Optimizing GPGPU Kernel Summation for Performance and Energy Efficiency

Jiajun Wang[1], Ahmed Khawaja[1], George Biros[2], Andreas Gerstlauer[1], and Lizy K. John[1]

[1]Department of Electrical and Computer Engineering
[2]Institute for Computational Engineering and Sciences
The University of Texas at Austin
{*jiajunwang,ahmedkhawaja*}*@utexas.edu, gbiros@acm.org,* {*gerstl, ljohn*}*@ece.utexas.edu*

***Abstract*—Kernel summation is a widely used computational kernel that involves matrix-matrix multiplication (GEMM) and matrix-vector multiplication (GEMV) computational primitives. The parallelism exhibited in kernel summation suggests performance improvement when running on GPGPU. State of the art GPU solutions apply cuBLAS library but cannot exploit much of the data locality because intermediate results are written back to main memory in between key operations. This paper presents an optimized implementation that yields better performance and high energy efficiency. Our contributions are *fusing* all steps of kernel summation into the matrix multiplication code structure and *optimizing memory access ordering* to make good use of shared memory and cache hierarchy. We decompose the kernel summation problem into individual tasks with few dependencies and strike a balance between finer grained parallelism and reduced data replication. Based on hardware characteristics, we map threads to matrix elements in an interleaved way, and reposition matrix elements to avoid shared memory load and store bank conflicts. We also apply *double buffering* to hide memory access latency.**

We analyze both performance and energy benefits of our fused kernel summation compared with the implementation based on cuBLAS. We show that in low dimensions our approach achieves a speedup of up to 1.8X, and saves up to 33% of total energy in all tested problem sizes.

I. Introduction

Kernel summation is a technique used to approximate the interactions between two sets of points in a high dimensional space. Kernel summation is widely used in data analysis, electrostatics, and particle physics, most famously N-body simulations. Given $\alpha_i, \beta_j \in \mathbb{R}^K$ from a set of source points and target points, a **kernel** $\mathbb{K}(\alpha_i, \beta_j)$ describes the pairwise interaction between two points. In this work, K is denoted as the dimension of the space. We select Gaussian kernel as an example in this work. This kernel is defined as

$$\mathbb{K}(\alpha_i, \beta_j) = \exp^{-\frac{\|\alpha_i - \beta_j\|_2^2}{2h^2}} \tag{1}$$

where h is a constant. The **kernel summation** problem is to compute a scalar value V_j (associated with the target point β_j) such that

$$\mathrm{V_j} = \sum_{i=1}^{N} \mathbb{K}(\alpha_i, \beta_j) W_i \tag{2}$$

where $V \in \mathbb{R}^N$ is an N dimensional potential vector, and $W \in \mathbb{R}^N$ is an N dimensional weight vector. Solving kernel summation problem requires computing the Gaussian interaction between every source point $\alpha_i \in \mathbb{R}^N$ and every target point $\beta_j \in \mathbb{R}^N$. An efficient way to compute the N^2 interactions is to use the following identity for the Euclidean distance between two K dimensional points α_i and β_j:

$$\|\alpha_\mathrm{i} - \beta_\mathrm{j}\|_2^2 = \|\alpha_i\|_2^2 + \|\beta_j\|_2^2 - 2\alpha_i^T \beta_j \tag{3}$$

Here we focused on accelerating the evaluation of the Equation 2, for modest N (O(10,000)).

An obvious way of evaluating all the pairwise interaction (Equation 1) is to treat source point set and target point set as two K by N input matrices and apply a GEMM to compute the $-2\alpha_i^T \beta_j$ component of the Equation 3 for all $i \in N$ and $j \in N$. Each elements of the GEMM output matrix will be added to the remaining components, $\|\alpha_i\|_2^2 + \|\beta_j\|_2^2$, to achieve the computation of Equation 3. By performing an exponential function (Equation 1) on the output matrix of the previous step, we get a matrix $\mathbb{K}$ whose element locating at (row i, column j) represents the Gaussian kernel interaction between the source point α_i and the target point β_j. In the end according to Equation 2, a GEMV is applied to the matrix $\mathbb{K}$ and the weight vector W to get the result vector V. However, as we discuss here, this is not the most efficient approach. The kernel summation problem typically involves large data sets, and the long memory access latency is the crucial bottleneck of program execution. Vendor-provided Basic Linear Algebra Subprograms (BLAS) library, such as the Intel Math Kernel Library (MKL) [1] and the NVIDIA CUDA Basic Linear Algebra Subroutines (cuBLAS) [2], are often hand-optimized in assembly code and usually achieve over 80% of the peak performance. Using vendor-provided libraries brings performance benefit through the highly optimized BLAS, but it also sacrifices data locality because the intermediate matrix, as the return value of GEMM call, is written back to main memory due to its huge size not fitting into caches.

Except from losing data locality, energy spent in memory accesses is another factor urging a better solution. The increased power and energy consumption and the resulting thermal issues have become major challenges. Memory, or DRAM, operations usually take 20%-40% share of total energy consumption in many applications. DRAM energy has been reported to account for 22% in UltraSPARC T1 systems

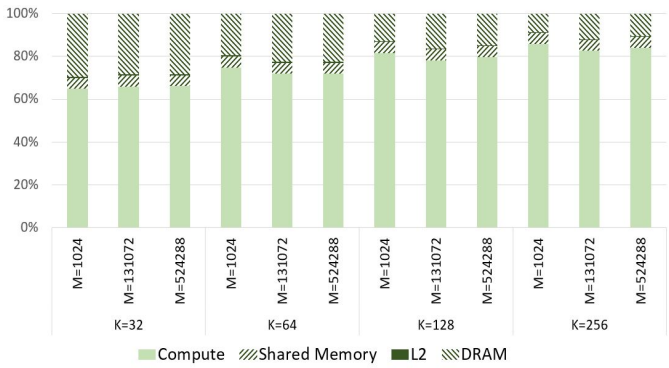


Fig. 1: Energy breakdown of kernel summation problem,with N=1024 in all cases

[3], more than 25% of data centers [4], and around 40% in a mid-range IBM eServer machine [5]. Figure 1 shows the energy breakdown of computing kernel summation using cuBLAS library. Around 10% to 30% of total energy is spent on DRAM accesses.

To address the performance and energy challenges, we propose fused kernel summation. Since most of the redundant memory accesses are coming from GEMM, we fuse steps of kernel summation into the GEMM structure. Fusion enables consumer operations to access data directly from registers and caches right after producer completes, and thus increases data locality and relieves memory burden. We decompose the problem into parallel tasks with minimal communication and synchronization, and assign each task to one GPU thread block. The GEMM part of each task is fully parallel. When a thread block completes the GEMM portion, it could go on to use intermediate value, i.e. the GEMM output, stored in registers or shared memory to perform kernel evaluation (Equation 1) without waiting for other thread blocks. The only data which a thread block stores back to main memory is a partial sum of the final result. Communication between thread blocks happens only in the GEMV part when all thread block outputs are accumulated to get a final result. Instead of the kernel waiting for all thread blocks outputs to be ready before aggregating, the reduction operation is done through each thread block adding its output to the latest reduction result in an atomic way. In other words, a thread block immediately retires after it updates the final result with its own its own output, and only one thread block is allowed to update the final result at any time. The problem size of these tasks and the size of thread blocks are selected to strike a balance between higher device occupancy and less data duplication, which infers less memory accesses.

