Chapter 1

GPU Matrix Multiplication

Junjie Li

jl3@cise.ufl.edu, Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611

Sanjay Ranka

ranka@cise.ufl.edu, Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611

Sartaj Sahni

sahni@cise.ufl.edu, Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611

1

[1]This work was supported, in part, by the National Science Foundation under grants CNS0829916, CNS0905308, CCF0903430, NETS0963812 and NETS1115194.

1.1 Introduction

Graphics Processing Units (GPUs) were developed originally to meet the computational needs of algorithms for rendering computer graphics. The rapid and enormous growth in sophistication of graphics applications such as computer games has resulted in the availability of GPUs that have hundreds of processors and peak performance near a teraflop and that sell for hundreds of dollars to a few thousand dollars. Although GPUs are optimized for graphics calculations, their low cost per gigaflop has motivated significant research into their efficient use for non-graphics applications. The effort being expended in this direction has long-lasting potential because the widespread use of GPUs in the vibrant computer games industry almost ensures the longevity of GPUs. So, unlike traditional multimillion dollar supercomputers whose development cost had to be borne entirely by a relatively small supercomputing community, GPUs are backed by a very large gaming industry. This makes it more likely that GPU architectures will remain economically viable and will continue to evolve.

Although the cost of a GPU measured as dollars per peak gigaflop is very low, obtaining performance near the peak requires very careful programming of the application code. This programming is complicated by the availability of several different memories (e.g., device memory, shared memory, constant cache, texture cache, and registers), each with different latency and the partitioning of the available scalar processors or cores into groups called streaming multiprocessors (Figure 1.1). In this chapter, we explore the intricacies of programming a GPU to obtain high performance for the multiplication of two single-precision square matrices. We focus our development to NVIDIA's Tesla series of GPUs of which the C1060 is an example (Figure 1.2). Our example programs are developed using CUDA.

1.2 GPU Architecture

NVIDIA's Tesla C1060 GPU, Figure 1.2, is an example of NVIDIA's general purpose parallel computing architecture CUDA (Compute Unified Driver Architecture) [10]. Figure 1.2 is a simplified version of Figure 1.1 with $N = 30$ and $M = 8$. The C1060 comprises 30 streaming multiprocessors (SMs) and each SM comprises 8 scalar processors (SPs), 16KB of on-chip shared memory, and 16,384 32-bit registers. Each SP has its own integer and single-precision floating point units. Each SM has 1 double-precision floating-point unit and 2 single-precision transcendental function (special function, SF) units that are shared by the 8 SPs in the SM. The 240 SPs of a Tesla C1060 share 4GB of off-chip memory referred to as *device* or *global* memory [5]. A C1060 has a peak performance of 933 GFlops of single-precision floating-point operations and 78 GFlops of double-precision operations. The peak of 933GFlops is for the case when Multiply-Add (MADD) instructions are dual issued with special function (SF) instructions. In the absence of SF instructions, the peak is 622GFlops (MADDs only) [6]. The C1060 consumes 188W of power. The architecture of the NVIDIA Tesla C2050 (also known as Fermi) corresponds to Figure 1.1 with $N = 14$ and $M = 32$. So, a C2050 has 14 SMs and each SM has 32 SPs giving the C2050 a total of 448 SPs or cores. Although each SP of a C2050 has its own integer, single- and double-precision units, the 32 SPs of an SM share 4 single-precision transcendental function units. An SM has 64KB of on-chip memory that can be "*configured as 48KB of shared memory with 16KB of L1 cache (default setting) or as 16KB of shared memory with 48KB of L1 cache*" [5]. Additionally,

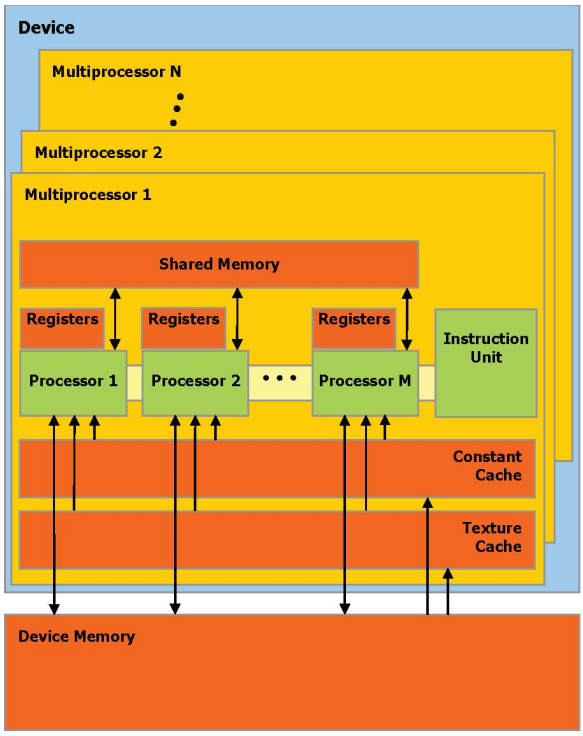

FIGURE 1.1: NVIDIA's GPU hardware model [5]

there are 32K 32-bit registers per SM and 3GB of off-chip device/global memory that is shared by all 14 SMs. The peak performance of a C2050 is 1,288 GFlops (or 1.288TFlops) of single-precision operations and 515GFlops of double-precision operations and the power consumption is 238W [7]. Once again, the peak of 1,288GFlops requires that MADDs and SF instructions be dual issued. When there are MADDs alone, the peak single-precision rate is 1.03GFlops. Notice that the ratio of power consumption to peak single-precision GFlop rate is 0.2W/GFlop for the C1060 and 0.18W/GFlop for the C2050. The corresponding ratio for double-precision operations is 2.4W/GFlops for the C1060 and 0.46W/GFlop for the C2050. In NVIDIA parlance, the C1060 has compute capability 1.3 while the compute capability of the C2050 is 2.0.

A Tesla GPU is packaged as a double-wide PCIe card (Figure 1.3) and using an appro-

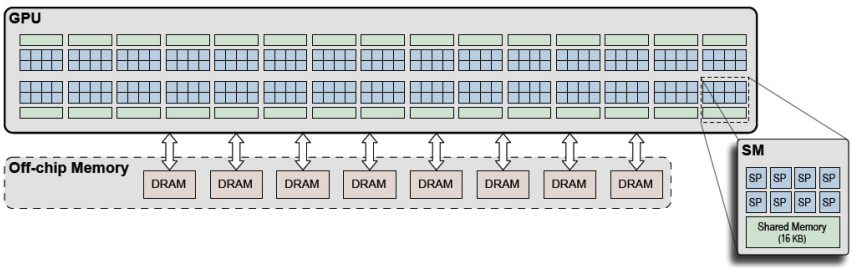

FIGURE 1.2: NVIDIA's Tesla C1060 GPU [10]

priate motherboard and a sufficiently large power supply, one can install up to 4 GPUs on the same motherboard. In this chapter, we focus on single GPU computation.

FIGURE 1.3: NVIDIA's Tesla PCIex16 Card (www.nvidia.com)

1.3 Programming Model

At a high-level, a GPU uses the master-slave programming model [12] in which the GPU operates as a slave under the control of a master or host processor. In our experimental set up, the master or host is a 2.8GHz Xeon quad core processor and the GPU is the NVIDIA Tesla C1060. Programming in the master-slave model requires us to write a program that runs on the master processor (in our case the Xeon). This master program sends data to the slave(s) (in our case a single C1060 GPU), invokes a kernel or function that runs on the slave(s) and processes this sent data, and finally receives the results back from the slave. This process of sending data to the slave, executing a slave kernel, and receiving the computed results may be repeated several times by the master program. In CUDA, the host/master and GPU/slave codes may be written in C. CUDA provides extensions to C to allow for data transfer to/from device memory and for kernel/slave code to access registers, shared memory, and device memory.

At another level, GPUs use the SIMT (single instruction multiple thread) programming model in which the GPU accomplishes a computational task using thousands of light weight threads. The threads are grouped into blocks and the blocks are organized as a grid. While a block of threads may be 1-, 2-, or 3-dimensional, the grid of blocks may only be 1- or 2-dimensional. Kernel invocation requires the specification of the block and grid dimensions along with any parameters the kernel may have. This is illustrated below for a matrix multiply kernel $MatrixMultiply$ that has the parameters a, b, c, and n, where a, b, and c are pointers to the start of the row-major representation of $n \times n$ matrices and the kernel computes $c = a * b$.

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

A GPU has a block scheduler that dynamically assigns thread blocks to SMs. Since all the threads of a thread block are assigned to the same SM, the threads of a block may communicate with one another via the shared memory of an SM. Further, the resources

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

Once a block is assigned to an SM, its threads are scheduled to execute on the SM's SPs by the SM's warp scheduler. The warp scheduler divides the threads of the blocks assigned to an SM into warps of 32 consecutively indexed threads from the same block. Multidimensional thread indexes are serialized in row-major order for partitioning into warps. So, a block that has 128 threads is partitioned into 4 warps. Every thread currently assigned to an SM has its own instruction counter and set of registers. The warp scheduler selects a warp of ready threads for execution. If the instruction counters for all threads in the selected warp are the same, all 32 threads execute in 1 step. On a GPU with compute capability 2.0, each SM has 32 cores and so all 32 threads can perform their common instruction in parallel, provided, of course, this common instruction is an integer or floating point operation. On a GPU with compute capability 1.3, each SM has 8 SPs and so a warp can execute the common instruction for only 8 threads in parallel. Hence, when the compute capability is 1.3, the GPU takes 4 rounds of parallel execution to execute the common instruction for all 32 threads of a warp. When the instruction counters of the threads of a warp are not the same, the GPU executes the different instructions serially. Note that the instruction counters may become different as the result of "*data dependent conditional branches*" in a kernel [5].

An SM's warp scheduler is able to hide much of the 400 to 600 cycle latency of a device-memory access by executing warps that are ready to do arithmetics while other warps wait for device-memory accesses to complete. So, the performance of code that makes many accesses to device memory can often be improved by optimizing it to increase the number of warps scheduled on an SM (i.e., increase the multiprocessor occupancy, see Section 1.4). This optimization could involve increasing the number of threads per block and/or reducing the shared memory and register utilization of a block to enable the scheduling of a larger number of blocks on an SM.

1.4 Occupancy

The *occupancy* of a multiprocessor (SM) is defined to be the ratio of the number of warps co-scheduled on a multiprocessor and the GPU's limit, $maxWarpsPerSM$, on the number of warps that may be co-scheduled on a multiprocessor. So,

$$occupancy = \frac{\text{number of warps co-scheduled per SM}}{maxWarpsPerSM} \tag{1.1}$$

In practice, the number of warps co-scheduled on a multiprocessor is determined by several factors. These factors may be broadly classified as those that are characteristics of the GPU and those that specify resource requirements of the application kernel. Therefore, for a given GPU, the occupancy of a multiprocessor varies from one kernel to the next and even when the GPU and kernel are fixed, the occupancy varies with the block size (i.e., number of threads per block) with which the kernel is invoked. Further, the same kernel and block size may result in different occupancy values on (say) a Tesla C1060 and a Tesla

C2050 GPU. Since, *applications with higher occupancy are better able to hide the latency of the GPU global/device memory*, increasing the occupancy of an application is one of the strategies used to improve performance.

In describing the factors that contribute to the occupancy of a multiprocessor, we use some of the variable names used in the CUDA occupancy calculator [2]. The GPU factors depend on the compute capability of the GPU.

1. GPU Factors
   (a) The maximum number, $maxThreadsPerWarp$, of threads in a warp.
   (b) The maximum number of blocks, $maxBlocksPerSM$, that may be co-scheduled on a multiprocessor.
   (c) Warp allocation granularity, $warpAllocationUnit$. For purposes of allocating registers to a block, the number of warps in a block is rounded up (if necessary) to make it a multiple of the warp allocation granularity. So, for example, when $warpAllocationUnit = 2$ and the number of warps in a block is 7, the number of registers assigned to the block is that for 8 warps.
   (d) Registers are allocated to a warp (compute capability 2.0) or block (compute capability 1.3) in units of size $regUnit$. So, for example, on a GPU with compute capability 2.0 and $regUnit = 64$, a warp that requires 96 registers would be allocated $\lceil 96/64 \rceil * 64 = 2 * 64 = 128$ registers. A block of 6 such warps would be assigned $6 * 128 = 768$ registers. The same block on a GPU with compute capability 1.3 and $regUnit = 512$ would be allocated $\lceil 6 * 96/512 \rceil * 512 = 1024$ registers.
   (e) Shared memory granularity, $sharedMemUnit$. Shared memory is allocated in units whose size equals $sharedMemUnit$. So, when $sharedMemUnit = 128$, a block that requires 900 bytes of shared memory is allocated $\lceil 900/128 \rceil * 128 = 8 * 128 = 1024$ bytes of shared memory.
   (f) Number of registers, $numReg$, available per multiprocessor.
   (g) Shared memory, $sharedMemSize$, available per multiprocessor.

   In addition to the explicit limit $maxBlocksPerSM$, the number of blocks that may be co-scheduled on a multiprocessor is constrained by the limit, $maxWarpsPerSM$, on the number of warps as well as by the number of registers and the size of the shared memory. Figure 1.4 gives the values of the GPU specific factors for GPUs with compute capabilities 1.3 (e.g., Tesla C1060) and 2.0 (e.g., Tesla C2050). From these values, we see that at most 8 blocks may be co-scheduled on a multiprocessor of a GPU with compute capability 1.3 or 2.0. Further, if a block comprises 10 warps, the number of blocks that may be co-scheduled on a multiprocessor is reduced from 8 to $\lfloor 32/10 \rfloor = 3$ when the compute capability is 1.3 and to $\lfloor 48/10 \rfloor = 4$ when the compute capability is 2.0.

