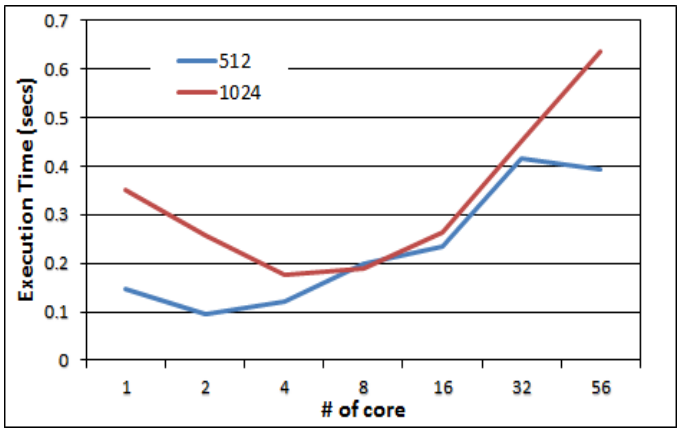

Fig. 11 Execution time of cublas_dgemm MKL library function

We have also run the same experiments on Intel Xeon E5 CPU with 16 cores and found similar trends of execution time with respect to the level of recursion in Strassen kernel execution. Figure 12 shows the execution time of our Strassen implementation and MKL CBLAS_DGEMM kernel library. Like Xeon Phi results, our Strassen implementation with up to 2 levels of recursion outperforms MKL CBLAS_DGEMM kernel.

6 Conclusion

Strassen matrix multiplication algorithm has $O(N^{2.807})$ time complexity instead of $O(N^3)$ for the standard approach. A possible formulation of Strassen algorithm is a depth first (DFS) traversal of a recursion tree where all cores work in parallel on computing each of the $N \times N$ sub-matrices, which are

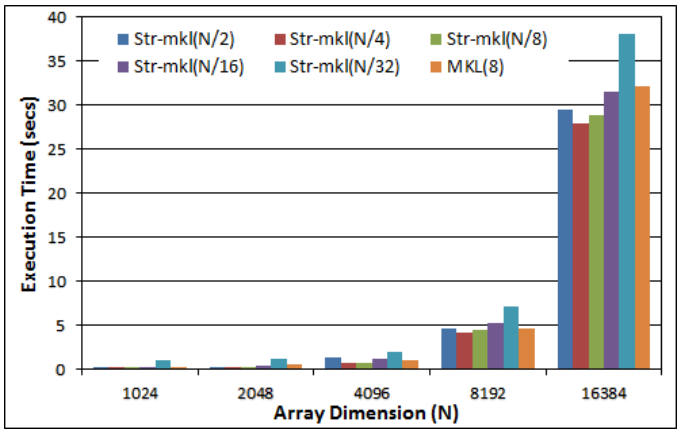

Fig. 12 Execution time of Strassen with MKL using 16 cores on Xeon E5 CPU

visited in sequence. Although this approach reduces the needed storage but it requires substantial data motion to gather and aggregate the results. We proposed Strassen and Winograd implementations based on three optimizations: (1) using a small set of basic Algebra functions to reduce overhead, (2) invoking efficient library (CUBLAS 5.5) for basic functions, and (3) using parameter-tuning of parametric kernel to improve resource occupancy. Evaluation of S-MM and W-MM is carried out on GPU and MIC. For GPUs, W-MM and S-MM with one recursion level outperforms CUBLAS 5.5 Library with up to twice as faster for large arrays satisfying $N > 2048$ and $N > 3072$, respectively. Similar trends are observed for S-MM with reordering, which is used to save storage. Compared to NVIDIA SDK library, S-MM and W-MM achieved a speedup between 20x to 80x for the above arrays. For MIC, two-recursion S-MM with reordering outperforms MKL library by 14% to 26% for $N >= 1024$. Similar encouraging results are obtained for 16-core Xeon-E5 server with improved computation scalability. This shows the profitability of S-MM procedures with limited recursion levels to tune the performance of standard MM algorithms. Specifically we conclude that proposed S-MM and W-MM implementations with a few recursion levels can be used to further optimize the performance of basic Algebra libraries. And the number of recursion levels can be increased for very large matrices.

