In the proceedings of
IEEE International Parallel & Distributed Processing Symposium (IPDPS) 2008
(presented at IPDPS Workshop on Large Scale Parallel Processing (LSPP))

Faster Matrix-Vector Multiplication on GeForce 8800GTX

Noriyuki Fujimoto
Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama, Toyonaka, Osaka, 560-8531, Japan
fujimoto@ist.osaka-u.ac.jp

Abstract

Recently a GPU has acquired programmability to perform general purpose computation fast by running ten thousands of threads concurrently. This paper presents a new algorithm for dense matrix-vector multiplication on NVIDIA CUDA architecture. The experimental results on GeForce 8800GTX show that the proposed algorithm runs maximum 15.69 *(resp.,* 32.88*) times faster than the sgemv routine in NVIDIA's BLAS library CUBLAS 1.1 (resp., Intel Math Kernel Library 9.1 on one-core of 2.0 GHz Intel Xeon E5335 CPU with SSE3 SIMD instructions) for matrices with order* 16 *to* 12800*. The performance, including the data transfer between CPU and GPU, of Jacobi's iterative method for solving linear equations shows that the proposed algorithm is practical for some real applications.*

1 Introduction

Matrix-vector multiplication is a kernel routine of many numerical algorithms. Recently a GPU has acquired programmability [6] to run ten thousands of threads concurrently and some algorithms [7, 1, 2, 9] appeared for GPU to compute matrix-vector multiplication. For NVIDIA CUDA (Compute Unified Device Architecture) architecture [5], NVIDIA provides a BLAS library (called CUBLAS [4]) which includes a routine called *sgemv* for single precision dense matrix-vector multiplication. The observed performance of sgemv in CUBLAS 1.1 on GeForce 8800GTX [3] is high at about 36.41 GFLOPS at the maximum for matrices with order 1024 to 12800 (without data transfer time between CPU and GPU). However, the performance rapidly and extremely fluctuates with an increase of the order of a given matrix with fluctuation period of 16 and is under 4 GFLOPS at the minimum. Note that the minimum repeatedly appears several times within every period of 16.

This paper presents a superior algorithm for dense matrix-vector multiplication on NVIDIA CUDA architecture. Then, this paper empirically compares the proposed algorithm with sgemv in CUBLAS 1.1 and also sgemv in Intel Math Kernel Library (MKL for short) 9.1 on 2.0 GHz Intel Xeon E5335 CPU with SSE3 SIMD instructions, in terms of both one matrix-vector multiplication and Jacobi's iterative method for solving linear equations.

2 An Overview of CUDA Architecture

This section briefly illustrates CUDA architecture by the case of GeForce 8800GTX, mainly using excerpts from [5] and [3]. For more detail, see [5] and [3].

When programmed through CUDA, the GPU is viewed as a compute device capable of executing a very high number of threads in parallel. A *kernel* is a portion of an application that is executed on the GPU and can be isolated into a function, of C language, that is executed as many different threads.

A *multiprocessor* consists of 8 scalar processors with 16KB *shared memory* and a total of 8192 registers. CUDA architecture calls so-called VRAM *device memory*. 8800GTX is a two-level shared-memory parallel machine such that 16 multiprocessors are connected via 768MB device memory. So, 8800GTX equips with a total of 128 scalar processors. The maximum number of threads that can run concurrently on a multiprocessor is 768. Hence, 8800GTX can run maximum 12288 threads concurrently by hardware management.

A *thread block* (*block* for short) is a batch of threads that can cooperate together by efficiently sharing data through fast shared memory and their execution to coordinate memory accesses. The maximum number of threads per block is limited to 512. A block has dimensionality. Blocks of same dimensionality and size that execute the same kernel can be batched together into a *grid* of blocks. Threads in different blocks from the same grid cannot communicate and synchronize with each other. Each thread (resp., block) is

identified by its *thread ID* (resp., *block ID*), which is the thread (resp., block) number within the block (resp., grid). An application can also specify a block (resp., a grid) as a two- or three-dimensional (resp., two-dimensional) array of arbitrary size and identify each thread (resp., block) using a 2- or 3-component (resp, 2-component) index instead. Each block is allocated to a multiprocessor. Each multiprocessor can run concurrently maximum 8 blocks [5]. Each block is split into groups of 32 threads called *warps*. Each of these warps is executed by the multiprocessor in a SIMD fashion. A *thread scheduler* periodically switches from one warp to another. A *half-warp* is either the first or second half of a warp.