In addition to fusion, another contribution of our work is to optimize the memory access ordering to make good use of shared memory and cache hierarchy. We maximize the compute-to-load ratio of data in main memory. Each thread block accesses main memory only once to fetch kernel input data, which is get get fully reused in caches and shared memory before being evicted back to main memory. When loading inputs from main memory into shared memory, the memory access ordering and the correlation between individual thread and fetching data are organized to avoid store bank conflicts, and the data placement in shared memory is reconstructed to avoid load bank conflicts.

Our work demonstrates that fused kernel summation provides up to 1.8X performance speedup in lower dimensions ($K < 128$) compared to the implementations based on cuBLAS, even though our CUDA-C implementation of GEMM routine is between 1.5X to 2.0X slower than the cuBLAS. At higher dimensions, the performance of GEMM routine dominates the overall performance of the kernel summation irrespective of fusion. From the energy prospective, our fused approach saves more than 80% of the DRAM access energy in all test configurations, which amounts to around 3%-33% of the total energy.

The rest of the paper is organized as follows: section II introduces prior works on the kernel summation problem, states the motivation behind our fused kernel summation and briefly discusses features of the NVIDIA Maxwell architecture. In section III we first talk about our GEMM implementation, and then introduce our fused kernel summation implementations. Our experimental setup is shown in section IV, followed by detailed results and analysis in section V. We conclude with a summary of results and potential avenues for future research in section VI.

II. Background

A. Related Work

The need for fast kernel summation methods first appeared in computational physics, for example, computing the 3D Laplace potential (reciprocal distance kernel) and the heat potential (Gaussian kernel). Kernel summations are also fundamental to non-parametric statistics and machine learning tasks such as density estimation, regression, and classification [6] [7] [8] [9]. Linear inference methods such as support vector machines [10] and dimension reduction methods such as principal components analysis [8] can be efficiently generalized to non-linear methods by replacing inner products with kernel evaluations [11]. Problems in statistics and machine learning are often characterized by very high dimensional inputs.

There are numerous studies that have proposed scalable algorithms and high-performance implementations of fast kernel summation schemes such as treecodes [12][13], fast multipole methods [14][15], particle-mesh methods [16], Ewald sums [17], etc. These algorithms can scale to billions or trillions of points for problems in two or three dimensions. However, they do not scale to higher values of K because they depend linearly or super-linearly on K. Other algorithms which are efficient for high dimension K apply GEMM defined in BLAS library [18].

The SGEMM (Single-precision GEMM) routine provided in the NVIDIA's cuBLAS library exhibits high FLOPs (floating-point operations per second). Architecture related optimizations play a vital role in GEMM performance and assembly-coding strategies are often employed for high performance. A custom implementation tailored towards the NVIDIA latest GPU architecuture, Maxwell, is available for software developers. While NVIDIA provide examples of GEMM on GPUs, they do not specifically mention how cuBLAS is implemented.

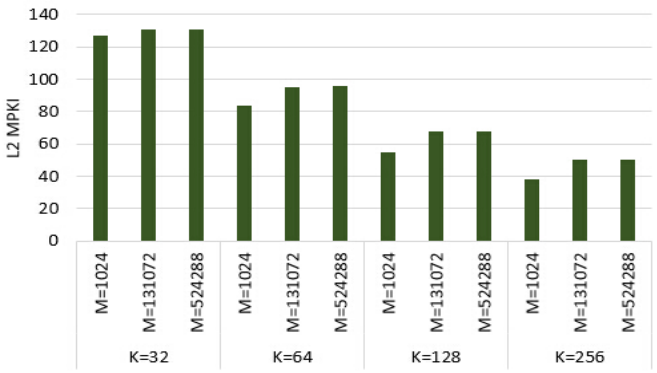


Fig. 2: L2 MPKI of kernel summation problem, with N=1024 in all cases

The fact that the cuBLAS library is closed-source makes it infeasible to fuse operations amidst the GEMM in cuBLAS. Previous works such as [19] [20] [21] discuss GEMM implementations on GPU. Although these implementations are comparable with old versions of the cuBLAS library (e.g. cuBLAS 3.5), new features in the latest version of the cuBLAS library are less understood. To the best of our knowledge, only GEMM implementation, *maxas* [22], targets the Maxwell architecture and beats the cuBLAS implementation. The *maxas* code is written in assembly and the order of assembly instructions are tuned to hide potential latency. Note that NVIDIA do not release official assembler so a self-designed compiler is used in *maxas*. The tool limitation makes it infeasible for us to use the *maxas* GEMM structure.

B. Motivation

A simplified kernel summation algorithm is shown in Algorithm 1. Inputs A and B are M-by-K and K-by-N matrices separately, and W is an N-dimensional weight vector. α_i is a K-dimensional row vector representation and β_j is a K-dimensional column vector representation.

Basic Linear Algebra Subprograms (BLAS) provide a user-friendly interface to compute GEMM and GEMV. Different vendors provide their own highly optimized BLAS libraries for users, including the Intel's MKL [1] and the NVIDIA's cuBLAS [2] library. They are often hand-optimized using assembly code. These libraries usually achieve over 80% of the peak performance. Although these are good options, they still have limitations in our application. First, we observed that using black-box BLAS libraries leads to performance degradation when the (geometric) dimension K is small (say less than 64). This is because, to the BLAS library the computation appears to be memory bound with small K; however, it could be turned into compute bound after modifying BLAS. Second, executing GEMM and other steps of kernel summation in a serial way leads to redundant memory accesses and poor data locality. Figure 2 illustrates the number of L2 misses per kilo instructions (MPKI) when applying the cuBLAS library in kernel summation problem. There is high L2 MPKI number in dimension $K = 32$. Since L2 is the last level cache of GPU memory hierarchy, program performance suffers a lot from high DRAM access latency and energy is wasted on redundant