2. Kernel Factors
   (a) Number, $myThreadsPerBlock$, of threads in a block.
   (b) Number, $myRegPerThread$, of registers required per thread.
   (c) Amount, $mySharedMemPerBlock$, of shared memory required per block.

To determine the occupancy of a kernel, we need to determine the number, WPM, of warps that will be co-scheduled on a multiprocessor (Equation 1.1). This number is the

variable↓/compute capability→	1.3	2.0
$maxBlocksPerSM$	8	8
$maxThreadsPerWarp$	32	32
$maxWarpsPerSM$	32	48
$numReg$	16,384	32,768
$regUnit$	512	64
$sharedMemUnit$	512	128
$sharedMemSize$	16,384	49,152
$warpAllocationUnit$	2	1

FIGURE 1.4: GPU constraints for compute capabilities 1.3 and 2.0 [2]

product of the number, BPM, of blocks that get co-scheduled on a multiprocessor and the number, WPB, of the warps per block. From the definition of the GPU and kernel factors that contribute to occupancy, we see that:

$$WPB = \left\lceil \frac{myThreadsPerBlock}{maxThreadsPerWarp} \right\rceil \tag{1.2}$$

$$\begin{aligned} BPM = \min\{&maxBlocksPerSM, \\ &\left\lfloor \frac{maxWarpsPerSM}{WPB} \right\rfloor, \\ &\left\lfloor \frac{numReg}{myRegPerBlock} \right\rfloor, \\ &\left\lfloor \frac{sharedMemSize}{myMemPerBlock} \right\rfloor\} \end{aligned} \tag{1.3}$$

where

$$myRegPerBlock = \begin{cases} \left\lceil \frac{GWPB*myRegPerThread*maxThreadsPerWarp}{regUnit} \right\rceil * regUnit \\ \text{(capability 1.3)} \\ \\ \left\lceil \frac{myRegPerThread*maxThreadsPerWarp}{regUnit} \right\rceil * regUnit * WPB \\ \text{(capability 2.0)} \end{cases} \tag{1.4}$$

$$GWPB = \lceil WPB/warpAllocationUnit \rceil * warpAllocationUnit \tag{1.5}$$

$$myMemPerBlock = \left\lceil \frac{mySharedMemPerBlock}{sharedMemUnit} \right\rceil * sharedMemUnit \tag{1.6}$$

As an example, consider a kernel that requires 20 registers per thread and 3000 bytes of shared memory. Assume that the kernel is invoked on a C1060 (compute capability 1.3) with a block size of 128. So, $myThreadsPerBlock = 128$, $myRegPerThread = 20$, and

$mySharedMemPerBlock = 3000$. From Equations 1.2-1.6, we obtain:

$$\begin{aligned} WPB &= \lceil 128/32 \rceil = 4 \\ GWPB &= \lceil 4/2 \rceil * 2 = 4 \\ myRegPerBlock &= \lceil 4 * 20 * 32/512 \rceil * 512 = 2560 \\ myMemPerBlock &= \lceil 3000/512 \rceil * 512 = 3072 \\ BPM &= \min\{8, \lfloor 32/4 \rfloor, \lfloor 16384/2560 \rfloor, \lfloor 16384/3072 \rfloor\} \\ &= \min\{8, 8, 6, 5\} = 5 \end{aligned}$$

For our example kernel, we see that number of blocks that may be co-scheduled on a multiprocessor is limited by the available shared memory. From Equation 1.1, we get:

$$occupancy = \frac{BPM * WPB}{maxWarpsPerSM} = 5 * 4/32 = 0.63$$

Since, $BPM \leq 8$ for the C1060 (Equation 1.3) and $WPB = 4$ for our example kernel, optimizing our example kernel to use less shared memory and fewer registers could raise occupancy to $8 * 4/32 = 1.0$. So, if we reduced the shared memory requirement of a block to 1000 bytes while keeping register usage the same,

$$BPM = \min\{8, 8, 6, \lfloor 16384/1024 \rfloor\} = \min\{8, 8, 6, 16\} = 6$$

and $occupancy = 6 * 4/32 = 0.75$. Now, the occupancy is limited by the number of registers. Note that if we reduce the number of threads per block to 64, $WPB = 2$ and $occupancy \leq 8 * 2/32 = 0.5$.

1.5 Single Core Matrix Multiply

Figure 1.5 gives the classical textbook algorithm to multiply two $n \times n$ matrices A and B stored in row-major order in the one-dimensional arrays a and b. The result matrix $C = A * B$ is returned, also in row-major order, in the one-dimensional array c. This classical algorithm is referred to as the ijk algorithm because the three nested `for` loops are in this order. It is well known that computing the product of two matrices in ikj order as in Figure 1.6 (see [9], for example) reduces cache misses and so results in better performance when n is large.

Figure 1.7 gives the run times for the ijk and ikj versions on our Xeon quad-core processor. As expected, the ijk version is faster for small n because of its smaller overhead. But, for large n, the cache effectiveness of the ikj version enables it to outperform the ijk version. For $n = 4096$, for example, the ijk version takes almost 7 times as much time as taken by the ikj version! In computing the ratio ijk/ikj, we have used the measured run times with 3 decimal digits of precision. This explains the slight difference in the ratios reported in Figure 1.7 and what you would get using the two-decimal-digit times given in this figure. In the remainder of this paper also, we computed derived results such as speedup using data with 3-digit precision even though reported data have 2-digit precision.

```
void SingleCoreIJK (float *a, float *b, float *c, int n)
{// Single core algorithm to multiply two
  // n x n row-major matrices a and b.
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
      {
         float temp = 0;
         for (int k = 0; k < n; k++)
            temp += a[i*n+k]*b[k*n+j];
         c[i*n+j] = temp;
      }
}
```

FIGURE 1.5: Single core matrix multiply

```
void SingleCoreIKJ (float *a, float *b, float *c, int n)
{// single core cache aware algorithm to multiply two
 // n x n row-major matrices a and b
   for (int i = 0; i < n; i++)
   {
      for (int j = 0; j < n; j++)
         c[i*n+j] = 0;

      for (int k = 0; k < n; k++)
         for (int j = 0; j < n; j++)
            c[i*n+j] += a[i*n+k]*b[k*n+j];
   }
}
```

FIGURE 1.6: Single core cache aware matrix multiply

	n				
Method	256	512	1024	2048	4096
ijk	0.11	0.93	10.41	449.90	4026.17
ikj	0.14	1.12	8.98	72.69	581.28
ijk/ikj	0.80	0.83	1.16	6.19	6.93

FIGURE 1.7: Run times (seconds) for the ijk and ikj versions of matrix multiply

```
void MultiCore (float *a, float *b, float *c, int n, int t)
{// Multicore algorithm to multiply two
 // n x n row-major matrices a and b.
 // t is the number of threads.
    #pragma omp parallel shared (a, b, c, n, t)
    {
       int tid = omp_get_thread_num(); // thread ID
       multiply(a, b, c, n, t, tid);
    }
}

void multiply(float *a, float *b, float *c, int n, int t, int tid)
{// compute an (n/t) x n sub-matrix of c; t is the number of threads
 // tid, 0 <= tid < t is the thread ID
    int height = n/t;
    int offset = height*n*tid;

    a += offset;
    c += offset;

    for (int i = 0; i < height; i++)
    {
       for (int j = 0; j < n; j++)
          c[i*n+j] = 0;

       for (int k = 0; k < n; k++)
           for (int j = 0; j < n; j++)
              c[i*n+j] += a[i*n+k]*b[k*n+j];
    }
}
```

FIGURE 1.8: Multicore matrix multiply using OpenMP

1.6 Multicore Matrix Multiply

A multicore version of the cache efficient single-core algorithm $SingleCoreIKJ$ is obtained quite easily using OpenMP [8]. When t threads are used, each thread computes an $(n/t) \times n$ sub-matrix of the result matrix C using the algorithm *multiply* (Figure 1.8). The similarity between *multiply* and $SingleCoreIKJ$ is apparent.

Figure 1.9 gives the time taken by $MultiCore$ to multiply $n \times n$ matrices for various values of n using 1, 2, 4, and 8 threads on our Xeon quad core processor. We note that, as expected, when $t = 1$, the run times for $MultiCore$ and $SingleCoreIKJ$ are virtually the same. As we did for Figure 1.7, in computing the speedups reported in Figure 1.9, we have used the measured run times with 3 decimal digits of precision. This explains the slight difference in the reported speedups and what you would get using the two-decimal-digit times given in Figure 1.9. The speedup for $t = 2$, 4, and 8 relative to the case $t = 1$, is given in the last three lines of Figure 1.9. The speedup relative to $t = 1$ is virtually the same as that relative to $SingleCoreIKJ$. It is interesting to note that the simple parallelization

	n				
t	256	512	1024	2048	4096
1	0.14	1.12	8.98	72.67	581.24
2	0.07	0.56	4.50	36.39	291.14
4	0.04	0.28	2.26	18.21	146.37
8	0.04	0.28	2.28	18.42	146.98
$T1/T2$	2.00	2.00	2.00	2.00	2.00
$T1/T4$	4.00	3.97	3.98	3.99	3.97
$T1/T8$	3.89	4.00	3.94	3.95	3.95

t is the number of threads
Ti is the time using i threads, $i \in \{1, 2, 4, 8\}$

FIGURE 1.9: Run times (seconds) for $MultiCore$

done to arrive at $MultiCore$ achieves a speedup of 2.00 when $t = 2$ and a speedup very close to 4.00 when $t = 4$. Further increase in the number of threads does not result in a larger speedup as we have only 4 cores to execute these threads on. Actually, when we go from $t = 4$ to $t = 8$, there is a very small drop in performance for most values of n because of the overhead of scheduling 8 threads rather than 4.

1.7 GPU Matrix Multiply

Although $SingleCoreIKJ$ (Figure 1.6) is faster than $SingleCoreIJK$ when n is large, we do not expect a massively threaded adaptation of $SingleCoreIKJ$ to outperform a similarly threaded version of $SingleCoreIJK$ because $SingleCoreIKJ$ makes $O(n^3)$ accesses to c while the $SingleCoreIJK$ makes $O(n^2)$ accesses and c resides in device memory. (Note that both versions make $O(n^3)$ accesses to a and b.) So, we focus on obtaining a GPU kernel based on $SingleCoreIJK$.

We begin by tiling an $n \times n$ matrix using $p \times q$ tiles as in Figure 1.10. For simplicity, we assume that p and q divide n so that an integral number of tiles is needed to cover the matrix. We may index the tiles using a 2-dimensional index (u, v), $0 \leq u < n/q$, $0 \leq v < n/p$. We adopt the convention that the top left tile is indexed (0,0), the first coordinate of an index increases left to right, and the second coordinate increases top to bottom. Our GPU code will use a block of threads to compute a tile (more accurately, the sub-matrix corresponding to a tile) of the result matrix C. To compute the entire matrix C, we will use an $n/p \times n/q$ grid of thread blocks with thread block (u, v) computing tile (u, v) of C. In CUDA, a thread may determine the coordinates of the block (u, v) that it is part of using the variables $blockIdx.x$ and $blockIdx.y$. So, $(u, v) = (blockIdx.x, blockIdx.y)$.

Several different GPU adaptations of $SingleCoreIJK$ are possible. These differ in their usage of registers, shared memory, number of C elements computed per thread, and so on. Different adaptations result in different performance. In the following, we examine some of the possible adaptations.

(0, 0)	(1, 0)	(2, 0)	(3, 0)
(0, 1)	(1, 1)	(2, 1)	(3, 1)
(0, 2)	(1, 2)	(2, 2)	(3, 2)
(0, 3)	(1, 3)	(2, 3)	(3, 3)
(0, 4)	(1, 4)	(2, 4)	(3, 4)
(0, 5)	(1, 5)	(2, 5)	(3, 5)
(0, 6)	(1, 6)	(2, 6)	(3, 6)
(0, 7)	(1, 7)	(2, 7)	(3, 7)

FIGURE 1.10: Tiling a 16×16 matrix using 32 2×4 tiles

1.7.1 A Thread Computes a 1×1 Sub-matrix of C

Kernel Code

In our first adaptation of $SingleCoreIJK$, each thread computes exactly one element of a tile. So, the number of threads in a block equals the number of elements $p * q$ in a tile. Threads of a block are indexed using the same convention as used to index blocks. That is, thread (0,0) computes the top left element of a tile, the first coordinate increases left to right, and the second coordinate increases top to bottom. With this convention, the index of a thread within a block is $(threadIdx.x, threadIdx.y)$. Notice that $threadIdx.x$ is in the range $[0, q)$ and $threadIdx.y$ is in the range $[0, p)$. A CUDA kernel may determine the dimensions of a thread block using the variables, $blockDim.x$ and $blockDim.y$. For our example, $p = blockDim.y$ and $q = blockDim.x$.

The convention to index elements of a matrix is different from that used to index blocks in a grid and threads in a block. $C(i, j)$ refers to element (i, j) of C and i increases top to bottom while j increases left to right; $C(0, 0)$ is the top-left element. Because of this difference in indexing convention, overlaying a sub-matrix tile with a thread block results in thread (d, e) corresponding to element (e, d) of the tile/sub-matrix.