Shared memory is organized into 16 banks and can be accessed as fast as accessing a register as long as there are no bank conflicts between the threads in a half-warp. If two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized. However, no bank conflict occurs provided that all threads of a half-warp read from an address within the same 32-bit word.

To access device memory with high bandwidth, the device memory addresses simultaneously accessed by each thread of a half-warp during the execution of a single read or write instruction should be arranged so that the memory accesses can be coalesced into a single contiguous, aligned memory access. To issue one arithmetic or memory instruction (except several instructions) for a warp, a multiprocessor takes 4 clock cycles [5]. Device memory bandwidth is very high at 86.4 GB/sec [3]. However, when accessing device memory, there are 400 to 600 clock cycles of memory latency [5]D This device memory latency can be hidden by the thread scheduler if there are sufficient number of threads. The number of threads that run concurrently on a multiprocessor is restricted by the fact that shared memory and registers in a multiprocessor are divided among concurrent blocks allocated for the multiprocessor. NVIDIA recommends at least 192 threads per block and at least 2 blocks per multiprocessor [5].

CUDA supports a subset of the texturing hardware that the GPU uses for graphics to access *texture memory*. Although the shared memory and the device memory are not cached, the texture memory is cached. The texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture addresses that are close together will achieve best performance. So, reading data from texture memory instead of device memory can have performance benefits if access pattern has such locality.

3 The Proposed Algorithm

Listing 1 shows a naive implementation of matrix-vector multiplication in C language. During the multiplication,

Listing 1. naive matrix-vector multiplication in C

```
1  // y = Ax
2  // A : m-by-n matrix , x : n elements vector ,
3  // y : m elements vector
4
5  void mv(float *y, float *A, float *x, int m, int n) {
6    for (int i = 0; i < m; i++) {
7      y[i] = 0;
8      for (int j = 0; j < n; j++)
9        y[i] += A[i * n + j] * x[j];
10   }
11 }
```

each element in the matrix A is used only once. In contrast, the vector x is used for every row of A. So, reusable data in matrix-vector multiplication is x only. Therefore, x is desirable to be loaded into shared memory. However, even if we can use whole shared memory only for x, shared memory can load only 4096 floats. Hence, this paper considers a block algorithm for matrix-vector multiplication such that: The size of data required for computation of a block is small enough to fit shared memory; Blocks can be computed in parallel by multiprocessors; Computation for a block itself can be performed efficiently in parallel on a multiprocessor.

As a block algorithm that satisfies the above requirements, this paper proposes a parallel algorithm shown in Listing 2. Figure 1 illustrates the behavior of the parallel algorithm. In Listing 2, matrix A is divided into blocks with size 16×16. The construct "`forall ... do in parallel`" represents a parallel for-loop. The outer `forall` in line 10 corresponds to the parallelism among multiprocessors. The body of the outer `forall` corresponds to the computation in a multiprocessor. Notice that two inner `forall`s in line 14 and 20 imply the computation itself has parallelism. To parallelize matrix-vector multiplication shown in Listing 1, it is common to utilize parallelism only in line 6. In this case, the amount of available parallelism is m. In contrast, the proposed algorithm utilizes higher amount of parallelism. The dominant part of the proposed algorithm is line 14 to 19 in Listing 2. The amount of the available parallelism is $16m$. The dominant part can be efficiently implemented because it is embarrassingly parallel computations in terms of both inter-multiprocessor and intra-multiprocessor. Line 20 to 21 in Listing 2 can be implemented as a set of simultaneous independent reduction operations of 16 floats. The set is performed only once during whole matrix-vector multiplication. On the other hand, the complexity of the dominant part is $O(n)$ per thread. So, the larger the number of the column of A is, the less significant the complexity of the reduction part is. The reduction operation itself is not embarrassingly

