Abstract—The paper presents results of several experiments evaluating the performance of NVIDIA processors, implementing a new Tesla architecture, in matrix-vector multiplication. Three matrix forms, dense, banded and sparse, are considered together with three hardware platforms: NVIDIA Tesla C870 computing board, NVIDIA GeForce 8800 GTX graphics card and one of the newest Intel Xeon processors, E5462, with 1.6 GHz front side bus speed. The conclusions from experiments indicate what speed-ups can be expected when, instead of standard CPUs, accelerators in the form of presented GPUs are used for considered computational kernels.

I. MOTIVATION

THE USE of graphics processing units (GPUs) in scientific computing is becoming an accepted alternative for calculations employing traditional CPUs [1]. The characteristics of GPUs, especially parallel execution capabilities and fast memory access, render them attractive in many application areas. One of the most important application domains is numerical linear algebra. The computational kernels from linear algebra are used in many scientific codes. Hence the widespread interest in porting and testing such kernels for GPUs [2].

The purpose of the present article is to assess the performance of the recent NVIDIA GPUs in performing one of linear algebra kernels, namely matrix-vector product. This kernel plays an important role in the implementation of iterative solvers for systems of linear equations. Moreover it is a typical memory-bound operation—its performance depends mainly on the speed of communication between a processor and memory chips, much less on the processing capabilities of the processor itself.

The organization of the paper is the following. In the next section performance characteristics of NVIDIA GPUs are described and compared to characteristics of typical contemporary CPUs. Section III presents the matrix formats considered in the paper and the corresponding matrix-vector multiplication algorithms. In Section IV the set-up of experiments as well as tests’ results are described. Finally, conclusions are drawn in Section V.

II. PERFORMANCE CHARACTERISTICS OF CPUs AND GPUs

A typical contemporary processor is a two- or four-core unit equipped with a memory hierarchy comprised of several layers of cache and the main memory. From the point of view of performance for scientific codes two parameters are of premium importance: the processing speed and the speed of data transfer from the memory.

The processing speed depends on the number of cores, clock frequency and the number of instructions completed in every clock cycle. This last number varies greatly depending on the application. The theoretical maximum is usually two to four instructions per cycle. The practical performance can be closed to the maximum whenever “the memory wall” is not hit, i.e. the processor gets all necessary data on time. This is the case for BLAS Level 3 routines, like e.g. matrix-matrix product, on which the direct solution of systems of linear equations is usually based.

There is a different situation with memory bound algorithms. If the processor cannot get the necessary data on time the performance can be equal to a small percentage of the maximum.

Graphics processing units differs significantly from general purpose CPUs in both aspects affecting performance. Their processing speed is much greater due to the large number of specialised cores (though usually operating at lower frequencies that CPU cores). Also the throughput to the memory is greater for GPUs due to usually wider buses then that of CPUs. Hence both, processing speed limited and memory limited algorithms can benefit from off-loading to GPUs. One of serious drawbacks of contemporary GPUs is the use of single precision floating point numbers only. However, today all major producers of GPUs aiming at general purpose computing start offering double precision floating point capabilities in their products, so this limitation should shortly be overcome.

A. An example CPU

Let us take, as an example CPU, one of the newest Intel Quad-core Xeon processors, E5462 with four cores, 2.8 GHz...
Algorithm 1 Simple matrix-vector product, $y := A \cdot x$, for dense matrices stored column-wise

```c
for (i = 0; i < n; i++) {
    t = x[i];
    for (j = 0; j < n; j++){
        y[j] += A[i+n+j] * t;
    }
}
```

clock speed and four double precision floating point instructions per clock cycle (maximum). The theoretical peak floating point performance is hence 44.8 GFlops. The processor is equipped with 12 MB cache memory and 1.6 GHz front side bus.

Let us consider a simple example of matrix-vector multiplication, a BLAS Level 2 operation that is memory bound. For a square matrix of size $n$ the number of floating point operations performed is $2n^2$. The number of memory accesses depends on the details of the algorithm. Let us consider a simple algorithm for column-wise stored matrix $A$ (Algorithm 1 on page 286). The minimal number of memory accesses (e.g. for small matrices fitting in cache together with vectors $x$ and $y$) is equal to $n^2 + 3n$ (one read for every element of $A$, $x$ and $y$ and one write for every element of $y$). The maximal number of memory accesses (e.g. for large matrices when at the end of the inner loop the first elements of $y$ are no longer in cache memory of any level) is $3n^2 + n$ (one read for every element of $A$, $n$ reads and $n$ writes for each element of $y$ and one read for every element of $x$). The number of memory accesses per one floating point operation varies, hence, between $\frac{1}{2} + \frac{1}{2n}$ and $\frac{1}{2} + \frac{2}{2n}$.