Figure 1.11 illustrates the work done by the thread that computes $C(5, 6)$ of an 8×8 matrix C. Matrix C is tiled using 2×4 tiles. The total number of tiles is 8. The sub-matrix defined by a tile is computed by a block of threads. The dimensions of a block are $(4, 2)$. That is, $blockDim.x = 4$ and $blockDim.y = 2$. The dimensions of the grid of blocks are $(2, 4)$. $C(5, 6)$ is in tile/block $(1, 2)$ of the grid and it is computed by thread $(2, 1)$ (i.e., $threadIdx.x = 2$ and $threadIdx.y = 1$) of the block. When writing the code, we have to do the inverse computation. That is, a thread has to determine which element, $C(i, j)$, of the C matrix it is to compute. A thread does this inverse computation using its index as well as the index of the block it is part of. To compute $C(5, 6)$, thread $(2, 1)$ of block $(1, 2)$ reads the pairs $(A(5, k), B(k, 6))$, one pair at a time, multiplies the two components of the pair, and adds to a variable, $temp$, whose final value will be $C(5, 6)$. $A(5, 0)$ and $B(0, 6)$ are

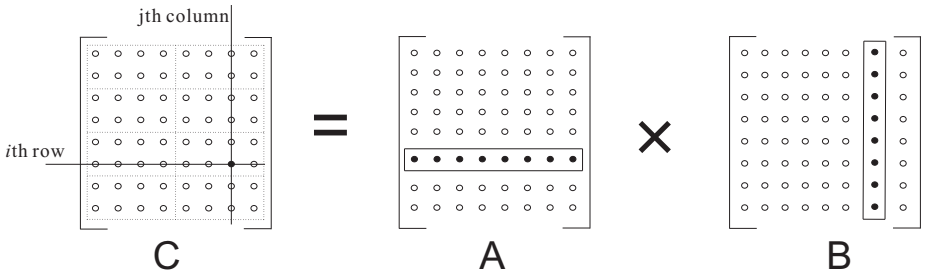

FIGURE 1.11: Thread (2,1) of block (1,2) computes $C(5,6)$

```
__global__ void GPU1 (float *a, float *b, float *c, int n)
{// thread code to compute one element of c
   // element (i,j) of the product matrix is to be computed
   int i = blockIdx.y * blockDim.y + threadIdx.y;
   int j = blockIdx.x * blockDim.x + threadIdx.x;

   float temp = 0;
   for (int k = 0; k < n; k++)
      temp += a[i*n+k]*b[k*n+j];
   c[i*n+j] = temp;
}
```

FIGURE 1.12: Thread code to compute one element of c

located in their row-major representations a and b using the standard row-major formula (i.e., $X(u,v)$ of and $n \times n$ matrix X is at position $u * n + v$ in its row-major representation x).

Figure 1.12 gives the kernel code for our first adaptation, $GPU1$, of $SingleCoreIJK$. This code defines the computation done by a single thread. The first two lines determine the matrix element (i,j) to be computed by the thread. The remaining lines of code in this figure compute $C(i,j) = c[i*n+j]$. The variables i, j, k, and $temp$ are assigned to registers while a, b, and c are arrays that reside in device memory.

Notice the similarity between the kernel code of Figure 1.12 and the single-core code of Figure 1.5. The total number of threads employed to compute the product of two $n \times n$ matrices is n^2 and these threads are partitioned into $n^2/(pq)$ blocks of pq threads each. The global block scheduler assigns these blocks to the SMs for execution in some unspecified order and, resources permitting, several blocks may be co-scheduled on an SM. The warp scheduler of an SM schedules the threads of the assigned blocks for execution in warps of 32 threads; each block has $\lceil pq/32 \rceil$ warps.

We note that $GPU1$ is identical to $MatMulKernel$ that is on page 22 of [5].

Host Code

Figure 1.13 gives a code fragment that could be used to compute the product of two $n \times n$ matrices using $GPU1$. In this fragment, d_A, d_B, and d_C refer to (previously allocated) device memory for the matrices A, B, and C while h_A, h_B, and h_C refer to host memory for the same matrices. The code fragment begins by copying the matrices A and B from

```
//copy from host to device
cudaMemcpy (d_A, h_A, memSize, cudaMemcpyHostToDevice);
cudaMemcpy (d_B, h_B, memSize, cudaMemcpyHostToDevice);

// define grid and block dimensions
dim3 grid (n/q, n/p);
dim3 block (q, p);

// invoke kernel
GPU1 <<<grid, block>>> (d_A, d_B, d_C, n);

// wait for all threads to complete
cudaThreadSynchronize ();

// copy result matrix from device to host
cudaMemcpy (h_C, d_C, memSize, cudaMemcpyDeviceToHost);
```

FIGURE 1.13: Fragment to invoke `GPU1`

the host to the device. Then, the grid and block dimensions are defined and the kernel that defines a thread computation invoked. Following the invocation of this kernel, we wait for all threads to complete and then copy the computed matrix product back to the host.

Tile/Block Dimensions

Before we can run our code on a GPU, we must decide the tile dimensions p and q, which in turn determine the block dimensions (q, p)[2]. Since NVIDIA GPUs limit the number of threads in a block to be no more than 512, $pq \leq 512$. Since threads are scheduled in units of a warp and a warp is 32 threads, we would like pq to be a multiple of 32. So, when using a square block, the only possible choices for the block dimension are: 8×8 and 16×16. When using a rectangular block, additional choices (e.g., 1×64, 128×2, 8×64) become available. Some of these block sizes may be infeasible because they require more resources (e.g., registers and shared memory) than are available on an SM. An instrumentation of the kernel $GPU1$ reveals that it needs 10 registers per thread and 44 bytes of shared memory (this includes registers and shared memory required for any compiler generated temporary variables and shared memory used to store the values of kernel parameters)[3]. So, the maximum block size (i.e., pq) for $GPU1$ is not constrained by the number of available registers or by the size of shared memory but only by the NVIDIA imposed limit of 512 threads per block. Using too small a block size will result in an occupancy less than 1 (Section 1.4). For example when the block size is 64 (say an 8×8 block is used), the number of blocks co-scheduled on an SM is 8 (this is the value of $maxBlocksPerSM$, Figure 1.4). So, only 16 warps are co-scheduled on an SM and the occupancy is $16/32 = 0.5$ for compute capability 1.3 and $16/48 = 0.33$ for compute capability 2.0 (Equation 1.1).

[2]By convention, the dimensions of a $p \times q$ block of threads are specified as (q, p)

[3]The register and shared memory utilization as well as the occupancy of our GPU codes reported in this chapter were obtained by using the `ptx` option to the CUDA compiler

(x, y)	occupancy	n			
		2048	4096	8192	16384
(1,32)	0.25	5.94	44.13	397.46	4396.39
(1,64)	0.50	5.62	43.69	441.52	6026.91
(1,128)	1.00	6.67	51.15	525.42	7341.49
(1,256)	1.00	6.53	51.80	594.71	8612.61
(1,512)	1.00	6.97	55.55	662.66	9373.16
(8,8)	0.50	1.18	7.95	64.49	540.03
(16,16)	1.00	0.76	5.59	46.75	420.82
(16,32)	1.00	0.74	5.64	49.37	499.72
(32,2)	0.50	0.65	5.25	44.59	372.27
(32,16)	1.00	0.74	5.53	43.50	361.24
(32,1)	0.25	0.73	5.79	45.09	356.42
(64,1)	0.50	0.64	5.16	41.50	360.11
(128,1)	1.00	0.72	5.30	40.90	336.87
(256,1)	1.00	0.71	5.23	40.86	327.64
(512,1)	1.00	0.76	5.51	42.30	327.68

FIGURE 1.14: C1060 run times (seconds) for GPU1 for different block dimensions

Run Time

Figure 1.14 gives the time taken by $GPU1$ for different values of n and different block dimensions. These times were obtained on a C1060, which is of compute capability 1.3. The shown times do not include the time taken to copy matrices A and B to device memory or that needed to copy C from device memory to host memory. This figure also gives the multiprocessor occupancy. Recall that when the block dimensions are (x, y), we use a $y \times x$ arrangement of threads and a $y \times x$ tile. So, each block computes a $y \times x$ sub-matrix of the product matrix. Hence, when $x = 1$ as is the case for the first 5 rows of Figure 1.14, each thread block computes y contiguous elements of a column of C and when $y = 1$ (last 5 rows of Figure 1.14), each thread block computes x contiguous elements of a row of C. Observe that when $n = 16,384$ the best ((256,1)) and worst ((1,512)) choice for block dimensions result in performances that differ by a factor of almost 30. This is despite the fact that both choices result in an occupancy of 1.0. Interestingly, when 16×16 thread blocks are used, as is done in [5], the run time for $n = 16383$ is 28% more than when the thread block dimensions area $(256, 1)$.

We can explain some of the difference in the observed run times of $GPU1$ by the number of device-memory accesses that are made and the amount of data transferred between the device memory and the SMs (we shall see shortly that the amount of data transferred may vary from one access to the next) for each chosen (x, y). Recall that the latency of a device-memory access is 400 to 600 cycles [5]. The warp scheduler is often able to hide much or all of this latency by scheduling the execution of warps that are ready to do arithmetics while other warps are stalled waiting for a device-memory access to complete. To succeed with this strategy, there must, of course, be ready warps waiting to execute. So, it often pays to minimize the number of device-memory accesses as well as the amount (or volume) of data transferred.

Number of Device-Memory Accesses

For devices with compute capability 1.3, device-memory accesses are scheduled on a per half-warp basis[4]. When accessing 4-byte data, a 128-byte segment of device memory may be accessed with a single device-memory transaction. If all data to be accessed by a half warp lie in the same 128-byte segment of device memory, a single memory transaction suffices. If the data to be accessed lie in (say) 3 different segments, then 3 transactions are required and each incurs the 400 to 600 cycle latency. Since, for compute capability 1.3, device-memory accesses are scheduled half-warp at a time, to service the device-memory accesses of a common instruction over all 32 threads of a warp requires at least 2 transactions. Devices with compute capability 2.0 schedule device-memory accesses on a per warp basis. So, if each thread of a warp accesses a 4-byte word and all threads access data that lie in the same 128-byte segment, a single memory transaction suffices. Since devices with compute capability 2.0 cache device-memory data in L1 and L2 cache using cache lines that are 128 bytes wide, the throughput of a memory transaction is much higher whenever there is a cache hit [5].

To analyze the number of device-memory transactions done by $GPU1$ and subsequent GPU codes, we assume that the arrays a, b, and c are all segment aligned (i.e., each begins at the boundary of a 128-byte segment of device memory). Consider the statement `temp += a[i*n+k]*b[k*n+j]` for any fixed value of k. When $blockDim.x \geq 16$ and a power of 2 (e.g., (32,1), (256,1), (16,16), (32,16)), in each iteration of the `for` loop of $GPU1$, the threads of a half warp access the same value of a and the accessed values of b lie in the same 128-byte segment. So, when the compute capability is 1.3, 2 transactions are needed to fetch the a and b values needed by a half warp in each iteration of the `for` loop. Note that 2 transactions (one for a and the other for b) suffice to fetch the values needed by a full warp when the compute capability is 2.0 and $blockDim.x$ is a power of 2 and ≥ 32. When the compute capability is 2.0 and $blockDim.x = 16$, 2 transactions are needed to read the values of b needed by a warp (because these lie in two different 128-byte segments) and 2 are needed for the two different values of a that are to be read (these also lie in 2 different segments).

The statement `c[i*n+j] = temp` does a write access on c. When $blockDim.x = 16$, a warp needs 2 transactions regardless of the compute capability. When $blockDim.x \geq 32$ and a power of 2, 2 transactions are required for compute capability 1.3 and 1 for capability 2.0.

We note that the `for` loop is iterated n times and that the total number of half warps is $n^2/16$. So, when $blockDim.x \geq 16$ and a power of 2, the total number of memory transactions is $n^3/16$ (for a) + $n^3/16$ (for b) + $n^2/16$ (for c) = $n^3/8+n^2/16$, for compute capability 1.3. For compute capability 2.0 and $blockDim.x = 16$, the number of transactions is also = $n^3/8+n^2/16$, because each warp accesses two values of a that lie in different segments, the b values accessed lie in 2 segments as do the c values. However, when $blockDim.x \geq 32$ and a power of 2, the number of transactions is $n^3/16+n^2/32$ for compute capability 2.0. Additionally, several of the memory transactions may be serviced from cache when the compute capability is 2.0.

To simplify the remaining discussion, we limit ourselves to devices with compute capability 1.3. For such devices, $GPU1$ has the same number, $n^3/8+n^2/16$, of memory transactions whenever $x \geq 16$ and a power of 2. This is true for all but 6 rows of Figure 1.14. The first 5 rows of this figure have $x = 1$. When $x = 1$, the threads of a half warp access a values

[4]The partitioning of a block of threads into half warps and warps is done by first mapping a thread index to a number. When the block dimensions are (D_x, D_y), the thread (x, y) is mapped to the number $x + yD_x$ [5]. Suppose that threads t_1 and t_2 map to the numbers m_1 and m_2, respectively. t_1 and t_2 are in the same half warp (warp) iff $\lfloor m_1/16 \rfloor = \lfloor m_2/16 \rfloor$ ($\lfloor m_1/32 \rfloor = \lfloor m_2/32 \rfloor$).

that are on the same column (and so different rows) of A and, for $n > 32$, the accessed values are in different 128-byte segments. Each half warp, therefore, makes 16 transactions to fetch the needed a values on each iteration of the `for` loop. On each iteration of the `for` loop, the threads of a half warp fetch a common b value using a single transaction. Since the c values written by a half warp are on the same column of C, the half warp makes 16 write transactions. The total number of transactions is n^3 (for a) + $n^3/16$ (for b) + n^2 (for c) = $17n^3/16 + n^2$, which is substantially more than when $x \geq 16$ and a power of 2.