Listing 2. A high-level description of the proposed algorithm for CUDA architecture

```
1  // y = Ax
2  // A : m-by-n matrix, x : n elements vector,
3  // y : m elements vector
4  // This code is illustrative. So, for simplicity,
5  // here, m and n are assumed to be a multiple of 16.
6  // This restriction is released later in
7  // the final CUDA code.
8
9  void mv(float *y, float *A, float *x, int m, int n) {
10   forall h (0 <= h < m / 16) do in parallel {
11     // Note that P, i, and j below are local to h
12     // respectively.
13     float P[16][16];
14     forall i, j (0 <= i, j < 16) do in parallel {
15       P[i][j] = 0;
16       for (int w = 0; w < n / 16; w++)
17         P[i][j] += A[(16 * h + i) * n + (16 * w + j)]
18                    * x[16 * w + j];
19     }
20     forall i (0 <= i < 16) do in parallel
21       y[16 * h + i] = the sum total of P[i][*];
22   }
23 }
```

parallel computations. However, it can be efficiently implemented here so that each reduction operation is performed within a multiprocessor with zero latency access to shared memory (i.e., without bank conflict) as shown in the next paragraph. Since the proposed algorithm reads from and writes to A, x, and y only in sequential access manner, all accesses to device memory can be coalesced.

Listing 3 shows the entire CUDA implementation of the proposed algorithm. Figure 2 illustrates the behavior of each thread in the proposed CUDA algorithm. For detail of CUDA library routines and language extensions for C/C++, see [5]. Listing 3 includes not only the kernel function `mv_kernel` but also the C interface function `mv` that invokes `mv_kernel` and performs preprocessing and postprocessing necessary for `mv_kernel`.

The body of the outer `forall` in Listing 2 can be implemented efficiently on CUDA architecture as shown in Listing 3. Line 33 to 76 (resp., 78 to 84) corresponds to the dominant part (resp., reduction part). Vector x is read via its copy `xs` on shared memory. The proposed implementation allocates 16×16 threads to a thread block which deals with a 16×16 submatrix of A. The data in x required to deal with a submatrix of A is 16 floats only. If x is naively read every 16 floats, then 240 threads of 256 threads in a thread block must be idle when the other 16 threads read 16 floats of x. So, the proposed implementation devises the read of x so that x is read every 256 floats. Another device is that matrix A is read via texture `texRefA`. As shown in Figure 2, access pattern for `A` has 2D spatial locality. So, reading `A` via texture improves the performance of the algorithm.

Since a thread block contains exactly 256 threads, at most three thread blocks can be active on a multiprocessor. So, each thread block can use shared memory at most 16/3 KB. The size of the shared memory required by a thread block (a little more than 2KB) is less than the limit. However, the kernel function of the proposed implementation uses 16 registers. So, only two thread blocks are active on a multiprocessor.

If m is not a multiple of 16, then `mv_kernel` is invoked as if m is the minimum multiple m' of 16 that is greater than m. In this case, `mv_kernel` computes a vector with length m' but returns only first m of the m' floats as a resultant vector. If n is not a multiple of 256, then let n' be the maximum multiple of 256 that does not exceed n. In this case, `mv_kernel` deals with n' columns first. Line 58 to 76 deal with remained (n mod 256) columns.

4 Experiments

This section compares the performance of the proposed algorithm with CUBLAS 1.1 and also MKL 9.1.025 running on a CPU. For each test, one core of 2.0 GHz Intel Xeon E5335 (FSB 1333MHz, L2 cache 8MB, quad cores) and NVIDIA GeForce 8800GTX was used. The used OS is Windows XP Professional with NVIDIA graphics driver Version 169.21. For CUDA and CUBLAS program compilation, Microsoft Visual Studio 2005 Professional Edition with optimization option /O2 and CUDA 1.1 SDK was used. For MKL program compilation, Intel C++ compiler 10.0.026 with optimization options /O2, /QaxT (directs the compiler to utilize SSE3 SIMD instructions and processor specific optimizations), and /Qfp-speculationfast was used.

4.1 One Matrix-Vector Multiplication

Figure 3 shows the performance of one matrix-vector multiplication with every matrix order from 16 to 12800. The performance was measured with NVIDIA's CUDA Profiler 1.1 for the CUBLAS program and the CUDA program. The first measurement for CUDA and MKL includes the overhead of the library initialization and so on. Therefore, for each point, the measurement was conducted 11 times consecutively and the average value except the first measurement was plotted.