The theoretical peak throughput to the memory for the E5462 processor is 12.8 GB/s (1.6 GHz FSB speed combined with 8 bytes wide bus). This translates to 1.6 billions double precision numbers supplied to the processor per second. For the considered matrix-vector product algorithm this means (taking into account the number of memory accesses to the number of operations ratio and neglecting the terms with $n$ in the denominator) the constraint on the performance at the level between 3.2 Gflops and 1.067 Gflops. These constitute only 7.14% and 2.38%, respectively, of the theoretical peak performance of the processor.

B. An example GPU

Graphics processors have evolved in recent years from special purpose devices, with a sequence of fixed-function elements performing subsequent phases of the standard graphics pipeline, to processors that, although keeping many of special purpose solutions, can serve general purpose computations. NVIDIA, being the company that introduced the first GPU in 1999, has recently developed a special architecture that unifies the hardware for different stages of graphics processing and at the same time offers general purpose computing capabilities. The Tesla architecture is depicted in Fig. 1 (taken from [3], from which also the presented below facts about the Tesla architecture are taken).

From the general computing point of view the important elements of the architecture are the following:

- **Host interface and Compute work distribution**—the elements responsible for getting instructions and data from the host CPU and its main memory; they also manage the threads of execution by assigning groups of threads to processor clusters and performing context switching
- **TPC**—texture/processor cluster, a basic building block of Tesla architecture GPUs (a single GPU can have from one to eight TPCs)
- **SM**—streaming multiprocessor, a part of TPC, consisting of eight streaming processor cores, two special function units (for performing interpolation and approximate evaluation of trigonometric functions, logarithms, etc.), a multithreaded instruction fetch and issue unit, two caches and a pool of shared memory
- **Texture unit**—each unit is shared by two SMs (each TPC contains two SMs and one texture unit), the unit plays its part in graphics calculations and is equipped with L1 cache memory accessible from SPs
- **Level 2 Cache memory units**—connected through fast network to TPCs (ROP—raster operation processors usually do not take part in general purpose computations)

The most important characteristic of Tesla processing is that streaming multiprocessors manage the execution of programs using so called "warps", groups of 32 threads. All threads in a warp execute the same instruction or remain idle (in such a way different threads can perform branching and other forms of independent work). Warps are scheduled by special units in SMs in such a way that, without any overhead, several warps execute concurrently by interleaving their instructions. So it is possible that, for example, with two warps the execution looks as follows: warp_1-instruction_1, warp_2-instruction_1, warp_1-instruction_2, warp_1-instruction_3, warp_2-instruction_2, etc. In such a way each SM can manage up to 24 warps, i.e. 764 threads. From the point of view of writing application codes it is, however, important to take into account the organization of the work, i.e. the use of 32 threads simultaneously. The code that does not break into 32 thread units can have much lower performance.

The processing capabilities of Tesla GPUs derive from many sources. Some of them, like texture and rasterization units, serve mainly the purposes of graphics processing. For general processing, especially scientific array operations, the main units are streaming processor cores (SP) and special function units (SFU). Each core can perform two floating point operations per cycle (by the multiply-add unit), each SFU is equipped with four multipliers, hence can perform 4 instructions per cycle. This gives 16 operations per cycle for 8 SPs and 8 operations per cycle for two SFUs in a single streaming multiprocessor. If an application can make both types of units work simultaneously (which is possible theoretically) it can achieve 24 operations per cycle per SM,
that for 1.5 GHz GPU with 16 SMs gives 576 Gflops. The performance possible to obtain in real applications is, however, much smaller.

One of reasons for smaller performance can be again limitations caused by too slow memory access. The memory system of Tesla architecture consists of even more layers than that of a CPU. Apart from the main memory of the host system, there is a DRAM card memory and two pools of cache memory - one within SMs, called shared memory, and the second within texture units. Both caches can be utilized by scientific codes. Important feature of the architecture is that the processor is not a vector unit, each thread issues its own memory requests. However, the performance is best when requests can be blocked by memory management unit to access contiguous memory areas.

