CHAPTER

28 Jacket: GPU Powered MATLAB Acceleration

Torben Larsen, Gallagher Pryor, and James Malcolm

This chapter describes Jacket, which is a software platform for offloading MATLAB computation onto NVIDIA CUDA-capable GPUs. The objective of Jacket is to combine the productivity and simplicity of the MATLAB environment and M-language with the raw computational power of GPUs. We describe the ideas behind Jacket and its basic use, and we provide a simple example that compares MATLAB, CUDA C, and Jacket with respect to code and computational performance. Further, we include a performance evaluation of various Jacket features including CPU-GPU memory transfer, floating point performance, etc.

28.1 INTRODUCTION

Jacket is a software platform developed at AccelerEyes [1], which allows users to execute MATLAB M-code on CUDA-capable GPUs. MATLAB (MATrix LABoratory) by The MathWorks [2] has become the standard platform for technical computing and graphics in science, engineering, and finance due to its ease of use and powerful functional capabilities. The combination of a simple matrix language, interactive prompt, automatic memory management, and on-the-fly compilation make MATLAB well suited to rapid prototyping of algorithms and exploring data. MATLAB's one drawback is performance, and Jacket alleviates this by seamless offloading of computations to the GPU. Jacket provides users access to a set of libraries, functions, and tools that facilitate numerical computation on the GPU [3] including multi-GPU support built on MATLAB's Parallel Computing Toolbox and Distributed Computing Server [4].

Over the last decade, GPUs have proliferated among both consumer and developer computers. Despite the growing mass of success stories, GPU software development appears to still be considered the realm of niche applications. The consensus is that this is due to the difficulty in programming such devices. While the hardware continues to improve and the software ecosystem continues to grow, there is still a steep learning curve for the average programmer to reach success. Jacket is one of many software tools that attempts to bridge this gap by mapping high-level languages onto the underlying hardware. Other projects have attempted to bridge the gap. Cg, GLSL, HLSL, and Brook marked the beginning of stream programming, a precursor to general purpose GPU programming, where computation is mapped onto the graphics pipeline and consequently subject to various constraints. Following on the heels of these technologies, CUDA introduced a more generally programmable software architecture.

GPU Computing Gems. DOI: 10.1016/B978-0-12-385963-1.00028-9
© 2012 Elsevier Inc. All rights reserved.

Several companies set out to extend the capabilities of these early tools. One of the first such companies, PeakStream (now at Google) built a C/C++ runtime and library of functions providing a richer tool set to GPU development. RapidMind (now at Intel) set out to build a flexible middle-layer supporting various front-end languages and back-end hardware targets. All of these projects have sought the same thing: to bridge the gap between the hardware and developers. One of the newest platforms for GPU development, `Jacket`, allows programmers to use the high-level `MATLAB` M-language and abstracts away the low-level details of GPU programming. The result is increased productivity and performance.

28.2 JACKET

`Jacket` has been designed for programmers who have large data-parallel tasks but who are not low-level programmers accustomed to dealing with GPU-specific constructs. To allow them to avoid having to learn new language features, `Jacket` introduces only a handful of new functions to the `MATLAB` experience. Once data is marked as "GPU" data using these functions, `Jacket` provides native GPU implementations of a large set of the standard `MATLAB` functions to operate on that data.

28.2.1 Basics

`Jacket` achieves transparency by defining a new set of classes dubbed "g" objects, where each element of this set corresponds to a base class of the `MATLAB` standard interface: `single`, `uint16`, `ones`, etc. map to `gsingle`, `guint16`, `gones`, etc. — see [5]. These new classes function exactly as their CPU-based counterparts. In order to facilitate this functionality, each standard method currently present on the standard classes is made available on the "g" objects via GPU-enabled mex-code, the architecture of which is described below. The set of standard `MATLAB` language constructs from displaying the object as text to assigning values to the variables are supported, but the core of `Jacket` involves GPU equivalents to element-wise arithmetic, Fast Fourier Transform, matrix multiplication, singular value decomposition, etc. Large portions of already existing `MATLAB` programs can therefore be run on the GPU simply by changing the data types from their base class such as `single` to the GPU base class equivalent `gsingle`. Data can be copied to or from the GPU by way of a typecast between the base class and its "g" equivalent, or data can be generated directly on the GPU (either as random or constant data or as the output of some earlier GPU operation).

As a small example, suppose we want to add two random matrices `Agpu` and `Bgpu` by use of `Jacket` — the key to do this is the following:

```
>> Agpu = grand(3,3);  % grand() is the GPU equivalent of rand()
>> Bgpu = grand(3,3);
>> Rgpu = Agpu + Bgpu

Rgpu =
    0.5100    0.8279    0.2301
    0.4642    0.8148    0.8196
    0.5814    1.1443    1.0804
```

The call to the Jacket function grand(3,3) produces a random matrix of size 3×3 directly on the GPU, and Rgpu=Agpu+Bgpu produces the result. Since the the matrices Agpu and Bgpu are both GPU matrices so is the result matrix Rgpu, and all matrices reside in GPU memory. The above example takes advantage of the Jacket function grand to keep all computations on the GPU, but it is also possible to use the MATLAB function rand and push that result to the GPU and use the GPU for only the matrix addition:

```
>> Acpu = rand(3,3);
>> Bcpu = rand(3,3);
>> Rgpu = gsingle(Acpu) + gsingle(Bcpu)

Rgpu =
    1.2004    1.4460    0.7559
    1.5045    1.6116    0.8960
    1.0673    0.9352    0.9360
```

To pull back the result to the CPU we simply cast the result to a single:

```
>> Rcpu = single(Rgpu);
```

To make computationally efficient code, this move of data back and forth between CPU and GPU should be avoided when possible. More on this issue later.