The proposed algorithm is a little slower than CUBLAS 1.1 only in 19 cases where a given matrix order is in $\{s|7712 \le s \le 8064, s \text{ is a multiple of } 32\} \cup \{s|8096 \le s \le 8192, s \text{ is a multiple of } 16\}$. However, for all other 12766 cases, the proposed algorithm runs faster than CUBLAS 1.1. For a matrix with order at least 2048, the performance of CUBLAS significantly varies from 3.15 GFLOPS to 36.41 GFLOPS with a fluctuation period of 16.

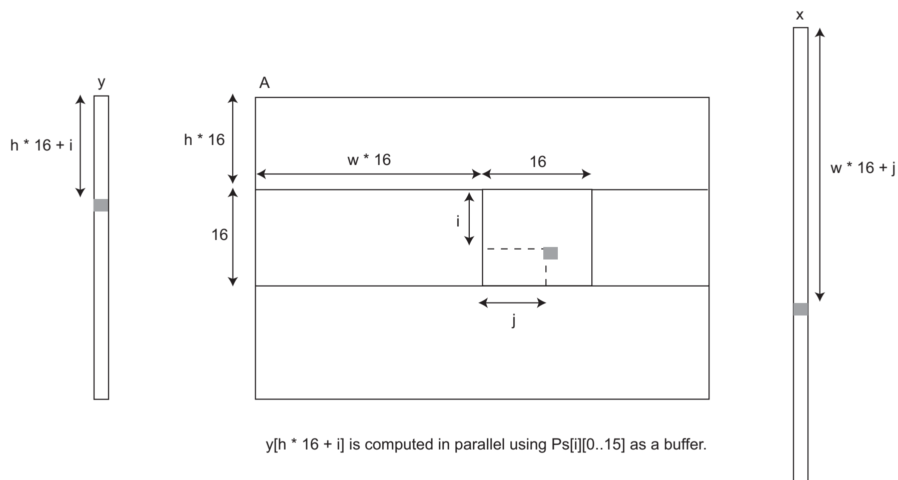


Figure 1. Behavior of the high level parallel block algorithm for matrix-vector multiplication

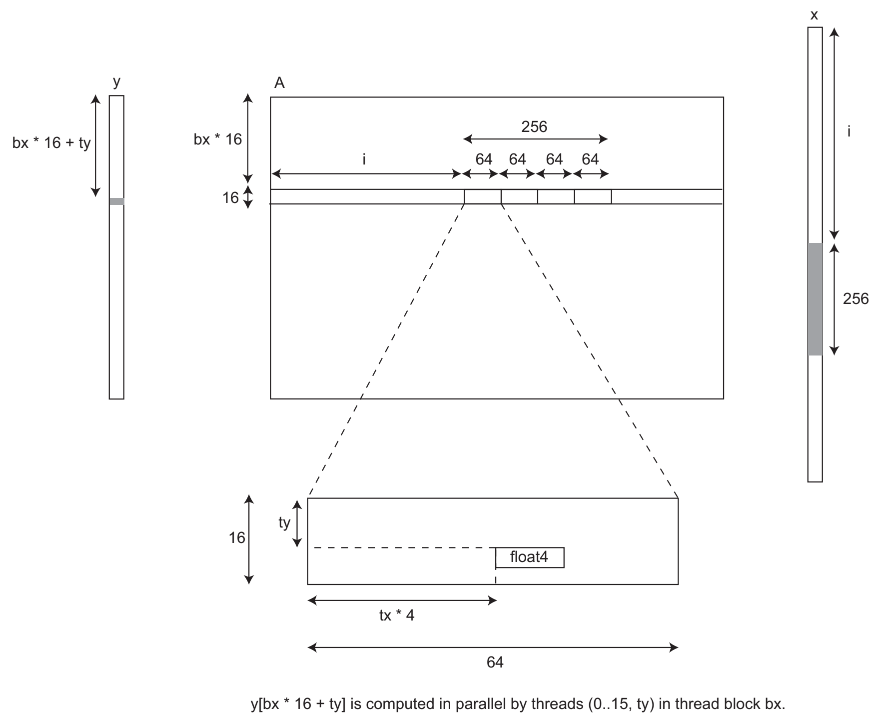


Figure 2. Behavior of each thread in the proposed CUDA algorithm for matrix-vector multiplication