The DRAM memory of a card consist of up to six modules, that comprise the whole physical memory space of a GPU. Each module is connected to the GPU by a link 64-bits wide. Combined with the DDR technology and clock frequency in the range of 1GHz this gives the memory throughput around 100 GB/s. This is four times faster than the theoretical memory throughput of the example Xeon CPU considered earlier. In our simple example of matrix-vector multiplication this translates to the speed between 37.5 and 50 Gflops, but now for single precision numbers only. Still this constitutes only less than 10% of the theoretical GPU performance.

C. CUDA programming model

CUDA programming model is a model for programming Tesla GPUs using some extensions of standard C/C++. The extensions include two main domains: parallel work organisation through concurrent threads and memory accesses making use of the memory hierarchy of Tesla architecture.

Threads in the CUDA model are grouped into so called thread blocks. A programmer selects the block size. All threads in a block executes on one SM and can use its shared memory. Threads in one block can communicate with each other using the shared memory. Threads in different blocks cannot communicate. This constraint has also some positive consequences since scheduling of different blocks is fast and flexible (independent of the number of SMs used for program execution).

Apart from shared memory variables, the programmer can explicitly address two other types of variables. Local variables reside in DRAM card’s memory and are private for every thread. Global variables also reside in DRAM card’s memory and are accessible to threads from different blocks, thus providing i.e. a way for global thread synchronisation. The DRAM memory is, however, much slower than the on-chip shared memory. For this reason threads within blocks can synchronise using a special instruction implemented using the shared memory.
Writing programs in CUDA consist in writing two types of routines. Standard routines are executed on CPU. From these routines kernel procedures are invoked, to be executed by the GPU. Kernels are written as procedures (declared with a keyword __global__) for one thread. Parallel execution is obtained by specifying in the invocation, using a special syntax, the number of thread blocks and the number of threads in each block. For our example of matrix-vector product the calling sequence in a program executed by CPU might look as follows:

```c
dim3 threads( BLOCK_SIZE )
dim3 grid( n/BLOCK_SIZE );
mv_kernel<<< grid, threads >>>
( A_G, x_G, y_G, n );
```

where threads is a special variable of type dim3 that has three components, each for one dimension of a thread block. Dimensions are used for identifying thread blocks within blocks (in our example each thread is uniquely tagged by a triple threads.x, threads.y, threads.z, the first one within the range (0, BLOCK_SIZE − 1), the two latter equal to 0 by default, since the declaration of threads omitted them. Thread blocks form a grid of blocks—the dimensions of grids are specified similarly to dimensions of blocks and are used to identify blocks within grids (using triples grid.x, grid.y, grid.z). Thread and block identifiers are accessible within the kernel code through variables threadIdx and blockIdx.

The number of threads executing the mv_kernel is fixed and implied by dimensions of threads and grid arrays—in our example it was assumed that the number of threads is equal to n and that n is divisible by BLOCK_SIZE (if not simple adjustment has to be made).

For GPU threads to access variables in the GPU’s DRAM memory (A_G, x_G, y_G in our example) they must be first allocated there and then their values transferred from the host memory, both operations done using special provided functions before kernel invocation.

A simple kernel function for implementing matrix-vector product on the Tesla architecture is given as Algorithm 2 on page 288.

```
Algorithm 2 CUDA version of a simple matrix-vector product algorithm
__global__ void mv_kernel( float* A_G, float* x_G, float* y_G, int n)
{
    int j;
    int i = blockIdx.x * BLOCK_SIZE + threadIdx.x;
    float t = 0.0f;
    for ( j = 0; j < n; j ++ ) {
        t += A_G[i + j*n] * x_G[j];
    }
    y[i] = t;
}
```

Such an arrangement is done in the implementation of the standard SGEMV BLAS routine provided in the CUDA SDK [4], cublasSgemv. Blocks of threads are 1-dimensional with 128 threads per block and additionally fast shared memory is used for storing parts of vector x.

In [5] certain improvements to the original cublasSgemv algorithm have been proposed, consisting of using a texture memory unit to access elements of matrix A_G and grouping threads into 2-dimensional blocks of a fixed size 16x16.

**B. Banded matrices**

For special types of matrices (banded, sparse, multi-diagonal, etc.) special algorithms can be used to exploit possible performance improvements. For all types of matrices the main indication is to perform operations only on non-zero elements. Despite the obvious intuitive advantage of such an approach the practical improvements may depend upon such details as the number of zeros in a matrix or influence of the memory access pattern on the performance.

In this and the next section two types of matrices frequently used in scientific computing are considered, banded matrices and sparse matrices stored in CRS (compressed row storage) format.