For the 8×8 case, a half warp fetches, on each iteration of the `for` loop, 2 a values from different segments and 8 different bs that are in the same segment. So, over all n iterations of the `for` loop, a half warp makes $2n$ a transactions and n b transactions. Since there are $n^2/16$ half warps, the number of read transactions is $n^3/8$ (for a) $+n^3/16$ (for b). The c values written out by a half warp fall into 2 segments. So, the number of write transactions is $n^2/8$. The total number of device-memory transactions is $3n^3/16 + n^2/8$.

Performance is determined not just by the number of memory transactions but also by their size. Although a GPU can transfer 128-bytes of data between the host and and SM in one transaction, the actual size of the transaction (i.e., the amount of data transferred) may be 32-, 64-, or 128-bytes. A 128-byte segment is naturally decomposed into 4 32-byte subsegments or 2 64-byte subsegments. If the data to be transferred lie in a single 32-byte subsegment, the transaction size is 32 bytes. Otherwise, if the data lie in a single 64-byte subsegment, the transaction size is 64 bytes. Otherwise, the transaction size is 128 bytes. For $GPU1$, we see that when $x \geq 16$ and a power of 2, 32-byte transactions are used for a and 64-byte transactions are used for b and c. For the remaining cases of Figure 1.14, 32-byte transactions are used for a, b, and c. Since it is possible to do 4 times as many 32-byte transactions (or twice as many 64-byte transactions) in the same time it takes to do one 128-byte transactions, performance may be improved by reducing the transaction size while keeping the number of transactions steady.

Average bandwidth utilization (ABU) and *volume* are two other important metrics with respect to device-memory transactions. Consider the case when $x \geq 16$ and a power of 2. The reads of a are accomplished using 32-byte transactions. However, each such 32-byte payload carries only 4-bytes of useful data (i.e., the solitary a value to be used by all threads of the half warp. So, these transactions effectively utilize only 12.5% of the bandwidth required by a 32-byte transaction. This is the minimum utilization possible for a 32-byte transaction (recall that we are limiting our discussion to 4-byte data). A 64-byte transaction has at least 4-bytes of useful data in each of its 2 32-byte subsegments (otherwise, the transaction size would be 32-bytes). So, its minimum utilization is also 12.5%. A 128-byte transaction, on the other hand, may have only 4 bytes of useful data in each of its 2 64-byte subsegments yielding a utilization of only 6.25%.

When $x \geq 16$ and a power of 2, the $n^3/16$ transactions on a have a utilization of 12.5%, the $n^3/16$ transaction on b have a utilization of 100% as do the $n^2/16$ transactions on c. For large n, we may ignore the transaction on c and compute the average bandwidth utilization of a transaction as $(12.5 + 100)/2 = 56.25\%$. When $x = 1$, the utilization of each transaction is 12.5%. So, the ABU is 12.5%. For the 8×8 case, the $n^3/8$ transactions on a have a utilization of 12.5% while the utilization of the $n^3/16$ transactions on b is 100%. So, the ABU is $(2 * 12.5 + 100)/3 = 41.7\%$.

The *volume* of data transfer between the SMs and device memory is the sum of the number of transactions of each size multiplied by the transaction size. So, when $x \geq 16$ and a power of 2, the volume is $n^3/16 * 32 + n^3/16 * 64 + n^2/16 * 64 = 6n^3 + 4n^2$ bytes. Data volume divided by the bandwidth between device memory and the SMs is a lower bound on the time spent transferring data between device memory and the SMs. This also is a lower bound on the time for the entire computation. So, it often pays to reduce the volume of data transfer.

(x, y)	Transactions			Volume	ABU
	Total	32-byte	64-byte		
$(1, *)$	$17n^3/16 + n^2$	$17n^3/16 + n^2$	0	$34n^3 + 32n^2$	12.5%
$(8, 8)$	$3n^3/16 + n^2/8$	$3n^3/16 + n^2/8$	0	$6n^3 + 4n^2$	41.7%
rest	$n^3/8 + n^2/16$	$n^3/16$	$n^3/16 + n^2/16$	$6n^3 + 4n^2$	56.25%

FIGURE 1.15: Device-memory transaction statistics for $GPU1$

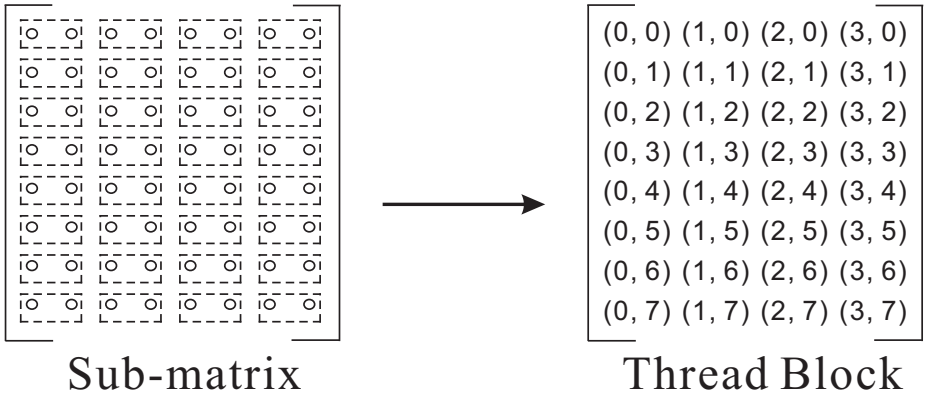

FIGURE 1.16: An 8×8 sub-matrix computed by a $(4, 8)$ block of threads

Figure 1.15 summarizes our analysis of device-memory transactions made by $GPU1$ for different block dimensions.

1.7.2 A Thread Computes a 1×2 Sub-matrix of C

Kernel Code

We can reduce the number of memory transactions by having each thread read 4 values of a (i.e., a 1×4 sub-matrix) at a time using the data type `float4`, read 2 values of b (i.e., a 1×2 sub-matrix) at a time using the data type `float2`, and write 2 values (i.e., a 1×2 sub-matrix) of c at a time also using the data type `float2`. Now, each thread computes a 1×2 sub-matrix of c. So, for example, we could use a thread block with dimensions $(4, 8)$ to compute an 8×8 sub-matrix of C (this is equivalent to using a tile size of 8×8) as in Figure 1.16. So, for example, thread $(0, 0)$ computes elements $(0, 0)$ and $(0, 1)$ of the sub-matrix while thread $(3, 7)$ computes elements $(7, 6)$ and $(7, 7)$ of the sub-matrix.

The solid circles of Figure 1.17 shows the A and B values read from device memory by a thread that computes the 1×2 sub-matrix of C identified by solid circles. The A values are read in units of 4 as shown by the boxes in the figure; the B values are read in units of 2; and the C values are written to device memory in units of 2.

Figure 1.18 gives the kernel $GPU2$ for a thread that computes a 1×2 sub-matrix of C. Suppose we index the 1×2 sub-matrices of C by the tuples (i, j), $0 \leq i < n$ and $0 \leq j < n/2$, top to bottom and left to right. Then the 1×2 sub-matrix of C computed by a thread has $i = blockIdx.y * blockDim.y + threadIdx.y$ and $j = blockIdx.x * blockDim.x + threadIdx.x$.

$GPU2$ uses 16 registers per thread and each block requires 44 bytes of shared memory.

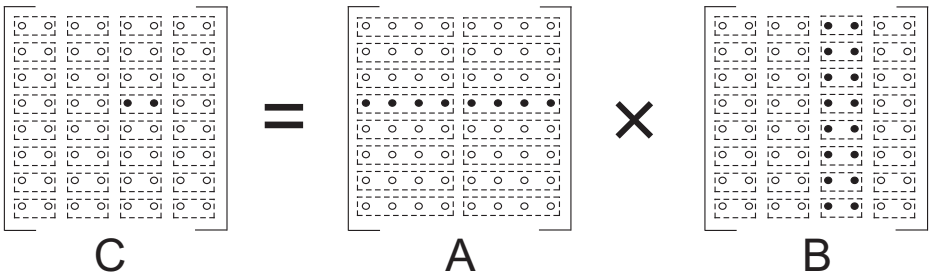

FIGURE 1.17: Computing a 1×2 sub-matrix of C

```
__global__ void GPU2 (float *a, float *b, float *c, int n)
{// thread code to compute a 1 x 2 sub-matrix of c
   // cast a, b, and c into types suitable for I/O
   float4 *a4 = (float4 *) a;
   float2 *b2 = (float2 *) b;
   float2 *c2 = (float2 *) c;

   // determine index of 1 x 2 c sub-matrix to compute
   int i = blockIdx.y * blockDim.y + threadIdx.y;
   int j = blockIdx.x * blockDim.x + threadIdx.x;

   int nDiv2 = n/2;
   int nDiv4 = n/4;
   int aNext = i*nDiv4;
   int bNext = j;
   float2 temp2;
   temp2.x = temp2.y = 0;
   for (int k = 0; k < nDiv4; k++)
   {
      float4 aIn = a4[aNext++];
      float2 bIn = b2[bNext];
      temp2.x += aIn.x*bIn.x; temp2.y += aIn.x*bIn.y;
      bNext += nDiv2;
      bIn = b2[bNext];
      temp2.x += aIn.y*bIn.x; temp2.y += aIn.y*bIn.y;
      bNext += nDiv2;
      bIn = b2[bNext];
      temp2.x += aIn.z*bIn.x; temp2.y += aIn.z*bIn.y;
      bNext += nDiv2;
      bIn = b2[bNext];
      temp2.x += aIn.w*bIn.x; temp2.y += aIn.w*bIn.y;
      bNext += nDiv2;
   }
   c2[i*nDiv2+j] = temp2;
}
```

FIGURE 1.18: A thread computes a 1×2 sub-matrix of c

(x, y)	Transactions				Volume	ABU
	Total	32-byte	64-byte	128-byte		
$(1, *)$	$5n^3/32 + n^2/2$	$5n^3/32 + n^2/2$	0	0	$5n^3 + 16n^2$	45.0%
$(8, 8)$	$3n^3/64 + n^2/16$	$n^3/64$	$n^3/32 + n^2/16$	0	$2.5n^3 + 4n^2$	83.3%
rest	$5n^3/128 + n^2/32$	$n^3/128$	0	$n^3/32 + n^2/32$	$4.25n^3 + 4n^2$	90.0%

FIGURE 1.19: Device-memory transaction statistics for $GPU2$

Since a compute capability 1.3 GPU has 16,384 registers, it should be possible to co-schedule up to 1,024 threads or 32 warps on an SM and achieve an occupancy of 1.0. This happens, for example, when the block size is 16×16.

Number of Device-Memory Accesses

Consider a single iteration of the `for` loop of $GPU2$. The 4 a values read from device memory are in the same 32-byte segment. For $(1, *)$ thread blocks, the 16 sets of a values read by a half warp are in different 128-byte segments. So, 16 32-byte transactions are done on a on each iteration. The number of iterations is $n/4$ and the number of half warps is $n^2/32$ (recall that each half warp computes 2 c values). So, a total of $n^3/8$ 32-byte transaction are done on a. The utilization of each of these transactions is 50%. For b, $GPU2$ makes 32-byte transactions whose utilization is 25%. Four such transactions are made per iteration of the `for loop` for a total of n transactions per half warp. So, the total number of b transactions is $n^3/32$. A half warp makes 16 c transactions of size 32 bytes with 25% utilization. The total number of transactions of all types is $5n^3/32 + n^2/2$.

When 8×8 thread blocks are used, we need 2 transactions per iteration of the `for` loop to read the a values and 4 to read the b values. A half warp also needs 2 transactions to write the c values. So, the number of device-memory transactions is $n^3/64$ (for a) + $n^3/32$ (for b) + $n^2/16$ (for c) = $3n^3/64 + n^2/16$. The size of the a transactions is 32-bytes and their utilization is 50%. The b and c transactions are 64-bytes each and have a utilization of 100%.

For the remaining block sizes of Figure 1.14, the total number of transactions is $n^3/128$ (for a) + $n^3/32$ (for b) + $n^2/32$ (for c). The a transactions are 32-bytes each and have a utilization of 50% while the b and c transactions are 128 bytes and have a utilization of 100%. Figure 1.19 summarizes the device-memory transaction properties of $GPU2$.