If a given matrix order is a multiple of 16, then the performance of CUBLAS is maximum 36.41 GFLOPS. However, otherwise, the performance of CUBLAS is about 4.5 to 7.4 GFLOPS only. In particular, for half matrix orders, the performance of CUBLAS is only 4.5 GFLOPS. In contrast to this, the performance of the proposed algorithm is relatively stable and keeps above 28.05 GFLOPS for matrix order 2048 to 12800. The proposed algorithm has a fluctuation period of 512. The reason is as follows: If there are enough many blocks, each of 16 multiprocessors is allocated 2 blocks such that each block computes 16 rows of a given matrix; Therefore, blocks are evenly distributed among multiprocessors if the number of rows of a given matrix is a multiple of $512(= 16 \times 2 \times 16)$.

4.2 Jacobi's Iterative Method

Figure 4 shows experimental results on Jacobi's iterative method for solving linear equations $Ax = b$ where $A = (a_{ij})$ is a given n-by-n matrix, b is a given vector, and x is the solution vector of the equations. To accelerate the data transfer between CPU and GPU, the CUDA program and the CUBLAS program used page-locked memory to store A, x, and the temporary buffer which corresponds to y. The performance of CUBLAS sgemv significantly varies over a given matrix order. Figure 4 (b) and (d) (resp., (a) and (c)) show comparisons among the proposed algorithm, CUBLAS, and MKL in two of the best (resp., worst) cases for CUBLAS. The execution order of floating point operations in matrix-vector multiplication depends on the used algorithm. Therefore, the number of iterations may be different among CUBLAS, the proposed algorithm, and MKL. So, Figure 4 plots only the cases where the number of iterations are the same among the three. Each point is not the average value of plural executions but the value of only one execution. The measurement was consecutively conducted 20 times for each α and the results except the first measurement are plotted in Figure 4 to abandon the first measurement.

The execution time of Jacobi's method depends on a given matrix order and the number of iterations but does not **directly** depend on the values in a given matrix and a given vector (of course, the number of iterations depends on the values) . So, for each matrix order, a matrix and a vector are randomly generated subject to the following constraint. The number of iterations is controlled as follows: Let L, D, and U be a lower triangular matrix, a diagonal matrix, and an upper triangular matrix respectively such that $A = L + D + U$; Let H be $-D^{-1}(L + U)$; Jacobi's method is convergent iff every eigenvalue λ of H satisfies $|\lambda| < 1$ [8]; The smaller the maximum absolute value λ_{max} of the eigenvalues is, the faster the convergence speed of Jacobi's method is [8]; $\lambda_{max} \leq \sum_{j \neq i} |a_{ij}/a_{ii}|$ follows; Therefore, the smaller $\sum_{j \neq i} |a_{ij}/a_{ii}| (< 1)$ is, the faster the convergence speed of Jacobi's method is; Hence, A is generated to have random nondiagonal elements from $[0.0, 1.0]$ and diagonal elements $\alpha \sum_{j \neq i} a_{ij}$ where α is 1.1, 1.2, or 1.4. b is generated randomly from $[-0.5, 0.5]$.

Although the proposed algorithm is a little slower than CUBLAS in case of matrix order 8192, for other four cases the proposed algorithm is fairly faster than CUBLAS. CUBLAS and the proposed algorithm transfer a given matrix to GPU only once because each iteration of Jacobi's iterative method uses the same matrix. So, the performance is improved with an increase of the number of iterations and approaches the performance without the data transfer in Figure 3. For example, if the matrix order is 8192 and the number of iterations is at least 140, the overhead of data transfer between CPU and GPU is negligible. This implies that, if the proposed algorithm is used, GPU is useful for some applications that perform matrix-vector multiplications many times but perform the data transfer between CPU and GPU a few times.