As an algorithm for banded matrices the implementation of SGBMV BLAS in CUDA SDK has been considered. The algorithm is very similar to the implementation of SGEMV in cublasSgemv. The main difference lies in accessing the elements of matrix A_G. As required by BLAS standards the matrix is stored columnwise, but with diagonals stored in rows. Hence, the matrix-vector product for the matrix A_B that stores entries of A_G in the required format and for a single thread that performs operations on a single row is shown, with some simplifications, as Algorithm 3 on page 289.

In the algorithm, b denotes bandwidth. b is the most important parameter determining the performance. For small values of bandwidth different threads still can use entries of A_G from cache but the values of x_G are different. The complex addressing of A_G may also pose problems for processing cores, since, unlike standard superscalar CPUs,
Algorithm 3 Simple CUDA matrix-vector product algorithm for banded matrices

```cpp
__global__ void mvb_kernel(float *A_G, float *x_G, float *y_G, int n, int b) {
    int j;
    int i = blockIdx.x * BLOCK_SIZE + threadIdx.x;
    int begin = max(0, i - b);
    int end = min(n, i + b + 1);
    float t = 0.0f;
    for (j = begin; j < end; j++) {
        t += A_G[i + j * (2 * b + 1) - j + b] * x_G[j];
    }
    y[i] = t;
}
```

Algorithm 4 Simple matrix-vector product algorithm for CRS storage format

```cpp
void mv_crs(float *VA, int *JA, int *IA, int n, float *x, float *y) {
    int size = IA[i + 1] - IA[i];
    float sdot = 0.0f;
    for (int j = 0; j < size; j++) {
        sdot += VA[j] * x[JA[j]];
    }
    y[i] = sdot;
}
```

they cannot perform integer and floating point operations in parallel.

C. Sparse matrices

Sparse matrices are the most common form of matrices arising in approximations of partial differential equations describing different scientific and engineering problems. Hence, the importance of providing an efficient implementation of matrix-vector product, on which many iterative methods for systems of linear equations, like e.g. conjugate gradient or GMRES, can be based.

The CRS format [6] stores all non-zero entries of a matrix in a single one-dimensional vector, say VA, together with two other vectors: one, JA, with the same dimension as VA for storing column indices of the corresponding entries of the matrix and the other, say IA, with the dimension equal to the number of rows, for storing the indices of the first entries of subsequent rows in the array VA. A simple implementation of matrix-vector multiplication for CRS storage is shown as Algorithm 4 on page 289.

A CUDA implementation of Algorithm 4 has been proposed in [7]. The algorithm is shown as Algorithm 5 on page 289.

Algorithm 5 Simple CUDA matrix-vector product algorithm for CRS storage format

```cpp
__global__ void mv_crs_kernel(float *VA, int *JA, int *IA, int n, float *x, float *y) {
    __shared__ float cache[BLOCK_SIZE];
    int begin = blockIdx.x * BLOCK_SIZE;
    int end = blockIdx.x + BLOCK_SIZE;
    int row = blockIdx.x + threadIdx.x;
    int col;
    if (row < n) cache[threadIdx.x] = x[row];
    __syncthreads();
    if (row < n) {
        int r_b = IA[row];
        int r_e = IA[row + 1];
        float sum = 0.0f;
        float xj;
        for (col = r_b; col < r_e; col++) {
            int j = IA[col];
            if (j >= begin && j < end)
                xj = cache[j - begin];
            else
                xj = x[j];
            sum += VA[col] * xj;
        }
        y[row] = sum;
    }
```

Again, one thread performs operations on a single row and elements of vector x are cached for better performance (the operation __syncthreads performs fast synchronization of threads belonging to a single block).

We propose a modification to Algorithm 5 consisting in using texture memory for elements of VA and JA. In Algorithm 5 the proper elements of both arrays are first put in texture memory (variable `texture` in the code) and than used by the corresponding threads. The update phase of the algorithm looks then as follows:

```cpp
for (col = r_b; col < r_e; col++) {
    int j = texture.y;
    if (j >= blockIdx.x & & j < blockIdx.y)
        xj = cache[j - blockIdx.x];
    else
        xj = x[j];
    sum += texture.x * xj;
}
y[row] = sum;
```

IV. EXPERIMENTS

A. Hardware setup

We tested the discussed algorithms on two platforms. The first was Intel Xeon E5335, 2.0GHZ, with NVIDIA Tesla C870...
as a CUDA device and ATI Radeon HD 2600 XT as a display device. The second was Intel Core 2 Duo E6750, 2.66 GHz, with ASUS EN8800GTX (with NVIDIA GeForce 8800GTX GPU as a CUDA and a display device). Both machines were running Fedora Core 8, 64-bit, with NVIDIA graphic driver version 177.13. The algorithms (CUDA and CUBLAS) presented previously, were implemented with CUDA SDK version 2.0 beta. Note that, this version is the first that supports Fedora Core 8. For compilation we used gcc-compiler with standard optimization options (-O).