Run Time

Figure 1.20 gives the time taken by $GPU2$ for different values of n and different block dimensions. The best performance is obtained using 8×8 blocks. This is not entirely surprising given that the volume of data movement between device-memory and SMs is least when the block dimensions are $(8, 8)$ (Figure 1.19). Dimensions of the form $(1, *)$ result in the maximum data volume and this translates into the largest execution times. It is interesting that $GPU2$ exhibits very little variation in run time over the different dimensions of the form $(i * 16, *)$. There is some correlation between the reduction in data volume when going from $GPU1$ to $GPU2$ and the reduction in run time. For example, $GPU1$ has a volume that is 6.8 times that of $GPU2$ when block dimensions are of the form $(1, *)$ and for these dimensions, the run times for the $GPU1$ code is between 6.6 and 7.1 times that of $GPU2$ when $n = 16,384$. For the $(8, 8)$ case, the ratio of the volumes is 2.4 and the ratio of the run times for $n = 16,384$ is 2.8. For the remaining cases, the volume ratio is 1.4 and the

(x, y)	occupancy	n			
		2048	4096	8192	16384
(1,32)	0.25	0.97	6.65	52.35	667.91
(1,64)	0.50	0.99	7.43	65.82	885.25
(1,128)	1.00	1.07	8.07	77.91	1058.65
(1,256)	1.00	1.11	8.79	92.86	1212.03
(1,512)	1.00	1.22	10.68	112.78	1378.81
(8,8)	0.50	0.26	2.06	17.65	194.04
(16,4)	0.50	0.41	3.29	27.23	223.48
(16,16)	1.00	0.42	3.30	27.26	213.73
(16,32)	1.00	0.42	3.34	27.38	217.04
(32,2)	0.50	0.41	3.28	26.69	217.58
(32,16)	1.00	0.42	3.33	26.88	210.48
(32,1)	0.25	0.41	3.28	26.69	229.14
(64,1)	0.50	0.41	3.28	26.72	216.23
(128,1)	1.00	0.41	3.29	26.76	216.13
(256,1)	1.00	0.42	3.30	26.77	212.96
(512,1)	1.00	0.42	3.31	26.75	211.35

FIGURE 1.20: C1060 run times (seconds) for GPU2 for different block dimensions

time ratio is between 1.53 and 1.67 except for 2 cases where the time ratio is 1.97 and 2.3. Though not reported in Figure 1.20, the dimensions $(4, 16)$ also did fairly well computing the product of two 16384×16384 matrices in 223 seconds with an occupancy of 0.50.

1.7.3 A Thread Computes a 1×4 Sub-matrix of C

Kernel Code

Extending the strategy of Section 1.7.2 so that each thread computes 4 elements of c (say, a 1×4 sub-matrix) enables each thread to read 4 elements of b at a time using the data type `float4` and also to write 4 elements of c at a time. Figure 1.21 gives the resulting code, $GPU3$. When, for example, $GPU3$ is invoked with 16×16 thread blocks, each thread block computes a 16×64 sub-matrix of c. When the block dimensions are $(4, 16)$, a 16×16 sub-matrix of c is computed by each thread block.

$GPU3$ uses 21 registers per thread while $GPU2$ uses 16. When the compute capability is 1.3, an occupancy of 1.0 cannot be achieved by codes that use more than 16 registers per thread as to achieve an occupancy of 1.0, 32 warps must be co-scheduled on an SM and 32 warps with 16 registers per thread utilize all 16,384 registers available on an SM of a device with compute capability 1.3. So, for example, when $GPU3$ is invoked with 16×16 thread blocks, the occupancy of $GPU3$ is 0.5.

Run Time

Figure 1.22 gives the time taken by $GPU3$ for different values of n. Although we experimented with many choices for block dimensions, only the times for the 6 best choices are reported in this figure. We remark that the remaining dimensions of the form $(*, 1)$ reported in Figures 1.14 and 1.20 performed almost as well as did $(64, 1)$ and while the remaining dimensions of the form $(1, *)$ performed better than they did for $GPU2$, they took between 50% to 110% more time than taken by $(1, 32)$. Of the 6 dimensions reported on in

```
__global__ void GPU3 (float *a, float *b, float *c, int n)
{// thread code to compute a 1 x 4 sub-matrix of c
   // cast a, b, and c into types suitable for I/O
   float4 *a4 = (float4 *) a;
   float4 *b4 = (float4 *) b;
   float4 *c4 = (float4 *) c;

   // determine index of 1 x 4 c sub-matrix to compute
   int i = blockIdx.y * blockDim.y + threadIdx.y;
   int j = blockIdx.x * blockDim.x + threadIdx.x;

   int nDiv4 = n/4;
   int aNext = i*nDiv4;
   int bNext = j;
   float4 temp4;
   temp4.x = temp4.y = temp4.z = temp4.w = 0;
   for (int k = 0; k < nDiv4; k++)
   {
      float4 aIn = a4[aNext++];
      float4 bIn = b4[bNext];
      temp4.x += aIn.x*bIn.x; temp4.y += aIn.x*bIn.y;
      temp4.z += aIn.x*bIn.z; temp4.w += aIn.x*bIn.w;
      bNext += nDiv4;
      bIn = b4[bNext];
      temp4.x += aIn.y*bIn.x; temp4.y += aIn.y*bIn.y;
      temp4.z += aIn.y*bIn.z; temp4.w += aIn.y*bIn.w;
      bNext += nDiv4;
      bIn = b4[bNext];
      temp4.x += aIn.z*bIn.x; temp4.y += aIn.z*bIn.y;
      temp4.z += aIn.z*bIn.z; temp4.w += aIn.z*bIn.w;
      bNext += nDiv4;
      bIn = b4[bNext];
      temp4.x += aIn.w*bIn.x; temp4.y += aIn.w*bIn.y;
      temp4.z += aIn.w*bIn.z; temp4.w += aIn.w*bIn.w;
      bNext += nDiv4;
   }
   c4[i*nDiv4+j] = temp4;
}
```

FIGURE 1.21: A thread computes a 1×4 sub-matrix of c

(x, y)	occupancy	n			
		2048	4096	8192	16384
(1,32)	0.25	0.51	3.64	29.11	336.89
(2,32)	0.50	0.29	2.14	17.27	199.92
(4,16)	0.50	0.27	2.06	16.28	130.28
(8,8)	0.50	0.38	3.01	24.10	192.71
(16,32)	0.50	0.71	5.63	43.98	351.63
(64,1)	0.50	0.69	5.50	43.97	351.67

FIGURE 1.22: C1060 run times (seconds) for GPU3 for different block dimensions

Figure 1.22, the performance for the dimensions $(64, 1)$ and $(16, 32)$ was better for $GPU2$ than $GPU3$. The other 4 saw a performance improvement using the strategy of $GPU3$. The best performer for $GPU3$ is $(4, 16)$.

Number of Device-Memory Accesses

We now derive the device-memory transaction statistics for the top 3 performers ($(4, 16)$, $(8, 8)$ and $(2, 32)$) of Figure 1.22. In all cases, the number of half warps is $n^2/64$. When the dimensions are $(4, 16)$, a thread makes 1 a access and 4 b accesses of device memory in each iteration of the `for` loop. The a accesses made by a half warp are to `float4`s from different rows of A and hence from different 128-byte segments. So, a half warp makes 4 32-byte device-memory transactions per iteration of the `for` loop. The utilization of each access is 50%. The total number of a transactions is, therefore, $4 * n/4 * n^2/64 = n^3/64$. When the threads of a half warp read a `float4` of b, they read 4 `float4`s that lie in the same 64-byte segment. So, this read is accomplished using a single 64-byte transaction whose utilization is 100%. A thread reads 4 `float4`s of b in each iteration of the `for` loop. So, in each iteration of the `for` loop, a half warp makes 4 64-byte transactions on b and the utilization of these transactions is 100%. Hence, the total number of b transactions is $4 * n/4 * n^2/64 = n^3/64$. For c, a half warp makes 4 64-byte transactions with utilization 100%. The total number of c transactions is $n^2/16$. The volume and ABU may be calculated easily now.

When the dimensions are $(8, 8)$, a half warp reads, in each iteration of the `for` loop, two `float4`s of a values that lie in different 128-byte segments. This requires 2 32-byte transactions and their utilization is 50%. So, the total number of a transactions is $2 * n/4 * n^2/64 = n^3/128$. A half warp accesses 32 b values that are in the same 128-byte segment each time its thread read a `float4` of b. A thread does 4 such reads in an iteration of the `for` loop. So, a total of $4 * n/4 * n^2/64 = n^3/64$ 128-byte transactions are done for b. The utilization of each of these is 100%. Each half warp uses 2 128-byte transactions to write the computed c values. So, a total of $n^2/32$ 128-byte transactions are done for c and the utilization of each of these is 100%.

The final analysis we do is the dimensions $(2, 32)$. In each iteration of the `for` loop, a half warp reads 8 float4s of a values that are in different 128-byte segments. So, $GPU3$ makes a total of $8 * n/4 * n^2/64 = n^3/32$ 32-byte transactions on a. The utilization of each of these is 50%. Further, on each iteration of the `for` loop, a half warp reads $2 * 4$ `float4`s of bs using 4 32-byte transactions whose utilization is 100%. So, a total of $4 * n/4 * n/64 = n^3/64$ 32-byte transactions with utilization 100% are made for b. To write the c values, a half warp makes 8 32-byte transactions whose utilization is 100%. Figure 1.23 summarizes the transaction statistics for $GPU3$. Of the 3 dimensions analyzed, $(2, 32)$ and $(4, 16)$ transfer the smallest volume of data between device memory and the SMs and $(4, 16)$ does this transfer using

(x, y)	Transactions				Volume	ABU
	Total	32-byte	64-byte	128-byte		
$(4, 16)$	$n^3/32 + n^2/16$	$n^3/64$	$n^3/64 + n^2/16$	0	$1.5n^3 + 4n^2$	75.0%
$(8, 8)$	$3n^3/128 + n^2/32$	$n^3/128$	0	$n^3/64 + n^2/32$	$2.25n^3 + 4n^2$	83.3%
$(2, 32)$	$3n^3/64 + n^2/8$	$3n^3/64 + n^2/8$	0	0	$1.5n^3 + 4n^2$	66.7%

FIGURE 1.23: Device-memory transaction statistics for $GPU3$

a smaller number of transactions. These factors, most likely, played a significant role in causing $GPU3$ to exhibit its best performance when the dimensions are $(4, 16)$.

1.7.4 A Thread Computes a 1×1 Sub-matrix of C Using Shared Memory

First Kernel Code and Analysis

To improve on the performance of $GPU2$, we resort to a block matrix multiplication algorithm in which each of A, B, and C is partitioned into n^2/s^2 $s \times s$ sub-matrices A_{ij}, B_{ij}, and C_{ij}. We assume that s divides n. The algorithm computes C using the equation:

$$C_{ij} = \sum_{0 \le k < n/s} A_{ik} * B_{kj}$$

In the strategy of this subsection, a block of threads computes one $s \times s$ sub-matrix of C and each thread computes one element of this sub-matrix. So, we use $s \times s$ thread blocks with thread (x, y) computing element (y, x) of the sub-matrix (recall the difference in the convention to name threads and matrix elements). The thread block that is to compute C_{ij} executes the following pseudocode:

Step 1: Repeat Steps 2 and 3 for $0 \le k < n/s$.

Step 2: Each thread of the $s \times s$ thread block reads one value of A_{ik} and one value of B_{kj} from device memory.

Step 3: Each thread updates the element of C_{ij} it is computing by multiplying the appropriate row of A_{ik} with the appropriate column of B_{kj}. This update step accesses shared memory but not device memory.

Step 4: Each thread writes the element of C_{ij} it has computed to device memory.

Figure 1.24 gives the kernel code for the case $s = 16$. We note that $GPU4$ is essentially the same as $MatMulKernel$ on page 25 of [5]. As noted in Section 1.7.1, the constraints of NVIDIA GPUs with compute capability 1.3 require us to use $s = 16$ if we are to have any prospect of achieving an occupancy of 1.0. Although, as we have seen earlier, it is possible to get better performance with a lower occupancy than a higher one, we verified experimentally that $s = 16$ gives best performance. Further, performance is not improved by storing $as[16][t]$ and $bs[t][16]$ in shared memory for $t \neq 16$.

We now obtain the device-memory access statistics for $GPU4$. Since each thread computes a single value of c, the number of half warps is $n^2/16$. In each iteration of the `for` loop, a half warp reads 64 bytes of a values from a single 128-byte segment using a 64-byte transaction and 64 bytes of b values using another 64-byte transaction. Since the `for` loop is iterated $n/16$ times, a half warp makes $n/8$ 64-byte transactions of a and b together. A half

```
__global__ void GPU4 (float *a, float *b, float *c, int n)
{// thread code to compute an element of a 16 x 16 sub-matrix of c
   // shared memory arrays to hold a 16 x 16 sub-matrix of a and b
   __shared__ float as[16][16], bs[16][16];
   int nDiv16 = n/16;
   int nTimes16 =  n*16;
   int aNext = (16*blockIdx.y+threadIdx.y)*n+threadIdx.x;
   int bNext = 16*blockIdx.x+threadIdx.y*n+threadIdx.x;
   float temp = 0;

   for (int u = 0; u < nDiv16; u++)
   {// threads in a thread block collectively read a 16 x 16 sub-matrix of
    // a and b from device memory to shared memory; each thread reads
    // 1 element of a and 1 element of b
      as[threadIdx.y][threadIdx.x] = a[aNext];
      bs[threadIdx.y][threadIdx.x] = b[bNext];
      __syncthreads(); // wait for read to complete

      // multiply a row of as with a column of bs
      for (int k = 0; k < 16; k++)
         temp += as[threadIdx.y][k]*bs[k][threadIdx.x];
      __syncthreads(); // wait for all threads in thread block to complete

      // update to work on next sub-matrix of a and b
      aNext += 16;
      bNext += nTimes16;
   }
   c[(16*blockIdx.y+threadIdx.y)*n + 16*blockIdx.x + threadIdx.x] = temp;
}
```

FIGURE 1.24: A $(16, 16)$ thread block computes a 16×16 sub-matrix of c using shared memory

warp also makes 1 64-byte write transaction on c. So, the total number of device-memory transactions made by $GPU4$ is $n^3/128 + n^2/16$. Each of these is a 64-byte transaction with 100% utilization. The volume is $n^3/2 + 4n^2$ and the ABU is 100%. $GPU4$ uses 11 registers per thread and 2092 bytes of shared memory; it achieves an occupancy of 1.0.