Algorithm 1 Basic kernel summation

Inputs:
$A = [\alpha_0, \alpha_1, ..., \alpha_{M-1}]^T$, M-by-K matrix
$B = [\beta_0, \beta_1, ..., \beta_{N-1}]$, K-by-N matrix
$W = [\omega_0, \omega_1, ..., \omega_{N-1}]^T$, N-by-1 vector
Outputs:
$V = [v_0, v_1, ..., v_{N-1}]^T$, N-by-1 vector
$vec\alpha \leftarrow [\|\alpha_0\|_2^2, \|\alpha_1\|_2^2, ..., \|\alpha_{M-1}\|_2^2]^T$
$vec\beta \leftarrow [\|\beta_0\|_2^2, \|\beta_1\|_2^2, ..., \|\beta_{N-1}\|_2^2]$
// duplicating $vec\alpha$ N times to form a M-by-N matrix
$squareA \leftarrow [vec\alpha, vec\alpha, ..., vec\alpha]$
// duplicating $vec\beta$ M times to form a M-by-N matrix
$squareB \leftarrow [vec\beta, vec\beta, ..., vec\beta]^T$
// GEMM
$C \leftarrow A \times B$
$R \leftarrow squareA + squareB - 2 \times C$
for each element in R **do**
    $K(i,j) \leftarrow exp\{-\frac{R(i,j)}{2h^2}\}$
end for
// GEMV
$V \leftarrow K \times W$

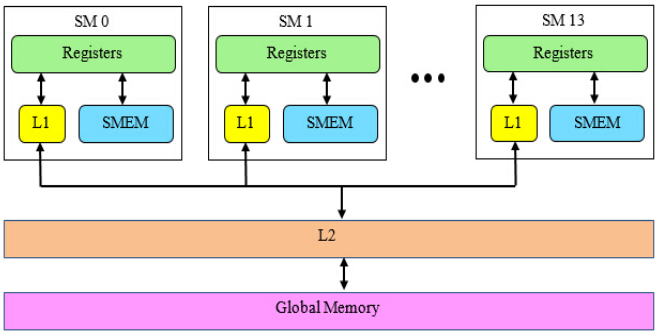


Fig. 3: GPGPU memory hierarchy

DRAM accesses. Redundant intermediate value accesses to main memory suggest opportunity in performance and energy optimization.

C. GPGPU Architecture

GPU has traditionally been an accelerator for graphics processing, but recently has seen large adoption as a general high-performance computing device. While GPU provides incredible speedups for embarrassingly data parallel applications, it often requires extensive optimization effort to achieve similar performance improvements as on other programs. Although many basic buildings blocks exist in library form for porting applications to the GPU, often a much more powerful solution is possible by tailoring these basic blocks to the specific application.

The GPGPU architecture studied in this work is the NVIDIA Maxwell architecture. It is composed of a set of compute units, a cooperative thread array (CTA) scheduler, a

unified L2 cache, and global memory. Each compute unit, also called "Streaming Multiprocessor" (SM), contains a number of arithmetic and logic units, a large register file, a shared memory, non-coherent caches, and a scheduler for units of execution. The units of execution are referred to as warps and each warp is composed of 32 scalar threads. All threads within a warp are scheduled together, and thus are implicitly synchronized. Those threads can exchange values using either shared memory or the shuffle instruction.

A CTA or a thread-block is a group of warps that execute concurrently on an SM. Threads executing on the same SM share a shared memory and are explicitly synchronized using barriers or memory fences. The CTA scheduler only allows a CTA to execute on an SM once the amount of required shared memory and registers are available. Having a large number of CTAs and scheduling warps concurrently on an SM allow the hardware to hide memory latency. When a warp experiences a long memory stall due to cache misses, bank conflicts or un-coalesced accesses, GPGPUs can hide latency by bringing in other warps to concurrently execute compute instructions.

The memory hierarchy of the Maxwell architecture is shown in Figure 3. Each SM contains separate shared memory (SMEM) and unified L1 cache. Memory accesses of all SMs must go through a shared L2 cache. The shared memory is a programmer managed cache, which is usually used in conjunction with barriers, to communicate values between threads in a CTA. Unlike the NVIDIA Fermi architecture [23], shared memory becomes an individual unit in the Maxwell architecture and L1 cache is unified with texture cache [24]. By default, the unified L1 and texture unit of the Maxwell architecture does not actually cache global loads, except for gather instructions, texture fetches, and surface writes. However, a compiler flag can be used to specify that all global loads must be cached at all levels.

The design of shared memory requires that shared memory is as large as possible and provides bandwidth high enough to service 32 threads per cycle. In order to provide high bandwidth, shared memory is laid out as a series of banks (32 for the Maxwell architecture), where each bank is four bytes wide. The width of bank is chosen to be the same as the size of float and *RGBA* data types, which are frequently used in graphics applications. Bank conflict occurs when different words in the same shared memory bank are accessed by threads in the same warp. NVIDIA makes it known that good performance from shared memory cannot tolerate bank conflicts. Programmers are encouraged to use memory access patterns that do not cause a bank conflict. All threads in a warp can issue a shared memory load in the same cycle. It is seen that all 32 banks share the same row select, which means that in order to avoid bank conflicts, in addition to using different banks, threads need to access memory within the same 128-byte region. If there is no bank conflict, 32 requests turn out to be one shared memory transaction and exploit the high bandwidth of the shared memory. The register file is also banked. Register bank conflicts are usually avoided with the help of compiler, and are only likely to occur when storing large vectors in registers.

III. Implementation of fused kernel summation

A. GEMM Algorithm Overview

The algorithmic view of our SGEMM is shown in Figure 4. Matrices A and B hold coordinate table of the source point set and the target point set respectively. For the remainder of this paper, M denotes the leading dimension of matrix A and K denotes the leading dimension of matrix B. So the matrix A is of size M by K, and the matrix B is of size K by N. We assume that the matrix A is in row major order and the matrix B is in column major order. As shown in the figure, all the three matrices are divided into sub-matrices. $C_{i,j}$ denotes a *submatrixC* which has i submatrices to its left and j submatrices on its top; A_i denotes a *submatrixA* which has i submatrices on its top; and B_i denotes a *submatrixB* which has i submatrices to its left. A thread block with a block ID (bx, by) is assigned to compute $C_{bx,by} = A_{by} \times B_{bx}$, and all thread blocks can be executed concurrently without race conditions. *submatrixA* and *submatrixB* are partitioned into tiles of size 128 by 8 and 8 by 128 separately. A thread block performs rank-8 update across the K dimension, i.e.,

$$\text{submatrixC} = \sum_{\text{i}=0}^{\text{K}/8} \text{tileA}_\text{i} \times \text{tileB}_\text{i}$$

A *submatrixC* is divided further into 16x16 microtiles. $microtileC_{i,j}$ denotes an 8 by 8 microtile which has i microtiles to its left and j microtiles on its top within the range of a *submatrixC*. Threads are organized into a 16 by 16 grid to form a thread block, and the thread with threadID (tx, ty) is corresponding to the $microtileC_{tx,ty}$. Therefore the task of a thread block computing $tileA \times tileB$ is decomposed into each thread computing $microtileC_{tx,ty} = microtileA_{ty} \times microtileB_{tx}$

Based on the data access pattern of each thread block, a *submatrixA* or a *submatrixB* will be accessed multiple time by different thread blocks. Taking a *submatrixA* of size m by K as an example, it would be accessed by N/m thread blocks, which indicates that the entire matrix A is repeatedly loaded N/m times. Even though the shared L2 cache would serve data reuse among thread blocks, it depends on the thread block scheduling policy to ensure that thread blocks accessing the same range of memory are activated at the same time. Besides, considering the limited size of L2 and the large matrix size, average L2 size per thread block is not large enough to reduce repeated memory accesses. Theoretically speaking, the value of m, which is the leading dimension of both *submatrixA* and *submatrixC*, should be sufficiently large to to reduce the N/m reloading times of matrix A. In other words, the partition of matrix A and matrix B should be relatively coarse grained in order to reduce reloading times, which directly influences the size of *submatrixC*.