We tested the performance of matrix-vector product for three different matrix format representations: dense, sparse and banded, and measured the times for: allocating memory on graphic board, sending data from main memory to GPU memory, computing matrix-vector multiplication using card’s DRAM and also getting results from GPU memory to main memory. All timing was performed by the functions provided by the CUDA environment.

B. Performance

1) Dense matrices: We present first the results of two matrix-vector product algorithms for dense matrices residing already in GPU DRAM. The performance of cublasSgemv provided with CUDA SDK and the algorithm proposed in [5], denoted by sgemv_impr, is compared in Fig. 2 for matrices of dimension being a multiplicity of 32. Algorithm sgemv_impr presents more uniform performance and, moreover, does not exhibit a sudden break in performance for matrices of dimension not being the multiplicity of 32 (the effect reported in [5] which was also reproduced in our experiments). Nevertheless, the performance of any of the presented algorithms does not increase beyond 30 Gflops i.e. approx. 6% of the theoretical peak performance of the GPU (the theoretical limit derived from card’s DRAM memory speed was between 37 and 50 Gflops).

For algorithm sgemv_impr we present also, in Figures 3–5, the timings for allocating data on a GPU and bandwidths for transporting data to and from GPU DRAM, both operations related to performing matrix-vector product on a GPU from a program running on a CPU (similar results were obtained for different types of matrices). The last figure, Fig. 6, shows the final performance of the matrix-vector operation performed on a GPU, taking into account the time necessary to transfer data to and from GPU.

The timings and performance presented in Figures 3–6 show that the overall performance is determined almost exclusively by the time for transferring the entries of a matrix to GPU memory (for a matrix of size 10000 the transfer takes approx. 300 ms while all the other operations approx. 30 ms). The performance in Fig. 6 is slightly lower than that of a CPU (using the implementations from Intel MKL library we obtained 0.76 Gflops for Intel Xeon E5335 and 0.88 Gflops for Intel Xeon E5462). Hence, for scientific codes, especially iterative solvers employing matrix-vector product it is important how
many iterations will be performed on a given matrix, once it is transferred to the GPU memory.

2) Banded matrices: The results of experiments for banded matrices and the \texttt{cublasSgbmv} algorithm are presented in Fig. 7. The matrices had dimensions from 1000 to 100000 and the bandwidth was proportional to the matrix dimension (it was equal to 1.64% of matrix size to make the memory requirements for the largest matrices similar to memory requirements in dense matrix-vector product tests). A dramatic decrease in performance as compared to dense matrices can be observed. Moreover, the performance is far from uniform and strongly depends on the matrix size.

3) Sparse matrices: Two algorithms were tested in matrix-vector multiplication for sparse matrices stored in CRS format: the algorithm proposed in [7] (Algorithm 5 on page 289) and its modification proposed in the current paper. The setting was similar to the setting for banded matrices, the dimensions of matrices changed from 1000 to 100000, but now the number of non-zero entries was equal to 1.64% of the matrix size. The results are presented in Fig. 8. Once again, a decrease in performance occurred. The performance diminished with the increasing matrix size, as a result of more and more cache misses, since the non-zero entries of matrices occupied arbitrary positions in rows.

V. Conclusion

Tesla architecture is still a new and evolving concept. For certain applications it can bring speed ups in the range of tens as compared to CPUs [3]. For the considered in the current paper problem of performing matrix-vector multiplication, the performance of cards with processors implementing the Tesla architecture varies significantly depending on the form of matrices as well as the overall context of calculations (e.g. whether the time necessary for transferring a matrix to GPU memory is taken into account or not). Recent experiences show that algorithms proposed for matrix-vector product, e.g. in CUDA SDK, still can be improved. One of slight modifications to published algorithms for sparse matrices in CRS format has been presented in the paper. For some matrix sizes it brings performance increase. We plan to investigate it further and report on the results in forthcoming papers.

Acknowledgment

The authors would like to thank Mr Stanisław Rejowski and Mr Jacek Szwarz from the Action Company for providing the hardware on which the experiments were performed.

References