Compared to the $(4, 16)$ case of $GPU3$, which results in best performance for $GPU3$, $GPU4$ has a lower volume (67% lower), a higher ABU (33% higher), and a higher occupancy (double that of $GPU3$). So, we expect $GPU4$ to have a much better performance than $GPU3$.

Improved Kernel Code

The performance of $GPU4$ may be improved by tuning shared-memory accesses. This tuning does not affect the volume, ABU, or occupancy of the kernel. A thread accesses a row of as and a column of bs. Although the CUDA manual does not document a cache for shared memory, it appears that it is faster for a thread to access data by rows rather than by columns. We can take advantage of this asymmetry in access time by storing in bs the transpose of the sub-matrix of b that is read from device memory. While this speeds access for a single thread, it results in shared-memory access conflicts for a half-warp of threads. To see why this is so, we need to be familiar with the organization of shared memory on a GPU.

On a device with compute capability 1.3, shared memory is divided into 16 banks using a round robin allocation of 4-byte words. So, in the shared-memory array as of type `float`, $as[i]$ and $as[j]$ are in the same bank iff $i \bmod 16 = j \bmod 16$. (A device of compute capability 2.0 has 32 banks of shared memory.) Shared memory read/writes from different banks can be serviced at the same time. When two or more threads need to access the same bank, there is a shared-memory conflict and the read/writes get serialized into a number of conflict free read/writes. So, performance is maximized when there are no bank conflicts.

When a and b are read from device memory and written to shared memory by a half warp in $GPU4$, the half warp accesses 16 adjacent elements of as and bs. These lie in different banks of shared memory and so, the shared-memory accesses are conflict free. Similarly, the reads of bs, by a half warp, in the inner `for` loop are conflict free. Since all threads in a half warp read the same as in the inner `for` loop, all threads in the half warp get this as value making a common access to shared memory. If we modify $GPU4$ to save the transpose of the b sub-matrix in bs by changing the statement

```
bs[threadIdx.y][threadIdx.x] = b[bNext];
```

to

```
bs[threadIdx.x][threadIdx.y] = b[bNext];
```

and correspondingly change the statement

```
temp += as[threadIdx.y][k]*bs[k][threadIdx.x];
```

to

```
temp += as[threadIdx.y][k]*bs[threadIdx.x][k];
```

then threads in a half warp write and read bs values that are in the same column of bs. Since the values in a column are in the same bank of shared memory (assuming the rows of bs are assigned to a contiguous block of $16 \times 16 \times 4 = 1024$ bytes of shared memory), the 16 threads of the half warp need to read 16 different bs values from the same bank (note that elements of a column of bs are 16 4-byte words apart and so are in the same

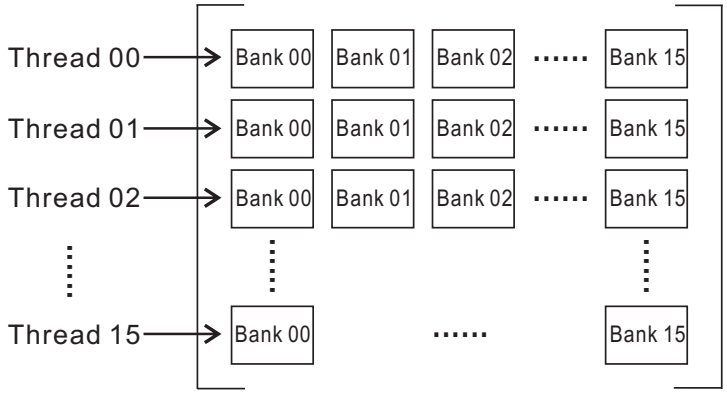

FIGURE 1.25: Bank mapping for $bs[16][16]$

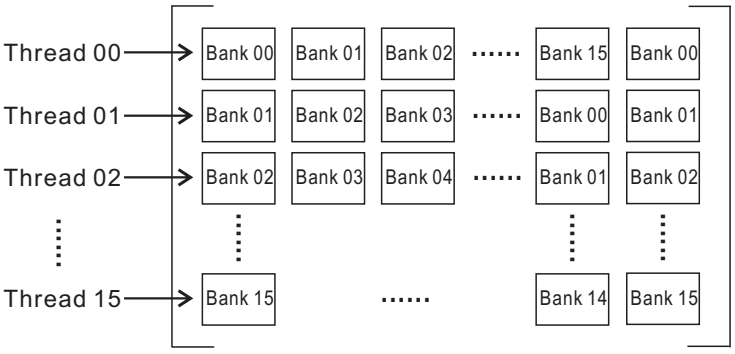

FIGURE 1.26: Bank mapping for $bs[16][17]$

bank). Figure 1.25 shows the shared-memory bank mapping for $bs[16][16]$. The resulting bank conflict is serialized into 16 shared-memory accesses.

To avoid bank conflicts in the reading of bs values, we define bs to be a 16×17 array. Now, elements in a column are 17 4-byte words apart and so are in different banks (see Figure 1.26).

Modifying $GPU4$ to store the transpose of sub-matrices of b in bs and defining bs so that elements of a column are in different banks of shared memory gives us $GPU5$ (Figure 1.27).

Figure 1.28 gives the time taken by $GPU4$ and $GPU5$ for different values of n. We see that for $n = 16384$, the shared-memory optimization done by $GPU5$ has reduced run time by a little over 4%. For this value of n, $GPU4$ achieves a speedup of a little over 2.8 relative to the best case ($(4, 16)$) for $GPU3$ while $GPU5$ achieves a speedup of almost 3.0. Amazingly, an additional speedup of almost 2 is possible!

```
__global__ void GPU5 (float *a, float *b, float *c, int n)
{// thread code to compute an element of a 16 x 16 sub-matrix of c
   // shared memory arrays to hold a sub-matrix of a and b
   __shared__ float as[16][16], bs[16][17];
   int nDiv16 = n/16;
   int nTimes16 =  n*16;
   int aNext = (16*blockIdx.y+threadIdx.y)*n+threadIdx.x;
   int bNext = 16*blockIdx.x+threadIdx.y*n+threadIdx.x;
   float temp = 0;

   for (int u = 0; u < nDiv16; u++)
   {// threads in a thread block collectively read a 16 x 16 sub-matrix of
    // a and b from device memory to shared memory; each thread reads
    // 1 element of a and 1 element of b
      as[threadIdx.y][threadIdx.x] = a[aNext];
      bs[threadIdx.x][threadIdx.y] = b[bNext];
      __syncthreads(); // wait for read to complete

      // multiply a row of as with a column of bs
      for (int k = 0; k < 16; k++)
         temp += as[threadIdx.y][k]*bs[threadIdx.x][k];
      __syncthreads(); // wait for all threads in thread block to complete

      // update to work on next sub-matrix of a and b
      aNext += 16;
      bNext += nTimes16;
   }
   c[(16*blockIdx.y+threadIdx.y)*n + 16*blockIdx.x + threadIdx.x] = temp;
}
```

FIGURE 1.27: Same as GPU4 except that the transpose of the b sub-matrix is stored in bs

	2048	4096	8192	16384
$GPU4$	0.09	0.69	5.54	46.19
$GPU5$	0.08	0.60	4.84	44.13

FIGURE 1.28: C1060 run times for $GPU4$ and $GPU5$

```
__device__ void update1(float *a, float b, float *c)
{
   for (int i = 0; i < 16; i++)
      c[i] += a[i] * b;
}
```

FIGURE 1.29: Updating c values

1.7.5 A Thread Computes a 16×1 Sub-matrix of C Using Shared Memory

First Kernel Code and Analysis

To improve the performance of the matrix multiply kernel further, we increase the computational load per thread to better mask device-memory accesses with arithmetic operations. In particular, each thread will compute 16 values of c, all on the same column. Though we can have a thread compute fewer or more values of c, 16 gave best performance. Since a thread will compute its assigned 16 values of c incrementally, we allocate a thread 16 registers $cr[0:15]$ to store the incremental values computed so far. When done, the thread will write its 16 computed values to device memory.

The incremental computation of the c values is done using an 8×16 block of threads (i.e., the block dimensions are $(16, 8)$). The 8×16 thread block first reads a 16×32 sub-matrix of a from device memory and stores its transpose into a two-dimensional shared-memory array $as[32][17]$. To do this, each thread must read in 4 values of a; the 4 values read in are from two adjacent columns of the a sub-matrix. Additionally, a half-warp reads 32 adjacent values of a that lie in the same 128-byte segment. Using a 32×17 array rather than a 32×16 array avoids bank conflicts when writing to shared memory and storing the transpose derives cache-like benefits as a thread will access as by row (see Section 1.7.4). The code of Figure 1.29 updates the value of each of the 16 cs being computed by a thread by adding in an $a[i] * b$ value. When invoked, a is a pointer to a row of as, which corresponds to a column of the 16×32 sub-matrix of a that was read in. Figures 1.30 and 1.31 give the complete code to multiply two $n \times n$ matrices using the described strategy.

To determine the device-memory statistics, we note that a thread computes 16 c values. So, a half warp computes 256 c values. Therefore, the number of half warps is $n^2/256$. In each iteration of the `for i` loop, the threads of a half warp use 2 128-byte transactions to read the required a values. The total number of transactions on a is $n^2/256 * 2 * n/32 = n^3/4096$. Each of these 128-byte transactions has a utilization of 100%. In each iteration of the `for i` loop, a half warp makes 32 64-byte transactions on b. The total number of b transactions is therefore $n^2/256 * 32 * n/32 = n^3/256$. Each of these has 100% utilization. A half warp makes 16 64-byte device-memory transactions to write out the 256 c values it computes. Therefore, the number of device-memory write-transactions for c is $n^2/16$ and each has a utilization of 100%. Combining the transactions for a, b, and c, we see that $GPU6$ makes a total of $17n^3/4096 + n^2/16$ device-memory transactions; the volume is $9n^3/32 + 4n^2$; and the ABU is 100%. Compared to $GPU4$ and $GPU5$, $GPU6$ makes about 47% fewer transactions and generates about 44% less volume. All three have the same ABU.

Second Kernel Code

Successive iterations of the `for` loop of Figure 1.31 need to wait for the preceding read of $b[0]$ to complete. Further, even after the read completes, there is a 24-cycle between the time

```
__global__ void GPU6 (float *a, float *b, float *c, int n)
{// thread code to compute one column of a 16 x 128 sub-matrix of c
 // use shared memory to hold the transpose of a 16 x 32 sub-matrix of a
   __shared__ float as[32][17];

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

   int nDiv32 = n/32;
   int sRow = threadIdx.y;
   int sCol = threadIdx.x;
   int sCol2 = sCol*2;
   int sCol2Plus1 = sCol2+1;
   int tid = sRow*16+sCol;
   int aNext = (16*blockIdx.y+sRow)*n+sCol*2;
   int bNext = 128*blockIdx.x+tid;
   int sRowPlus8 = sRow+8;
   int nTimes8 = 8*n;

   a += aNext;
   b += bNext;

   int i, j;
   float2 temp;


    for (i = 0; i< nDiv32; i++)
    {// threads in a thread block collectively read a 16 x 32
     // sub-matrix of a from device memory to shared memory
        temp = *(float2 *)a;
        as[sCol2][sRow] = temp.x;
        as[sCol2Plus1][sRow] = temp.y;
        temp = *(float2 *)(a+nTimes8);
        as[sCol2][sRowPlus8] = temp.x;
        as[sCol2Plus1][sRowPlus8] = temp.y;
        __syncthreads(); // wait for read to complete

        #pragma unroll
        for (j = 0; j < 32; j++)
        {
         float br = b[0];
            b += n;
            update1 (&as[j][0], br, cr);
        }

        a += 32;
        __syncthreads(); // wait for computation to complete

    }
```