Factors like GPU limits, trade-offs between high SM occupancy and less data locality, inter-influence between matrix size and matrix partition are taken into consideration when determining the size of *submatrixC* and decomposing *submatrixC* computation to thread level tasks. In the best scenario of our experiments, a thread block of dimension 16 by 16 computes a *submatrixC* of 128 by 128, and each thread computes 8 by 8 elements.

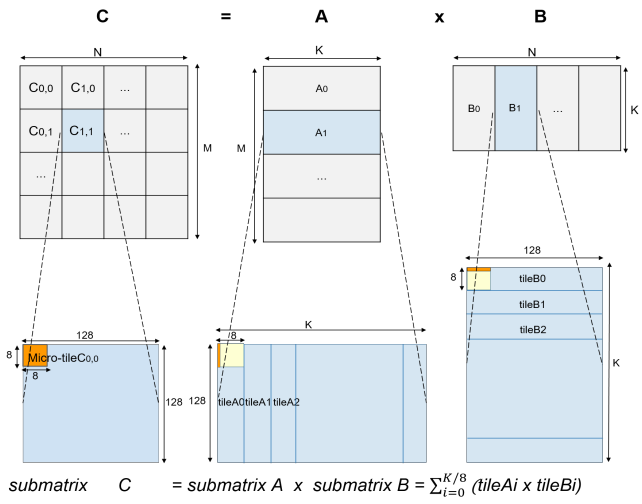


Fig. 4: GEMM algorithmic view

The number of physical registers is one of the performance bottlenecks of our solution. Programmers have the ability to choose how much shared memory space is consumed using the CUDA C programming language, but they can not explicitly control register usage without support from assembly, which is not yet released by NVIDIA. Our test machine, GTX970, provides an upper bound of 65536 registers per SM. In other words, up to 255 registers can be allocated to each thread when the thread block dimension is 16 by 16. If nothing is limiting performance, larger number of registers used per thread would lead to lower SM occupancy. In our solution, each thread takes 64 registers to hold 64 partial sums of *microtileC* in order to achieve the best data locality. Each thread performs a rank-1 update to maximize the computation to load ratio, hence vector operands from *tileA* and *tileB* take another 16 registers. Including miscellaneous essential demands like thread index and control flow variables, 96 to 128 registers are consumed by each thread and this leads to having up to two thread blocks executed simultaneously in the same SM. Although the compiler option of "*–maxregcount*" helps achieve higher occupancy, register spilling creates huge negative impact on performance because of additional L1 transactions due to spilling.

In our implementation, partial sums of *submatrixC* are stored in the thread register file, and *tileA* and *tileB* are loaded into the shared memory sequentially. We use double buffering to hide shared memory load latency. Double buffering requires size of tiles to be restricted in such a way that shared memory can hold at least two pairs of tiles at any moment. When one pair of ($tileA_i$, $tileB_i$) are used in computation, next pair of ($tileA_{i+1}$, $tileB_{i+1}$) could be loaded into shared memory. In the Maxwell assembly, each load is marked by an integer. Explicit synchronization is inserted to guarantee that loading a pair of tiles completes before being consumed in the next computation step.

In our solution, up to two thread blocks could be executed simultaneously in the same SM. Each thread computing more than 8x8 *C* elements will reduce the occupancy to one thread block per SM due to the register count limit. On the contrary, computing fewer *C* elements will transfer the bottleneck to other parts. For example, if 128x128 elements of *submatrixC* are computed by one thread block and 4x4 *C* elements per thread, it would then require 1024 threads per block. Occupancy is still two thread blocks per SM due to the device limit of 2048 threads per SM.

B. Shared Memory Data Mapping

The shared memory in GPU serves like a scratch pad. Programmer is directly responsible for all shared memory accesses. The shared memory performance is a combined effect of the number of bank conflicts, the granularity of access, and the total number of accesses. Larger granularity of access means less load instructions and higher bus bandwidth utilization. For example, loading four float values in one load instruction in the *float4* data type rather than four load instructions in the float type results in fewer load instructions. One important consideration in using shared memory is to avoid bank conflicts. When a bank conflict occurs, shared memory instructions are required to be replayed for certain threads. There won't be any shared memory bank conflicts when all threads in the same warp access the same data, because shared memory does have some broadcast capabilities. For instance, if all 32 threads access the same four bytes in a single bank, all requests can be serviced in a single cycle. The broadcast capability also extends beyond a single broadcast, such as the same value requested by eight threads within the same warp would be served in one broadcast within single cycle.

When loading *tileA* and *tileB* into shared memory, one half of the thread block (128 threads) loads *tileA* and the other half loads *tileB*. Threads are carefully mapped to elements in memory to avoid shared memory store bank conflicts. Figure 5 illustrates how to avoid both load and store bank conflicts when bringing *tileB* into shared memory. Loading and placing *tileA* is similar to *tileB*, and *tileA* is also divided similarly into 16 eight by eight microtiles. The numbers illustrated on the figure are linear thread index and they tell which thread is accessing that part of the main memory or writing that bank of shared memory. A tile in main memory is partitioned into 16 eight by eight microtiles, and each microtile is further divided into eight eight by one tracks, each of which is accessed by a different thread. In the shared memory, *tileB* is stored in a two-dimensional array data structure in which each row consists of 32 elements sitting in the 32 banks of shared memory.

Intuitively, each of the all 32 threads within a warp would load a track from a group of eight neighboring microtiles, and store the track to one bank of shared memory. However such track placement would not solve the problem of load bank conflicts. Since thread with threadId (tx,ty) will touch $microtileA_{ty}$ and $microtileB_tx$ during multiplication, a warp needs to load all 16 microtiles of *tileB*, and evenly spread them among 32 banks to get rid of load bank conflicts. Therefore, the placement of *tileB* in shared memory need to be re-arranged to avoid load bank conflicts, and the match between a thread and a track is not that intuitive.