5 Related Works

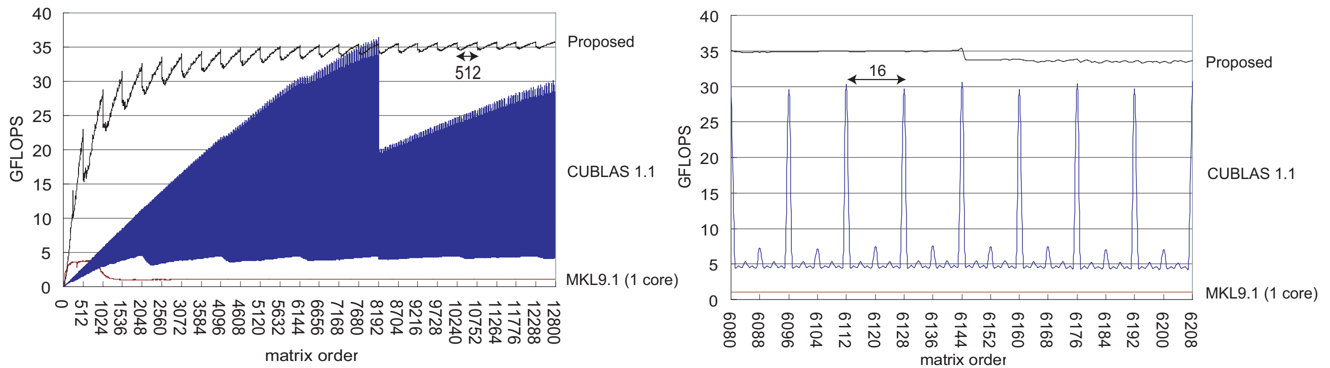
As far as the author knows, except CUBLAS sgemv, no result exist on dense matrix-vector multiplication for CUDA architecture (As for sparse matrix-vector multiplication for CUDA, there is a result [7]). However, some results are known for traditional GPGPU, which reduces general computations to a series of rasterization problems on a GPU using OpenGL or DirectX 3D. This section briefly summarizes these related works.

Krüger et al. [2] presented a dense matrix-vector multiplication algorithm using Pixel Shader 2.0 API of DirectX9. The experiments on Windows XP with a 2.8GHz Pentium 4 and an ATI 9800 graphics card showed that the performance (without data transfer between CPU and GPU) for matrices with order 512 to 2048 is about 12 to 15 times faster compared to an optimized CPU program (the detail on the CPU program is not described).

Buck et al. [1] proposed for GPUs a high level data-parallel programming language called Brook, which is based on stream computation model. The performance (without data transfer between CPU and GPU) of the Brook program (resp., the hand-written GPGPU program) for dense matrix-vector multiplication on Windows XP with 3 GHz Intel Pentium 4 and ATI Radeon X800 XT Platinum is 2.251 GFLOPS (resp., 2.335 GFLOPS) for a matrix with order 1024.

Tarditi et al. [9] proposed an even higher level data-parallel programming language called Accelerator, which is based on array computation model. The performance (without data transfer between CPU and GPU) of the Accelerator program, whose source code is translated into the code

CPU : Intel Xeon E5335 (Clock 2.0GHz, FSB 1333MHz, L2 cache 8MB)
GPU : NVIDIA GeForce 8800GTX



Note that the performance of CUBLAS 1.0 rapidly and extremely fluctuates with period of 16. So, the performance curve looks like painted area.

The above graph shows the enlargement of the left graph with matrix order 6080 to 6208.

Figure 3. Performance of matrix-vector multiplication (without data transfer between CPU and GPU)

of a virtual machine called CLR, (resp., the hand-written GPGPU program) for dense matrix-vector multiplication on Windows XP with a 3.2GHz Pentium 4 and an ATI X1800 graphics card is 4 times slower (resp., 1.3 times faster) than a CPU program optimized by Intel MKL 7.0 for a matrix with order 1000.

6 Conclusion

A new algorithm has been proposed for dense matrix-vector multiplication on NVIDIA CUDA architecture. On GeForce 8800GTX, the proposed algorithm runs maximum 15.69 (resp., 32.88) times faster than the sgemv routine in NVIDIA's CUBLAS 1.1 (resp., Intel MKL 9.1 on one-core of 2.0 GHz Intel Xeon E5335 CPU with SSE3 SIMD instructions) for matrices with order 16 to 12800. The performance, including the data transfer between CPU and GPU, of Jacobi's iterative method has showed that the proposed algorithm is practical for some real applications.

Acknowledgment

This research was supported in part by Grant-in-Aid for Young Scientists (B)(18700058) from the Japan Society for the Promotion of Science.

References

[1] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, August 2004.

[2] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics*, 22(3):908–916, 2003.

[3] NVIDIA. Technical brief: NVIDIA GeForce 8800 GPU architecture overview. In *http://www.nvidia.com/page/8800_tech_briefs.html*, November 2006.