28.2.2 Functions and Architecture

As with programming the GPU in C or C++, care must be taken by the programmer to ensure minimum memory transfer to the GPU and maximum data-parallelism — i.e., the maximum number of homogeneous operations are performed on the maximum number of data elements at a time. It is critical that MATLAB code be written in vectorized form in which large parts of data are operated upon by only a few operations in parallel. This vectorized paradigm of programming is exactly the style of programming necessary to meet the data-parallel requirement of GPU programming for maximum performance.

Unfortunately, a major feature of the M-language is M's standard convention of utilizing pass-by-value semantics: copies of objects are passed to functions instead of references. Thus, in many instances, multiple copies of objects are made throughout the execution of a MATLAB program. This convention in the M-language thus makes it impossible to effectively utilize standard MATLAB classes with completely exposed data-stores for use with the GPU. Many copies of objects on the GPU would be created, inefficiently utilizing bus bandwidth and GPU memory. Additionally, as copies of objects are made, there is no way within the MATLAB environment to intercept these events, making it impossible to achieve coherence between MATLAB and GPU memory state.

To bypass M's pass-by-value calling convention, the Jacket architecture uses object-oriented programming to handle references to data (reference counting to determine variable liveness). Such objects retain information about the location, size, and type of underlying data in memory, as well as any computations that have been performed on this data.

28.2.3 Comparing Jacket to CUDA C

To get an idea of the low-level details that Jacket hides, we construct an example of summing a vector of length n. Starting from a vector of random data, in MATLAB and Jacket this is rather straightforward:

```
% MATLAB
Acpu = rand(n,1,'single');
sum(Acpu)

% Jacket
Agpu = gsingle(Acpu);
sum(Agpu)
```

Programming just the sum function alone in CUDA C for maximum performance requires more care [6]. To achieve peak throughput, the final kernel looks something like the following and involves carefully exploiting parallelism at the level of thread warps:

```
__global__
static void kernel(unsigned n, float *d_dst, float *d_src)
{
  const unsigned tid = threadIdx.x;
  const unsigned grid = THREADS * gridDim.x;
  unsigned i = THREADS * blockIdx.x + tid;
  float sum = 0;
  while (i < n) { sum += d_src[i]; i += grid; }

  __shared__ float smem[THREADS];
  float *s = smem + tid;
  *s = sum;
  __syncthreads();

  if (tid < 64) { *s = sum = sum + s[64]; }
  __syncthreads();

  if (tid < 32) {
      volatile float *vs = s;
      *vs = sum = sum + vs[32]; *vs = sum = sum + vs[16];
      *vs = sum = sum + vs[ 8]; *vs = sum = sum + vs[ 4];
      *vs = sum = sum + vs[ 2]; *vs = sum = sum + vs[ 1];
  }

  if (tid == 0) d_dst[blockIdx.x] = *s;
}
```

To provide an idea of the relative performance of CUDA C vs. MATLAB vs. Jacket we have tested this small example on an Intel Core 2 Quad Q9400 (2.66 GHz) with an NVIDIA GeForce GTX480. The original MATLAB vector sum for 5 million elements runs in 4.80 ms while Jacket ran in 0.252 ms. Pushing everything into CUDA C (no MATLAB overhead) shaves this timing down to 0.236 ms.

This means a speed-up for Jacket of 19 and CUDA C achieved a speed-up of 20 relative to the MATLAB execution. The speed-up by this tuned device code compared with a naïve implementation can easily be 30 [6]. Compared to manually developing such routines on the GPU, examples such as this underscore both the productivity and performance gains achieved when moving to a high-level platform such as Jacket. All of the low-level tuning details are handled automatically for the user.