As shown in the figure, in order to spread 16 microtiles among 32 banks, an eight by eight microtile in main memory

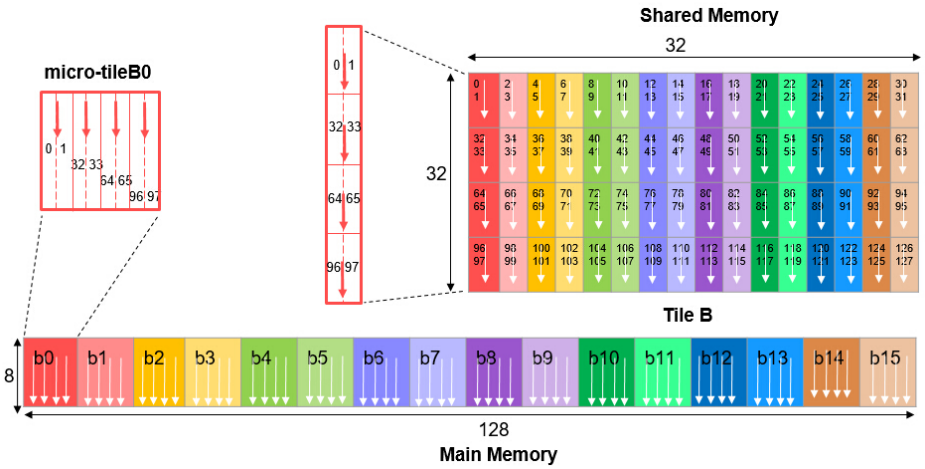


Fig. 5: Data-thread mapping when loading *tileB* into shared memory

is reconstructed as 32 by two. A warp loads 32 tracks from main memory by picking two tracks per microtile, and places them side by side in shared memory. It takes collaboration of four warps to store one microtile. For example, $microtileB_0$ is divided into four groups of tracks. Thread 0, 1 in warp 0 will store data of group 0 to location (bank 0-1, row 0-7); and thread 32, 33 belonging to warp 1 will write group 1 tracks into location (bank0-1, row 8-15), and so on. This guarantees that the 32 threads in the same warp are writing to 32 different banks in shared memory and no load bank conflicts would occur. Generally speaking, a thread will touch track $[tx \bmod 2 + 2 \times (ty \bmod \frac{blockDim.y}{2})]$ of $microtileB[\lfloor \frac{tx}{2} \rfloor]$, and store the track into bank[$tx \bmod 32$], row (8ty to 8ty+7).

C. Kernel Summation Fused with GEMM

We fuse steps of kernel summation into the GEMM framework described above. The pseudo code shown in the Algorithm 2 demonstrates an overview of the fused kernel summation routine executed by each thread block. There are seven inputs :*subA* and *subB* are 128 by K and K by 128 matrices separately; *subA*2 and *subB*2 are 128 by 128 matrices, which are submatrices of the *squareA* and *squareB* computed in the Algorithm 1; *subW* is part of the weight vector W; *subV* frames the final result V of the kernel summation problem; and two-dimensional thread block index (bx, by). The output of each thread block is vector *partialV*. A representation of (tx, ty) refers to the two-dimensional thread index. We follow the same partitioning scheme of matrices A, B, and C in the previous part. Additionally *squareA* and *squareB* are divided into sub-matrices the same way as C and the same denotation rule. Both vector W and V are evenly split into sub-vectors of dimension $blockDim.y$. The variables $sharedA_0, sharedA_1, sharedB_0, sharedB_1$ and T are declared per thread block and their sizes are the same as tileA and tileB. Matrix T is used to store thread level reduction result in shared memory. In our implementation code, T explicitly reuses the shared memory spaces of $sharedA_0$ in order to limit the amount of shared memory resources required per thread block and to increase SM occupancy. Denotation $X[i, j]$ represents the element in the i-th column and j-th row of matrix X, and $Y[k]$ represents the k-th element of vector Y. The mapping of thread blocks to the first element of their input matrices and input vectors are shown below. For example, when matrix A and B are partitioned into blocks and indexed in the same way as the one shown in Figure 4, a thread block whose index is (bx,by) will load the by-th block of matrix A as its program input subA and bx-th block of matrix B as subB.

$$\begin{aligned} \text{subA} &= A + 128 \times by \\ \text{subB} &= B + 128 \times bx \\ \text{subA2} &= squareA + N \times by + 128 \times bx \\ \text{subB2} &= squareB + N \times by + 128 \times bx \\ \text{subW} &= W + 128 \times by \\ \text{subV} &= V + 128 \times by \end{aligned}$$

Line 5-13 in Algorithm 2 are the same GEMM structure as we described in the previous part. In the function $load\text{-}nonblocking(sharedA_j, sharedB_j, tileA_i, tileB_i, tx, ty)$, each thread in the first half of thread block (i.e. $ty \leq \frac{blockDim.y}{2}$) would load a track from the $tileA_i$ into the shared memory variable $sharedA_j$, and each thread in the other half would load a track from the $tileB_i$ into the shared memory variable $sharedB_j$. The mapping of threads to tracks and data placement are already discussed before. At the end of the GEMM routine, each thread completes updating a *microtileC*, and this intermediate product is held in thread registers. The kernel evaluation according to Equation 1 becomes embarrassingly parallel. In order to make full use of the benefit brought by register locality, each thread performs kernel evaluation in the next step (line 14).

There are three levels of reductions during kernel summation: intra-thread level, intra-thread-block level, and inter-thread-block level. Synchronization needs to be carefully taken care of in the last two levels. During the intra-thread level summation, each thread performs row reduction on its eight by eight microtile, and stores the result in shared memory. The ntra-thread-block level reduction needs to wait until all threads complete its own reduction work. A thread block level synchronization function, *syncthreads()*, is called to ensure

Algorithm 2 Fused kernel summation pseudo code for each thread block

Inputs:
matrix $subA$ (128 by K), $subB$ (K by 128), $subA2$ (128 by 128), $subB2$ (128 by 128),
vector $subW$ (128 by 1), $subV$ (128 by 1), blockId (bx, by)
Outputs:
vector $partialV$ (128 by 1)
Initialize:
$j \leftarrow 0, i \leftarrow 0$, declare $sharedA_0$, $sharedB_0$, $sharedA_1$ and $sharedB_1$ as arrays in shared memory
temporal matrix T (128 by 8), $T = [\tau_0, \tau_1, ..., \tau_{127}]^T$, τ is an 8 by 1 row vector,
γ is T's 8-dimensional column vector, $\gamma_{i,j} = [T[8i, j], T[8i+1, j], ..., T[8i+7, j]]^T$
parfor each thread with threadId (tx, ty) **do**
  $load\text{-}nonblocking(sharedA_j \leftarrow tileA_i, sharedB_j \leftarrow tileB_i, tx, ty)$
  syncthreads();
  for i from 1 to $\frac{K}{8} - 1$ **do** // GEMM. $subC = subA \times subB$
    $j \leftarrow j \oplus 1$ // $\oplus$ is Exclusive OR operator
    $load\text{-}nonblocking(sharedA_j, sharedB_j, tileA_i, tileB_i, tx, ty)$ // Memory access
    $microtileC_{tx,ty} += microtileA_{ty} \times microtileB_{tx}$ // Hide memory access latency with Computation
    syncthreads()
  end for
  $microtileC_{tx,ty} += microtileA_{ty} \times microtileB_{tx}$
  $subC[tx, ty] \leftarrow \exp\{-\frac{subA2[tx,ty]+subB2[tx,ty]-2\times subC[tx,ty]}{2h^2}\}$ // Gaussian Kernel Evaluation
  // Summation
  $\gamma_{tx,ty} \leftarrow microtileC_{tx,ty} \times subW_{tx}$ // Intra-thread level reduction.
  syncthreads()
  if $ty \leq \frac{blockDim.y}{2}$ **then**
    $tid \leftarrow ty \times blockDim.x + tx$
    $partialV[tid] \leftarrow rowReduction(\tau_{tid})$ // Intra thread block level reduction
    atomicAdd($subV[tid], partialV[tid]$) // Inter thread block level reduction
  end if
end parfor