[4] NVIDIA. CUDA CUBLAS library 1.1. In *http://www.nvidia.com/object/cuda_develop.html*, September 2007.

[5] NVIDIA. CUDA programming guide 1.1. In *http://www.nvidia.com/object/cuda_develop.html*, November 2007.

[6] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[7] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *Graphics Hardware 2007*, pages 97–106, August 2007.

[8] G. Strang. *Linear Algebra and Its Applications 3rd Edition*. Harcourt, Inc., 1988.

[9] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. In *12th international conference on Architectural support for programming languages and operating systems*, pages 325 – 335, 2006.

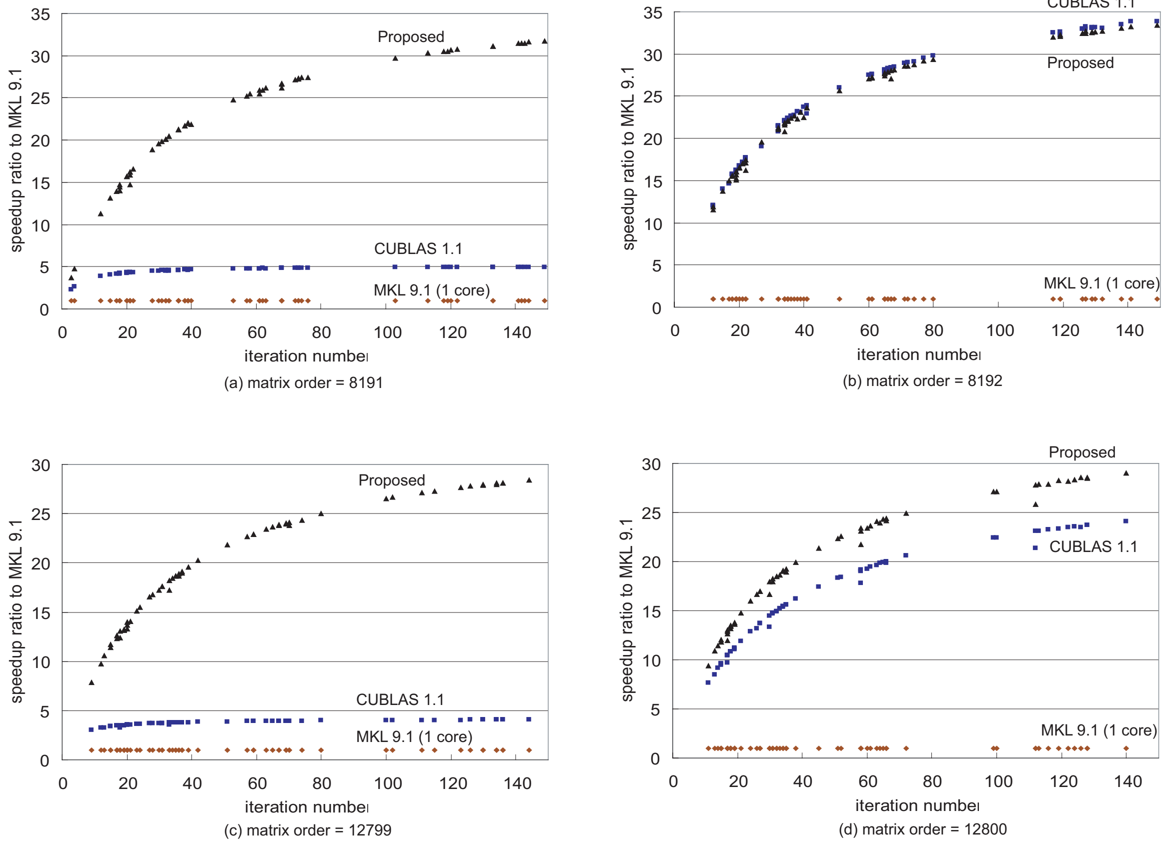


Figure 4. A comparison of whole execution time (including data transfer time between CPU and GPU) of Jacobi's iterative method for solving linear equations