FIGURE 1.30: A $(16, 8)$ thread block computes a 16×128 sub-matrix of C using shared memory (Part a)

```
   // output cr[]
   int cNext = 16*blockIdx.y*n + 128*blockIdx.x + tid;
   c += cNext;

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

FIGURE 1.31: A $(16, 8)$ thread block computes a 16×128 sub-matrix of C using shared memory (Part b)

a value is written to a register and the time this value can be used [4]. To reduce the impact of these delays, we read in several values of b from device memory and use these b values in round robin fashion. Figure 1.32 does this using 4 values of b in each round. Although, we could use this strategy with a different number of bs such as 2 and 8, we found that 4 gives best performance. The code of this figure assumes that $nTimes2$, $nTimes3$, and $nTimes4$ have, respectively, been defined to be $2 * n$, $3 * n$, and $4 * n$. $GPU6$ modified to use 4 values of b per round as in Figure 1.32 is referred to as $GPU7$.

Final Kernel Code

As in $GPU6$ and $GPU7$, our final matrix multiply code, $GPU8$ (Figures 1.34 and 1.35), uses $(16, 8)$ thread blocks. However, a thread block now reads a 16×64 sub-matrix of a rather than a 16×32 sub-matrix from device memory to shared memory. Each half warp reads the 64 a values in a row of the 16×64 sub-matrix, which lie in two adjacent 128-byte segments of device memory, using two 128-byte transactions. To accomplish this, each thread reads a 1×4 sub-matrix of a using the data type `float4`. The 16×64 a sub-matrix that is input from device memory may be viewed as a 16×16 matrix in which each element is a 1×4 vector. The transpose of this 16×16 matrix of vectors is stored in the array $as[16][65]$ with each 1×4 vector using four adjacent elements of a row of as. This mapping ensures that the 16 elements in each column of the 16×64 sub-matrix of a that is input from device memory are stored in different banks of shared memory. So, the writes to shared memory done by a half warp of $GPU8$ are conflict free. Further, by storing the transpose of a 16×16 matrix of 1×4 vectors rather than the transpose of a 16×64 matrix of scalars, we are able to do the writes to shared memory using `float4`s rather than `float`s as is done in $GPU6$ and $GPU7$. This reduces the time to write to shared memory. The scheme used to map a 16×64 sub-matrix of a into a 16×65 array as necessitates the use of a slightly different update method, $update2$ (Figure 1.33).

The number of half warps is $n^2/256$. In each iteration of the `for i` loop, a half warp makes 4 128-byte transactions to read in a values and 64 64-byte transactions to read in b values. $GPU8$ makes $n^3/4096$ 128-byte device-memory transactions on a and $n^3/256$ 64-byte transactions on b. Additionally, $n^2/16$ 64-byte transactions are made on c. Each transactions has 100% utilization. So, the device memory statistics for $GPU8$ are the same as for $GPU6$ and $GPU7$.

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

#pragma unroll
for (j = 0; j< 7; j++)
{
    b += nTimes4;
    update1 (&as[j*4][0], br0, cr); br0 = b[0];
    update1 (&as[j*4+1][0], br1, cr); br1 = b[n];
    update1 (&as[j*4+2][0], br2, cr); br2 = b[nTimes2];
    update1 (&as[j*4+3][0], br3, cr); br3 = b[nTimes3];
}

b += nTimes4;
update1 (&as[28][0], br0, cr);
update1 (&as[29][0], br1, cr);
update1 (&as[30][0], br2, cr);
update1 (&as[31][0], br3, cr);
```

FIGURE 1.32: Figure 1.31 modified to handle 4 *b*s in round robin fashion

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

FIGURE 1.33: Updating *c* values when *a*s read using `float2`

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

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

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

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

    float4 *a4 = (float4 *)a;

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

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

        b += nTimes4;