function correctness. In our case since there are 16 threads in the x-dimension of thread block, intra-thread-block level summation reduces results of every 16 threads in the row to form a partial result of that thread block. Because there are 128 rows in $subC$, only half of the thread block (i.e. 128 threads) is required to perform intra-thread-block reduction, with each thread responsible for one row. Notice that the output $partialV$ of each thread block is not the subvector of final result V. Instead $subV$ is the sum of $partialV$ distributed across thread blocks with the same by. Data communication between thread blocks is done through main memory, and requires waiting for all thread blocks to finish execution. In order to avoid synchronization latency between thread blocks and to prevent accessing memory twice to store and reload $partialV$, an atomic add operation is chosen to update $subV$ whenever $partialV$ is ready.

IV. Experimental methodology

The kernel summation application is run on a desktop system equipped with a Corei5-4960K connected to an NVIDIA GTX970 Maxwell GPU (4GB of GDDR5 video memory) over a PCIe interconnect. Technical specifications of the GTX970 are listed in Table I, and all numbers are based on the latest compute capability of 5.2. All the performance metrics and events in this work are measured with the nvprof [25] profiling tool provided by NVIDIA. The cuBLAS library used in this work is version 7.0.

Three different implementation of kernel summation problem are run and compared, which are denoted ***Fused***, ***CUDA-Unfused*** and ***cuBLAS-Unfused***. *Fused* is the kernel fusion implementation we discussed in section III. We also program two unfused version of solution. One is to pair our own SGEMM implementation with the kernel evaluation and the summation routine, denoted by *CUDA-Unfused*. Another one is to call SGEMM function provided in the cuBLAS library followed by the kernel evaluation and the summation routine, denoted by *cuBLAS-Unfused*. All runs were repeated multiple times to ensure the repeatability and consistency of the results. We test and compare different kernel summation solution with four groups of parameters. The value of dimension K is set to

32, 64, 128, and 256 in each group, and the value of dimension N is fixed to 1024 in all groups. Within each group, the value of M dimension increases from 1024 to 524288.

We show the advantage of fused kernel summation from both performance and energy perspectives. Energy model of the GPU memory is built based on CACTI [26] and McPAT [27], and the statistics are collected from the counter value reported by nvprof. We model the shared memory as an SRAM with 32 banks, each of which has separate read port and write port. In each cycle, four-byte read and four-byte write could be serviced by a bank. We derive energy per floating point unit access from McPAT. Similar to [28], Intel Xeon architecture configuration file is used and parameters are modified according to the Maxwell architecture.

V. Results and Evaluation

A. Performance

Figure 6 demonstrates normalized execution time of the *Fused* and the *CUDA-Unfused* kernel summation implementations with respect to the *cuBLAS-Unfused* implementation on primary axis, as well as the speedup of the *Fused* kernel summation versus both *cuBLAS-Unfused* and *CUDA-Unfused* on secondary axis. *Fused* approach beats *cuBLAS-Unfused* approach by up to 1.8X speedup when dimension K is not extremely large, i.e. $K < 128$. Largest speedup of 1.8X happens to the group of $K = 32$. Performance gain comes from reducing unnecessary main memory accesses. As dimension K increases the performance degradation due to our inferior CUDA-C GEMM implementation outweighs the benefits of fused computation. The speedup against *CUDA-Unfused* is a projected speedup which suggests performance benefit of fusion when a GEMM as good as the one in cuBLAS is applied in *Fused*. *Fused* shows much better performance than *CUDA-Unfused* in all problem sizes. Compared to the *CUDA-Unfused* implementation, *Fused* gains a maximum 3.7X performance speedup when $K = 32$ and around 1.5X speedup when $K = 256$. This demonstrates the benefits of fusing over an unfused implementation. It is noticed that in lower dimension scenarios, performance benefit of fusion becomes more obvious as the number of points (M,N value) increases.

Table II demonstrates the ratio of achieved operations to peak single-precision floating-point operations, which is essentially flop efficiency. Since NVIDIA profiler reports efficiency value on the granularity of kernel launched, the efficiency of *cuBLAS-Unfused* kernel summation is a weighted sum of the SGEMM kernel and the summation kernel based on their total cycle count. Higher FLOP efficiency indicates better performance. When the efficiency of fused kernel summation is lower than that of cuBLAS approach, the speedup over cuBLAS drops below 1X.

TABLE I: Configuration

Number of Multiprocessors	13
Maximum number of threads per block	1024
Warp size	32
Maximum number of resident threads per multiprocessor	2048
Number of 32-bit registers per multiprocessor	64K
Maximum number of 32-bit registers per thread	255
Maximum amount of shared memory per multiprocessor	96KB
Shared Memory Bank Size	4B
Number of shared memory banks	32
Number of warp schedulers	4
L2 size	1.75MB

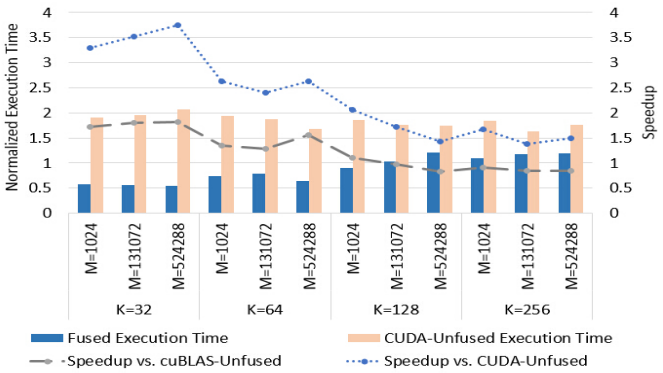


Fig. 6: Execution time and speedup of the fused kernel summation in comparison with unfused implementations.