Listing 3. The proposed matrix-vector multiplication algorithm in CUDA

```
// y = Ax
// A : m-by-n matrix, x : n elements vector, y : m elements vector
// m and n are arbitrary positive integers.

texture<float4, 2, cudaReadModeElementType> texRefA;

void mv(float *y, float *A, float *x, int m, int n) {
    int blkNum = (m >> 4) + ((m & 15) ? 1 : 0); int height = blkNum << 4;
    int width = (n & 255) ? (256 * ((n >> 8) + 1)) : n;
    dim3 threads(16, 16), grid(blkNum, 1);
    cudaArray *d_A; float *d_x, *d_y;

    cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float4>();
    cudaMallocArray(&d_A, &channelDesc, width >> 2, height);
    cudaMemcpy2DToArray(d_A, 0, 0, A, n * sizeof(float), n * sizeof(float), m, cudaMemcpyHostToDevice);
    cudaBindTextureToArray(texRefA, d_A);
    cudaMalloc((void**) &d_x, n * sizeof(float));
    cudaMalloc((void**) &d_y, m * sizeof(float));

    cudaMemcpy(d_x, x, n * sizeof(float), cudaMemcpyHostToDevice);
    mv_kernel<<< grid, threads >>>(d_y, d_A, d_x, m, n);
    cudaMemcpy(y, d_y, m * sizeof(float), cudaMemcpyDeviceToHost);

    cudaFree(d_y); cudaFree(d_x); cudaUnbindTexture(texRefA); cudaFreeArray(d_A);
}

#define bx blockIdx.x
#define tx threadIdx.x
#define ty threadIdx.y
__global__ void
mv_kernel(float* y, cudaArray* A, float* x, int m, int n)
{
    __shared__ float xs[16][16];
    __shared__ float Ps[16][16];
    float4 a;
    float *Psptr = (float *) Ps + (ty << 4) + tx;
    int ay = (bx << 4) + ty;
    float *xptr = x + (ty << 4) + tx;
    float *xsptr = (float *) xs + (tx << 2);

    *Psptr = 0.0f;
    int i;
    for (i = 0; i < (n & ~255); i += 256, xptr += 256) {
        xs[ty][tx] = *xptr;
        __syncthreads();
        int ax = tx + (i >> 2);
        a = tex2D(texRefA, ax     , ay);
        *Psptr += a.x * *xsptr         + a.y * *(xsptr +   1) + a.z * *(xsptr +   2) + a.w * *(xsptr +   3);
        a = tex2D(texRefA, ax + 16, ay);
        *Psptr += a.x * *(xsptr +  64) + a.y * *(xsptr +  65) + a.z * *(xsptr +  66) + a.w * *(xsptr +  67);
        a = tex2D(texRefA, ax + 32, ay);
        *Psptr += a.x * *(xsptr + 128) + a.y * *(xsptr + 129) + a.z * *(xsptr + 130) + a.w * *(xsptr + 131);
        a = tex2D(texRefA, ax + 48, ay);
        *Psptr += a.x * *(xsptr + 192) + a.y * *(xsptr + 193) + a.z * *(xsptr + 194) + a.w * *(xsptr + 195);
        __syncthreads();
    }

    if (i + (ty << 4) + tx < n) {
        xs[ty][tx] = *xptr;
    }
    __syncthreads();
    int j;
    for (j = 0; j < ((n - i) >> 6); j++, xsptr += 61) {
        a = tex2D(texRefA, tx + (i >> 2) + (j << 4), ay);
        *Psptr += a.x * *xsptr++ + a.y * *xsptr++ + a.z * *xsptr++ + a.w * *xsptr;
    }
    __syncthreads();
    int remain = (n - i) & 63;
    if ((tx << 2) < remain) {
        a = tex2D(texRefA, tx + (i >> 2) + (j << 4), ay);
        *Psptr += a.x * *xsptr++;
    }
    if ((tx << 2) + 1 < remain) *Psptr += a.y * *xsptr++;
    if ((tx << 2) + 2 < remain) *Psptr += a.z * *xsptr++;
    if ((tx << 2) + 3 < remain) *Psptr += a.w * *xsptr;
    __syncthreads();

    if (tx < 8) *Psptr += *(Psptr + 8);
    if (tx < 4) *Psptr += *(Psptr + 4);
    if (tx < 2) *Psptr += *(Psptr + 2);
    if (tx < 1) *Psptr += *(Psptr + 1);

    __syncthreads();
    if (ty == 0 && (bx << 4) + tx < m) y[(bx << 4) + tx] = Ps[tx][0];
}
```