References

1. Al-Mouhamed, M., ul Hassan Khan, A.: Exploration of automatic optimization for cuda programming. to appear in International Journal of Parallel, Emergent and Distributed Systems (IJPEDS) (2014)
2. Badin, M., D'Alberto, P., Bic, L., Dillencourt, M., Nicolau, A.: Improving numerical accuracy for non-negative matrix multiplication on gpus using recursive algorithms. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS '13, pp. 213–222. ACM (2013). DOI 10.1145/2464996.2465010

3. Bailey, D.H.: Extra high speed matrix multiplication on the cray-2. SIAM J. Sci. Stat. Comput. **9**(3), 603–607 (1988). DOI 10.1137/0909040
4. Ballard, G., Demmel, J., Holtz, O., Lipshitz, B., Schwartz, O.: Communication-optimal parallel algorithm for strassen's matrix multiplication. In: Proceedinbgs of the 24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '12, pp. 193–204. ACM (2012). DOI 10.1145/2312005.2312044
5. Ballard, G., Demmel, J., Holtz, O., Schwartz, O.: Communication costs of strassen's matrix multiplication. Commun. ACM **57**(2), 107–114 (2014)
6. Chen, C., Taha, T.: A communication reduction approach to iteratively solve large sparse linear systems on a gpgpu cluster. Cluster Computing **17**(2), 327–337 (2014). DOI 10.1007/s10586-013-0279-2
7. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. In: Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing, STOC '87, pp. 1–6. ACM, New York, NY, USA (1987). DOI 10.1145/28395.28396
8. Costarelli, S., Storti, M., Paz, R., Dalcin, L., Idelsohn, S.: Gpgpu implementation of the bfecc algorithm for pure advection equations. Cluster Computing **17**(2), 243–254 (2014). DOI 10.1007/s10586-013-0329-9
9. Cui, X., Chen, Y., Zhang, C., Mei, H.: Auto-tuning dense matrix multiplication for gpgpu with cache. In: Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on, pp. 237–242 (2010). DOI 10.1109/ICPADS.2010.64
10. Dumitrescu, B., Roch, J.L., Trystram, D.: Fast matrix multiplication algorithms on mimd architectures. Parallel Algorithms Appl. **4**(1-2), 53–70 (1994)
11. Heinecke, A., Vaidyanathan, K., Smelyanskiy, M., Kobotov, A., Dubtsov, R., Henry, G., Shet, A.G., Chrysos, G., Dubey, P.: Design and implementation of the linpack benchmark for single and multi-node systems based on intel® xeon phi coprocessor. Parallel and Distributed Processing Symposium, International pp. 126–137 (2013). DOI http://doi.ieeecomputersociety.org/10.1109/IPDPS.2013.113
12. Intel Corporation: Intel Knights Corner: Software Developer Guide (2012)
13. Intel Corporation: Intel Xeon Phi: Coprocessor Instruction Set Architecture Reference Manual (2012)
14. Kirk, D.B., Hwu, W.m.W.: Programming Massively Parallel Processors: A Hands-on Approach, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2010)
15. Kurzak, J., Tomov, S., Dongarra, J.: Autotuning gemms for fermi. Tech. Rep. 245, LAPACK Working Note (2011)
16. Lee, C., Ro, W., Gaudiot, J.L.: Boosting cuda applications with cpugpu hybrid computing. International Journal of Parallel Programming **42**(2), 384–404 (2014). DOI 10.1007/s10766-013-0252-y
17. Li, J., Ranka, S., Sahni, S.: Strassen's matrix multiplication on gpus. In: Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems, ICPADS '11, pp. 157–164. IEEE Computer Society, Washington, DC, USA (2011). DOI 10.1109/ICPADS.2011.130
18. Li, Y., Dongarra, J., Tomov, S.: A note on auto-tuning gemm for gpus. In: Proceedings of the 9th International Conference on Computational Science: Part I, ICCS '09, pp. 884–892. Springer-Verlag, Berlin, Heidelberg (2009). DOI 10.1007/978-3-642-01970-8_89
19. Lipshitz, B., Ballard, G., Demmel, J., Schwartz, O.: Communication-avoiding parallel strassen: Implementation and performance. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, pp. 101:1–101:11 (2012)
20. Nath, R., Tomov, S., Dongarra, J.: An improved magma gemm for fermi graphics processing units. International Journal of High Performance Computing Applications **24**(4), 511–515 (2010). DOI 10.1177/1094342010385729
21. NVIDIA: Cublas (2013). URL https://developer.nvidia.com/cuBLAS
22. Pan, V.Y.: How to Multiply Matrices Faster, *Lecture Notes in Computer Science*, vol. 179. Springer (1984)
23. Reinders, J.: An Overview of Programming for Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors. Intel Corporation (2012)
24. Robinson, S.: Toward and optimal algorithm for matrix multiplication (2005)

25. Strassen, V.: Gaussian elimination is not optimal. Numerische Mathematik **13**(4), 354–356 (1969). DOI 10.1007/bf02165411
26. Volkov, V., Demmel, J.W.: Benchmarking gpus to tune dense linear algebra. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC '08, pp. 31:1–31:11. IEEE Press, Piscataway, NJ, USA (2008)
27. Winograd, S.: Some Remarks on FAST Multiplication of Polynomials. Research reports // IBM. IBM (1973)
28. Yang, Y., Zhou, H.: The implementation of a high performance gpgpu compiler. International Journal of Parallel Programming **41**(6), 768–781 (2013). DOI 10.1007/s10766-012-0228-3