TABLE II: FLOP Efficiency

	cuBLAS-Unfused	*Fused*
K=32		
M=1024	19.92%	33.14%
M=131072	29.30%	50.86%
M=524288	29.02%	51.05%
K=64		
M=1024	31.15%	41.86%
M=131072	45.22%	57.01%
M=524288	36.83%	56.26%
K=128		
M=1024	44.32%	49.08%
M=131072	62.15%	60.03%
M=524288	61.76%	50.29%
K=256		
M=1024	58.42%	53.75%
M=131072	74.02%	62.9%
M=524288	74.15%	62.05%

Since GEMM dominates the performance of kernel summation, a comparison between the cuBLAS GEMM and our CUDA-C GEMM would be helpful to better understand the overall performance. Figure 7 presents the normalized run time of the two GEMM implementation. As expected, the CUDA-C GEMM is two times slower than the cuBLAS GEMM. One of the main reasons causing performance deterioration is the coarse-grained control of the CUDA-C language on hardware compared with assembly. For example, it is infeasible to avoid register file bank conflict when coding in the CUDA-C programming language and the *__syncthreads()* function is the primary synchronization method between threads, which is more expensive than the low level synchronization instructions available in the Maxwell assembly. Another reason that leads

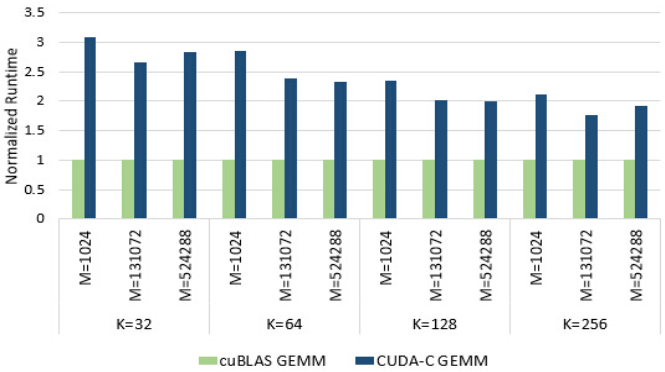


Fig. 7: Execution time comparison of different GEMM implementations

to inferior performance is that we do not optimize the part of storing results back to main memory since it is unnecessary in kernel fusion. Even though we optimize memory access ordering and rearrange data location in shared memory to avoid bank conflicts and we apply float4 type of load/store instructions as many as we can, there are still some unknown optimization schemes in the cuBLAS library that contributes better performance.

B. Influence on Memory

Analyzing the performance of *Fused* kernel summation involves discussing the trade-offs between its lower GEMM performance and reduction in the number of memory accesses. The primary effect of the proposed code optimizations is the reduction in memory transactions. Figure 8 compares the number of L2 and DRAM transactions in both *Fused* and *CUDA-Unfused* normalized with respect to *cuBLAS-Unfused*. In Figure 8a, the number of L2 transactions in the *Fused* approach is less than 50% of the *cuBLAS-Unfused* approach in most cases, except for two configurations "M=N=1024, K=128" and "M=N=1024, K=256". In higher K-dimensional scenarios CUDA-C version of SGEMM has more L2 transactions compared with the SGEMM in cuBLAS library. In configurations where product of MN is small and K value is large, the benefit of saving L2 transactions through kernel fusion is offset by additional L2 transactions in SGEMM. As shown in 8b, the number of DRAM transactions in *Fused* is less than 10% of *cuBLAS-Unfused* in all problem sizes.

C. Energy

In addition to performance benefit, fused kernel summation brings considerable energy savings thanks to reduction in main memory accesses. Table III summarizes energy savings of the *Fused* approach compared to the *cuBLAS-Unfused*. Within in a group of same dimension K, there is a trend of saving more energy when the value of dimension M increases, which is because the number of redundant memory reads and writes are $O(MN)$ in kernel summation problem. The amount of energy savings obtained from fusion is greatly affected by the K value. Up to 33% of *cuBLAS-Unfused* energy is saved when $K = 32$, and energy savings decreases as value K increases, around 8% is saved when $K = 256$.

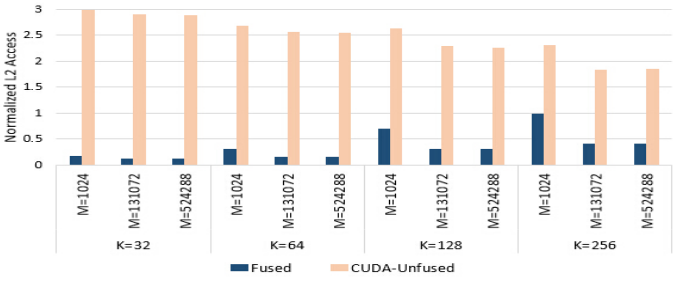
TABLE III: Energy Savings of *Fused* compared to *cuBLAS-Unfused*

	M=1024	M=131072	M=524288
K=32	31.3%	32.5%	32.5%
K=64	18.7%	23.6%	23.4%
K=128	10.2%	14.8%	13.1%
K=256	3.5%	8.5%	7.2%

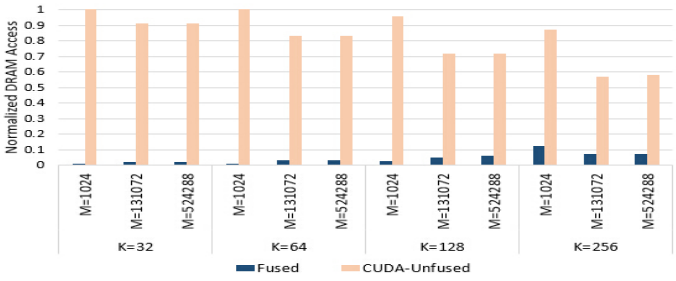
Figure 9 compares energy consumption of three different solutions, and illustrates energy breakdown into computation, shared memory, L2, and DRAM parts. Compared to the DRAM access energy in the *cuBLAS-Unfused* approach, the *Fused* approach saves more than 80% which amounts to 8% to 24% of total energy. The largest energy saving which is up to 33% happens to the group of $K = 32$. Out of 33%, DRAM access reduction contributes 26% , and the remaining 7% comes from reduction in the number of executed instructions. This is consistent with the performance speedup. When the *Fused* approach performance is better than the *cuBLAS-Unfused* approach, we get additional energy savings. In high dimension scenarios, the energy benefit from fusion is less. One reason is because DRAM access savings will balance extra energy consumption from more shared memory accesses. Another reason is because more than 80% of energy is spent on floating point computing operations such as fused multiply add.

VI. Conclusion

This paper presents a fused approach of implementing kernel summation on the state of the art GPU. Various software optimizations to improve performance and energy efficiency are implemented into the kernel summation code. Fusing series of steps in the kernel summation leads to improvement in



(a) L2 Transaction



(b) DRAM Transaction

Fig. 8: L2, DRAM transaction number normalized to *cuBLAS-Unfused*.