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

            b+= nTimes4;
        }
```

FIGURE 1.34: *GPU*7 modified to handle 4 *b*s in round robin fashion and read a 16×32 sub-matrix of a (Part a)

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


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

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

FIGURE 1.35: $GPU7$ modified to handle 4 bs in round robin fashion and read a 16×32 sub-matrix of a (Part b)

1.8 A Comparison

GPU Kernels

To compare the 8 GPU kernels, $GPU1$ through $GPU8$, we use, for each, the block dimensions that yielded best performance. Specifically, for $GPU1$ we use $(256, 1)$ (i.e., $blockDim.x = 256$ and $blockDim.y = 1$) thread blocks, for $GPU2$ we use $(8, 8)$ blocks, for $GPU3$, we use $(4, 16)$ blocks, for $GPU4$ and $GPU5$, we use $(16, 16)$, and for $GPU6$, $GPU7$, and $GPU8$, we use $(16, 8)$ blocks. In our comparisons, we included also the code *sgemm* that is in the MAGMA BLAS 0.2 library [3] and is based on the algorithm of Volkov and Demmel [11] as well as the matrix multiply in the CUDA CUBLAS 3.0 library [1]. The *sgemm* code, which assumes a column-major mapping of matrices into one-dimensional arrays, is very similar to $GPU7$ in its strategy and uses $(16, 4)$ blocks. Since we do not have access to the source code for CUBLAS, we do not know precisely the multiplication strategy it uses. However, given the proximity of its performance to that of *sgemm*, it is very likely that *sgemm* and CUBLAS use the same strategy.

Figure 1.36 gives the number of device-memory transactions, volume, and ABU for each of our 8 GPU matrix multiply kernels as well as the *sgemm* kernel. We are unable to provide data for CUBLAS as we do not have access to its source code. Figures 1.37 and 1.38 plot the number of transactions (in billions) and the volume for 16384×16384 matrices. We note that each refinement of the kernel resulted in either a reduction or no change in the total number of transactions and in the volume of data moved between the device memory and the SMs. So, there is a good correlation between performance and number of transactions as well as volume. This isn't too surprising as when there are more transactions, there are more latencies to hide and the volume divided by the bandwidth between device memory and the SMs is a lower bound on the time needed to do the data transfer. We note also that $GPU6$, $GPU7$, and $GPU8$ make 15% fewer transactions than made by *sgemm* and generate 10% lower volume.

Version	Transactions	Volume	ABU
$GPU1$	$n^3/8 + n^2/16$	$6n^3 + 4n^2$	56%
$GPU2$	$3n^3/64 + n^2/16$	$2.5n^3 + 4n^2$	83%
$GPU3$	$n^3/32 + n^2/16$	$1.5n^3 + 4n^2$	75%
$GPU4$	$n^3/128 + n^2/16$	$0.5n^3 + 4n^2$	100%
$GPU5$	$n^3/128 + n^2/16$	$0.5n^3 + 4n^2$	100%
$GPU6$	$17n^3/4096 + n^2/16$	$9n^3/32 + 4n^2$	100%
$GPU7$	$17n^3/4096 + n^2/16$	$9n^3/32 + 4n^2$	100%
$GPU8$	$17n^3/4096 + n^2/16$	$9n^3/32 + 4n^2$	100%
$sgemm$	$5n^3/1024 + n^2/16$	$5n^3/16 + 4n^2$	100%

FIGURE 1.36: Device memory statistics for compute capability 1.3

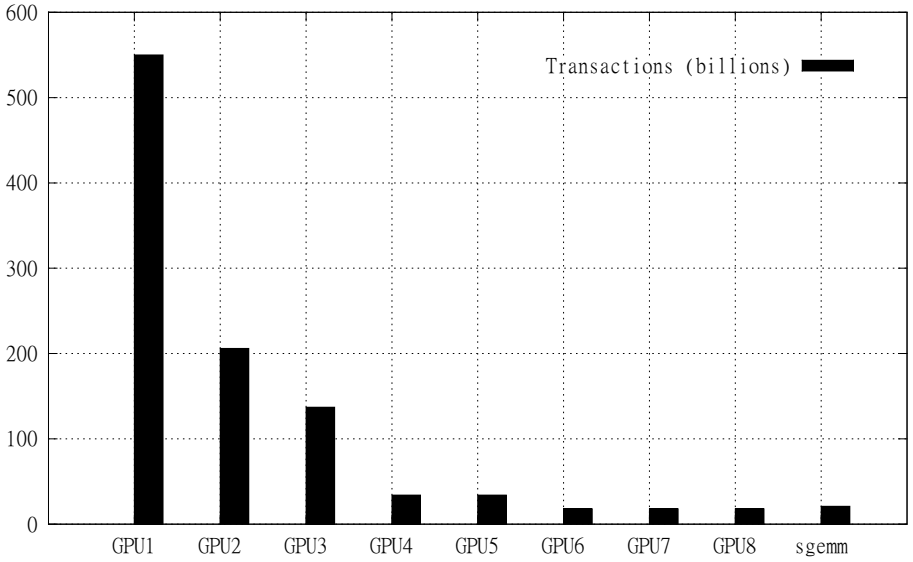

FIGURE 1.37: Transactions (in billions) for 16384×16384 matrices

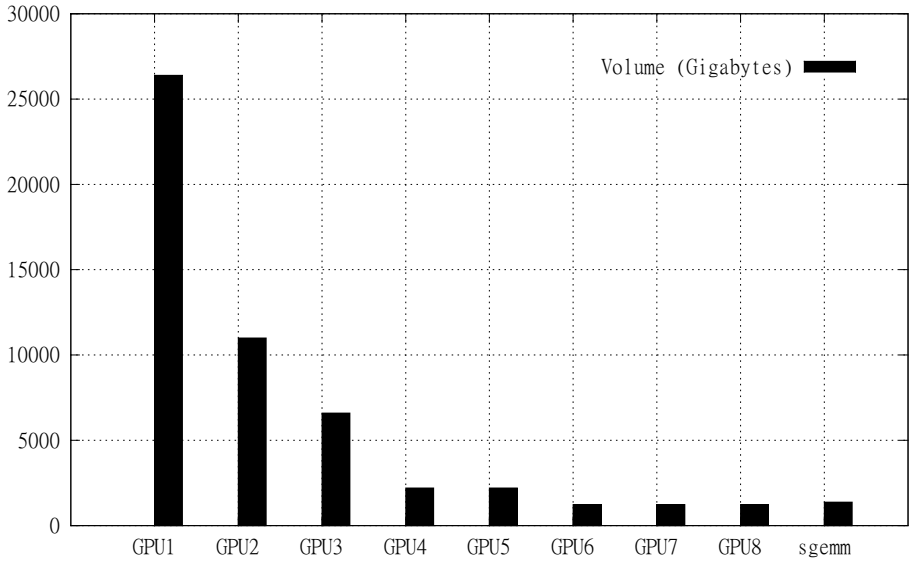

FIGURE 1.38: Volume (in gigabytes) for 16384×16384 matrices

Figure 1.39 gives the block dimensions, number of registers per thread, shared memory used by an SM, and the occupancy for the different kernels. We see that although high occupancy is needed to hide device-memory latency, high occupancy does not necessarily translate into better performance. In particular, our best performers, $GPU6$, $GPU7$, $GPU8$, and *sgemm* have the lowest occupancy. The number of transactions and volume are a better indicator of performance than is occupancy.

Figures 1.40-1.42, respectively, give the run time, effective gigaflops, and efficiency of our kernels. Figures 1.43-1.45 plot these metrics for 16384×16384 matrices. For the gigaflop rate computation, we used $2n^3 - n$ as the total number of floating point operations (multiplies and adds) done by a kernel and divided this by the run time. The efficiency was measured relative to the manufacturers stated peak rate of 933 GFlops for the C1060. That is we divided the gigaflop rate given in Figure 1.41 by 933 to come up with the efficiency given in Figure 1.42. It is interesting to note that, for $n = 16384$, there is a speedup of 14 when

	Block	Registers	Memory	Occupancy
$GPU1$	$(256, 1)$	10	44	1.00
$GPU2$	(8,8)	16	44	0.50
$GPU3$	$(4, 16)$	21	44	0.50
$GPU4$	$(16, 16)$	11	2092	1.00
$GPU5$	$(16, 16)$	11	2156	1.00
$GPU6$	$(16, 8)$	38	2220	0.38
$GPU7$	$(16, 8)$	38	2220	0.38
$GPU8$	$(16, 8)$	38	4204	0.38
sgemm	$(16, 4)$	35	1160	0.38

FIGURE 1.39: Number of registers per thread, shared memory (bytes) per block, and occupancy for GPU matrix multiply methods

	2048	4096	8192	16384
$GPU1$	0.71	5.24	40.85	327.77
$GPU2$	0.27	2.06	17.65	193.02
$GPU3$	0.27	2.05	16.28	130.25
$GPU4$	0.09	0.69	5.54	46.18
$GPU5$	0.08	0.60	4.84	44.12
$GPU6$	0.05	0.37	2.91	23.28
$GPU7$	0.05	0.37	2.91	23.26
$GPU8$	0.05	0.36	2.88	22.97
sgemm	0.05	0.37	2.97	23.70
CUBLAS	0.05	0.37	2.95	23.53

FIGURE 1.40: Run time (seconds) for GPU matrix multiply methods

	2048	4096	8192	16384
$GPU1$	24	26	27	27
$GPU2$	64	67	62	46
$GPU3$	64	67	68	68
$GPU4$	197	199	198	190
$GPU5$	226	229	227	199
$GPU6$	373	377	377	378
$GPU7$	373	377	378	378
$GPU8$	373	381	382	383
sgemm	358	368	371	371
CUBLAS	354	371	373	374

FIGURE 1.41: GFlops for GPU matrix multiply methods

going from the most simple adaptation of $SingleCoreIJK$ (i.e., $GPU1$) to the most efficient adaptation $GPU8$. Further, $GPU6$, $GPU7$, $GPU8$, *sgemm*, and CUBLAS all take about the same time with $GPU8$ being the fastest and *sgemm* the slowest of the five. $GPU8$ is faster than CUBLAS by 2% and faster than *sgemm* by 3% when $n = 16384$. This rather small reduction in run time relative to CUBLAS and *sgemm* despite the 15% reduction in number of transactions and 10% reduction in volume suggests that CUBLAS and *sgemm* do a very good job of masking device-memory latency. These five kernels are able to achieve a gigaflop rate of over 370GFlops when $n = 16384$. Since there is no opportunity to use SFs in a matrix multiply, the theoretical peak is actually only 622GFlops and not 933GFlops. Using 622GFlops as the theoretical maximum rate, the efficiency of the five kernels $GPU6$, $GPU7$, $GPU8$, *sgemm*, and CUBLAS becomes about 60%. In Figure 1.43, we show also the theoretical minimum time needed to move the required volume of data between device memory and the SMs. This theoretical minimum was computed by dividing the data volume by the theoretical peak transfer rate of 102GB/sec. Since we do not have the volume data for CUBLAS, the theoretical minimum data transfer time for CUBLAS is not shown.

Figures 1.46 and 1.47, respectively, show the run time of our GPU kernels as a function of the number of the device memory transactions and the volume of data transferred between device memory and the SMs. CUBLAS is not included in these figures as we do not have the transactions and volume data for CUBLAS. These figures show a fairly strong correlation between run time and transactions as well as between run time and volume.

	2048	4096	8192	16384
$GPU1$	0.03	0.03	0.03	0.03
$GPU2$	0.07	0.07	0.07	0.05
$GPU3$	0.07	0.07	0.07	0.07
$GPU4$	0.21	0.21	0.21	0.20
$GPU5$	0.24	0.25	0.24	0.21
$GPU6$	0.40	0.40	0.40	0.41
$GPU7$	0.40	0.40	0.41	0.41
$GPU8$	0.40	0.41	0.41	0.41
sgemm	0.38	0.39	0.40	0.40
CUBLAS	0.38	0.40	0.40	0.40

FIGURE 1.42: Efficiency of GPU matrix multiply methods

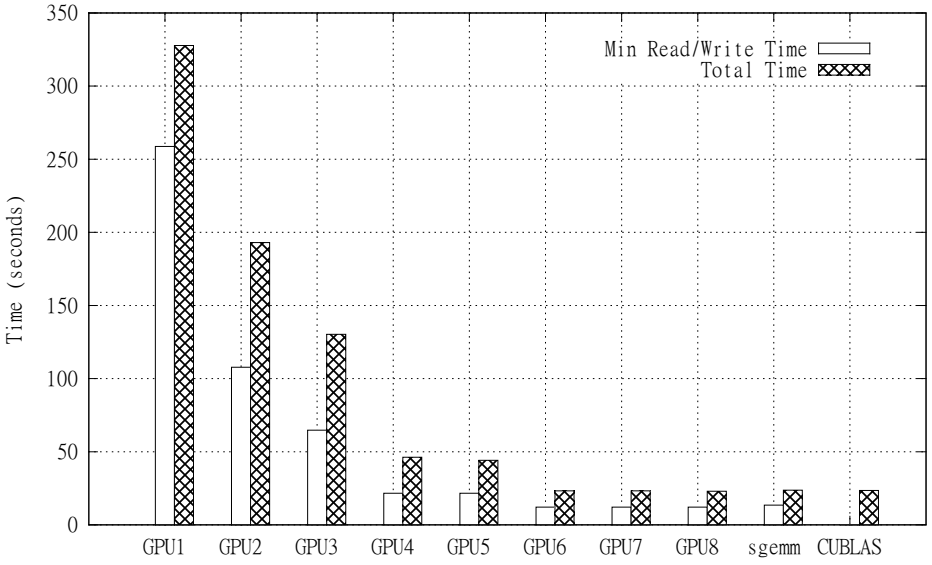

FIGURE 1.43: Run times (in seconds) for 16384×16384 matrices

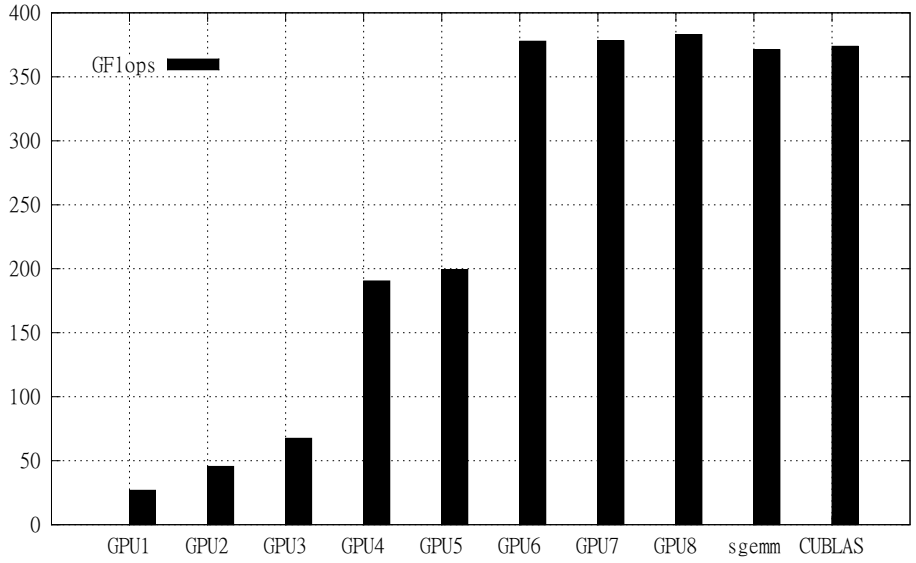

FIGURE 1.44: GFlops for 16384×16384 matrices

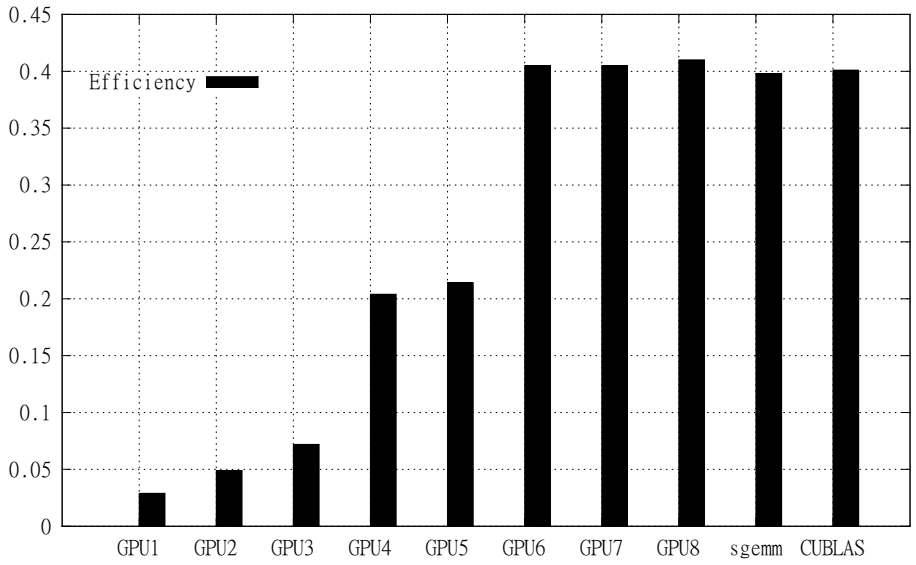

FIGURE 1.45: Efficiency for 16384×16384 matrices

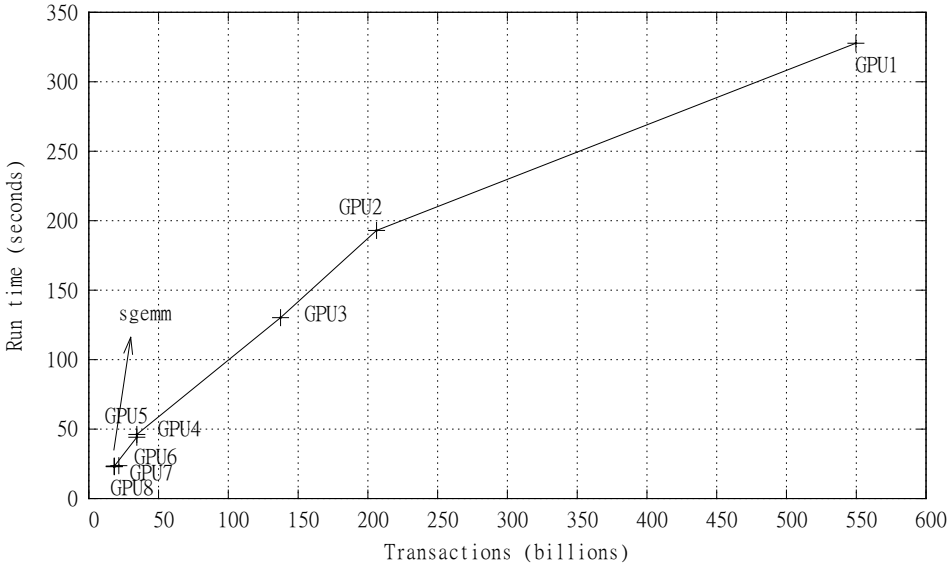

FIGURE 1.46: Time versus transactions for 16384×16384 matrices

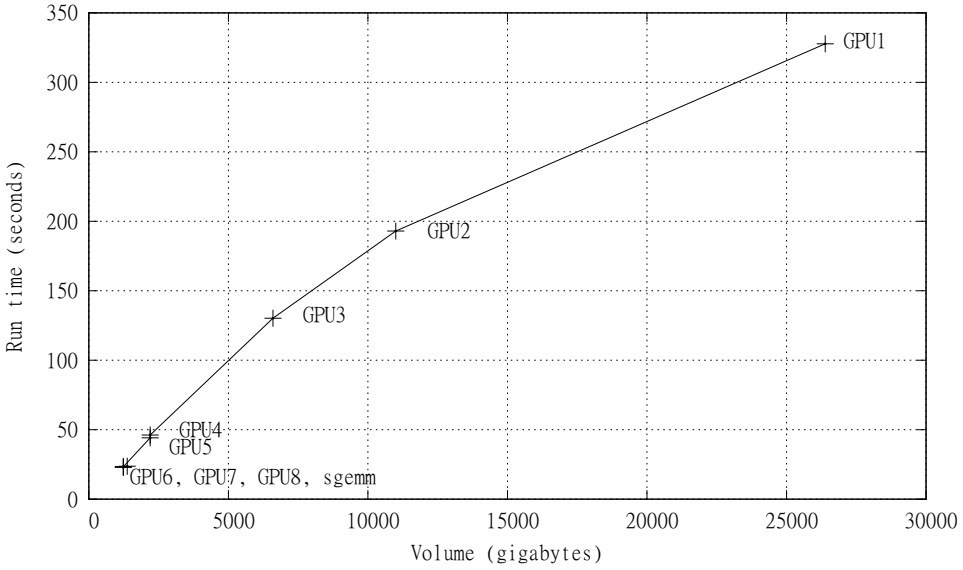

FIGURE 1.47: Time versus volume for 16384×16384 matrices

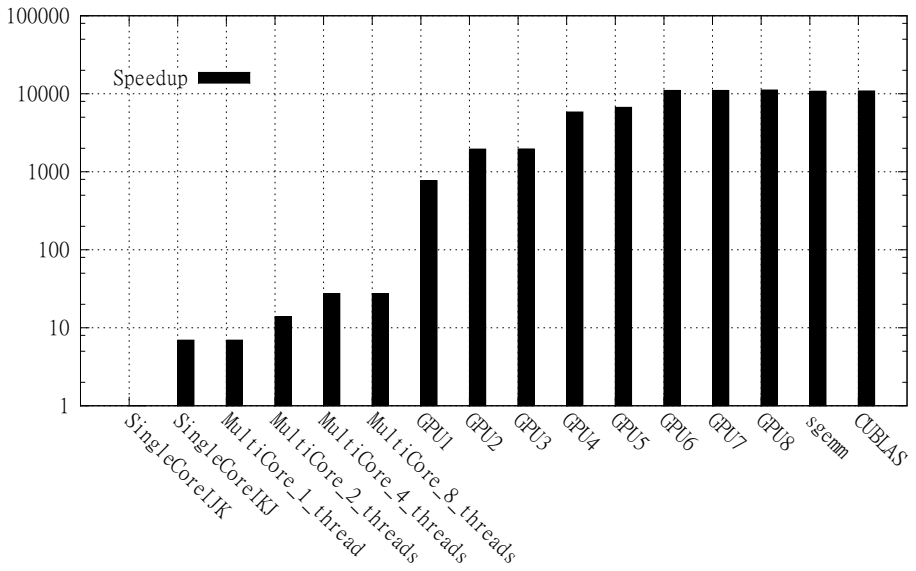

FIGURE 1.48: Speedup relative to $SingleCoreIJK$ for 4096×4096 matrices

Comparison With Single- and Quadcore Code

Because of the excessive time required to multiply large matrices on our Xeon quadcore host, we obtained run times for the single- and quadcore host codes only for $n \leq 4096$ (Figures 1.7 and 1.9). For $n = 2048$ and $n = 4096$, our fastest GPU code, $GPU8$ achieved speedups of 8998 and 11183 relative to $SingleCoreIKJ$ and 364 and 407 relative to the 4-thread OpenMP version of $SingleCoreIKJ$ (Figure 1.8). Figure 1.48 shows the speedup achieved by our various matrix multiply codes relative to $SingleCoreIJK$ for $n = 4096$. transactions (in billions) and the volume for 16384×16384 matrices.

Bibliography

[1] CUBLAS. Website: `http://developer.download.nvidia.com/compute/cuda/3\_0/toolkit/docs/CUBLAS\_Library\_3.0.pdf`.

[2] CUDA Occupancy Calculator. Website: `http://developer.download.nvidia.com/compute/cuda/CUDA\_Occupancy\_calculator.xls`.

[3] MAGMA BLAS. Website: `http://icl.cs.utk.edu/magma`.

[4] NVIDIA CUDA C Best Practices Guide, Version 3.2, 2010. Website: `http://developer.nvidia.com/object/gpucomputing.html`.

[5] NVIDIA CUDA Programming Guide, Version 3.0, 2010. Website: `http://developer.nvidia.com/object/gpucomputing.html`.

[6] NVIDIA Tesla. Website: `http://en.wikipedia.org/wiki/Nvidia_Tesla`.

[7] NVIDIA Tesla C2050. Website: `http://www.nvidia.com/object/product\_tesla\_C2050\_C2070\_us.html`.

[8] OpenMP. Website: `http://www.openmp.org/wp/`.

[9] S. Sahni. *Data Structures, Algorithms, and Applications in C++*. Silicon Press, NJ, USA, 2nd edition, 2005.

[10] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1 –10, 2009.

[11] Vasily Volkov and James W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press.

[12] Youngju Won and Sartaj Sahni. Hypercube-to-host sorting. *The Journal of Supercomputing*, 3:41–61, 1989.