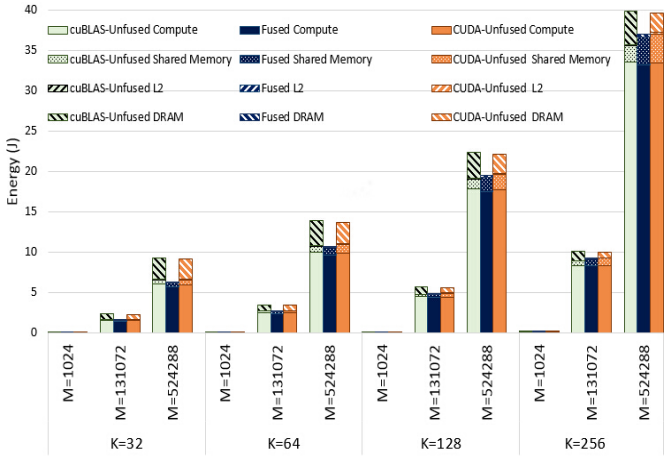


Fig. 9: Energy consumption breakdown into Compute, Shared memory, L2, and DRAM

locality and reduction of memory accesses. In addition to fusion, steps of kernel summation are optimized to increase locality by adjusting the blocking and panel sizes, and by tailoring the working set to fit in the fast on-chip memory. A major challenge in this work was the implementation of an SGEMM comparable to the cuBLAS SGEMM. Fusion is seen to improve overall performance of kernel summation up to 1.8X. We show that in lower dimensions our approach achieves higher performance compared with the approach using the cuBLAS library. Performance loss in high dimensions is due to our less efficient SGEMM. If an SGEMM as good as cuBLAS is applied, fused implementation is able to achieve up to 3.7X performance improvement. From the energy perspective, fused kernel summation shows 3% to 33% of total energy saving across various experimented dimensions. This is because eliminating redundant memory accesses via fusion results in less memory access energy. Overall our fused kernel summation is faster than the approach of calling cuBLAS library in lower dimensions. We also show that fused approach always brings energy saving benefits. This paper demonstrates optimizations at the CUDA-C level while further improvements can possibly be obtained by optimizing the code at assembly level. Steps similar to those implemented in this paper can be applied to other algorithms.

VII. Acknowledgements

This work has been supported by NSF grant CCF-1337393. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF. We would also like to thank the anonymous reviewers for their helpful suggestions to improve the paper.

References

[1] M. Intel, "Intel math kernel library," 2007.

[2] C. Nvidia, "Cublas library," *NVIDIA Corporation, Santa Clara, California*, vol. 15, p. 27, 2008.

[3] C. Isen and L. John, "Eskimo-energy savings using semantic knowledge of inconsequential memory occupancy for dram subsystem," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 337–346, IEEE, 2009.

[4] K. T. Malladi, B. C. Lee, F. A. Nothaft, C. Kozyrakis, K. Periyathambi, and M. Horowitz, "Towards energy-proportional datacenter memory with mobile dram," in *ACM SIGARCH Computer Architecture News*, vol. 40, pp. 37–48, IEEE Computer Society, 2012.

[5] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller, "Energy management for commercial servers," *Computer*, vol. 36, no. 12, pp. 39–48, 2003.

[6] A. G. Gray and A. W. Moore, "N-body'problems in statistical learning," in *NIPS*, vol. 4, pp. 521–527, Citeseer, 2000.

[7] T. Hofmann, B. Schölkopf, and A. J. Smola, "Kernel methods in machine learning," *The annals of statistics*, pp. 1171–1220, 2008.

[8] S. Mika, B. Schölkopf, A. J. Smola, K.-R. Müller, M. Scholz, and G. Rätsch, "Kernel pca and de-noising in feature spaces.," in *NIPS*, vol. 4, p. 7, Citeseer, 1998.

[9] B. Schölkopf and A. J. Smola, *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press, 2002.

[10] J. A. Suykens and J. Vandewalle, "Least squares support vector machine classifiers," *Neural processing letters*, vol. 9, no. 3, pp. 293–300, 1999.

[11] Y. Anzai, *Pattern Recognition & Machine Learning*. Elsevier, 2012.

[12] W. B. March, B. Xiao, C. D. Yu, and G. Biros, "An algebraic parallel treecode in arbitrary dimensions," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pp. 571–580, IEEE, 2015.

[13] J. Bédorf, E. Gaburov, and S. P. Zwart, "A sparse octree gravitational n-body code that runs entirely on the gpu processor," *Journal of Computational Physics*, vol. 231, no. 7, pp. 2825–2839, 2012.

[14] H. Cheng, L. Greengard, and V. Rokhlin, "A fast adaptive multipole algorithm in three dimensions," *Journal of computational physics*, vol. 155, no. 2, pp. 468–498, 1999.

[15] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros, "A massively parallel adaptive fast multipole method on heterogeneous architectures," *Communications of the ACM*, vol. 55, no. 5, pp. 101–109, 2012.

[16] I. F. Sbalzarini, J. H. Walther, M. Bergdorf, S. E. Hieber, E. M. Kotsalis, and P. Koumoutsakos, "Ppm–a highly efficient parallel particle–mesh library for the simulation of continuum systems," *Journal of Computational Physics*, vol. 215, no. 2, pp. 566–588, 2006.

[17] T. Darden, D. York, and L. Pedersen, "Particle mesh ewald: An n log (n) method for ewald sums in large systems," *The Journal of chemical physics*, vol. 98, no. 12, pp. 10089–10092, 1993.

[18] D. Y. Chenhan, J. Huang, W. Austin, B. Xiao, and G. Biros, "Performance optimization for the k nearest-neighbor kernel on x86 architectures," 2015.

[19] N. Nakasato, "A fast gemm implementation on the cypress gpu," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 50–55, 2011.

[20] R. Nath, S. Tomov, and J. Dongarra, "An improved magma gemm for fermi graphics processing units," *International Journal of High Performance Computing Applications*, vol. 24, no. 4, pp. 511–515, 2010.

[21] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun, "Fast implementation of dgemm on fermi gpu," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 35, ACM, 2011.

[22] "Maxas." https://github.com/NervanaSystems/maxas.

[23] "Fermi compute architecture whitepaper." http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf.

[24] "Tuning cuda applications for maxwell." http://docs.nvidia.com/cuda/maxwell-tuning-guide/#axzz3op9EeX3M.

[25] "Profiler user's guide." http://docs.NVIDIA.com/cuda/profiler-users-guide/#axzz3oNg3mHRn.

[26] P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An integrated cache timing, power, and area model," tech. rep., Technical Report 2001/2, Compaq Computer Corporation, 2001.

[27] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 469–480, ACM, 2009.

[28] J. Lim, N. B. Lakshminarayana, H. Kim, W. Song, S. Yalamanchili, and W. Sung, "Power modeling for gpu architectures using mcpat," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 19, no. 3, p. 26, 2014.