28.2.4 Lazy Evaluation

There are several major considerations Jacket must account for when compiling M-code into GPU instructions. First, M is an untyped language, meaning that types must be inferred, and since code can be loaded on the fly, much of this inference is delayed until execution. Second, GPU instructions must be grouped into a "kernel" that reads from GPU memory, performs computations, and finally writes to GPU memory. Therefore, instruction sequences cannot be prepared and issued at the same fine granularity as they can be for x86 processors. Jacket performs a high-order analysis of the M-code to determine appropriate segmentation of computation into independent kernels. This analysis considers the arithmetic intensity of the program, the amount of GPU memory it references, and thread block configurations for various phases of the computation. Next, data parallel algorithms might have a number of possible implementations, and the selection of the best one to use can vary substantially depending on the size of the input data, thus leading to a need for runtime optimizations such as on-the-fly compilation. Furthermore, there is a measurable overhead involved in preparing and emitting instructions for the GPU. In practice, long running computations typically contain loops and other repeated segments of code. Jacket maintains a cache of commonly used expressions and the code generated from those expressions for later reuse. Before beginning compilation, the Jacket runtime compares a candidate expression against the cache. Communication between CPU and GPU over the PCI bus introduces latency. To minimize communication, Jacket takes a lazy approach to evaluating computations and batches these sequences. In addition, seeing more of the computation sequence, Jacket has more opportunity for optimizations. The functions geval and gsync are available to provide the user with more control over this process if desired.

28.2.5 Graphics

Jacket includes a graphics toolbox that provides a simple method of displaying computational results on the GPU without bringing those results back to the host. Thus, the same dedicated hardware that computes results may be utilized to present those results, culminating in tightly integrated code-compute-visualize loop suitable for data mining, rapid prototyping, or production of real-time graphical applications. To balance the resources involved in computation, Jacket delays the dispatch of OpenGL instructions until a render pass is requested. At that point, the OpenGL pipeline is filled for the next display loop.

The Jacket Graphics Toolbox is exposed to the end user at varying levels and the user has a choice of interacting with any of these depending on their requirements. At the core of the API is a set of primitives that mirror MATLAB functionality. For example, gsurf mimics surf to draw a surface plot but adds surface texturing and lighting effects. The function gplot mimics plot to produce standard 2-D line plots, and gscatter3 mimics scatter3 to produce a scatterplot.

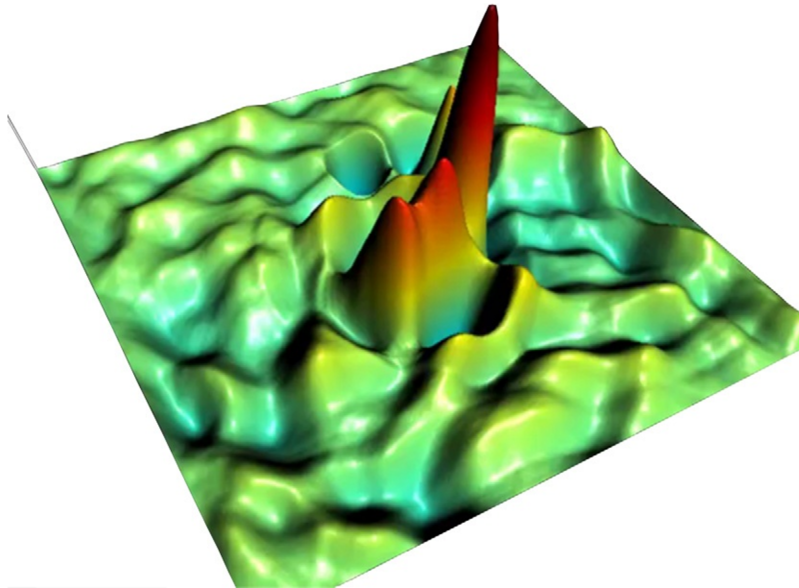

FIGURE 28.1

A droplet hitting a viscous surface utilizing the `gsurf` drawing primitive.

Figure 28.1 simulates the effect of a droplet hitting a viscous surface by use of `gsurf`. It begins with one droplet placed in the center. Each iteration diffuses the surface outward and computes the spring pull on the surface. At the end of the loop iteration, the surface is rendered (`gsurf`).

28.3 BENCHMARKING PROCEDURES

Benchmarking usually involves performing some specific `MATLAB` function where a certain parameter is swept (typically an array size) while timing the computation for both `MATLAB` (CPU) and `Jacket` (GPU). When performing benchmarks there are several issues to consider to reach reliable and reproducible results:

1. It is essential to choose a power setting for "high performance" to ensure the CPU is running at full speed, etc.
2. The `MATLAB` process should be given highest possible priority by the operating system.
3. Most modern CPUs allow some kind of multithreading, which may affect results significantly. By default, `MATLAB` uses the maximum number of threads available, and this is also what usually should be chosen when performing benchmarks.
4. A repetition loop should be put around the benchmark computations to achieve peak throughput and amortize any timer overhead.

5. Initialize pseudo-random number generators identically across runs that are to be compared, as the timing can depend on the actual values being processed.
6. Run the test code once to warm up both `MATLAB` (the CPU) and `Jacket` (the GPU). This allows instruction and data caching.
7. Use `geval` and `gsync` to ensure computations are completed. `gsync` should be used immediately before the timing functions `tic` and `toc` to ensure CPU-GPU synchronization, and `geval` is used to force evaluation prior to setting the stop timer.

Unless these issues are addressed, it can be virtually impossible to reproduce results.

28.4 EXPERIMENTAL RESULTS

The experimental results fall into three categories: (1) CPU/GPU memory transfer, (2) floating point performance, and (3) a selection of `MATLAB` functions. The results are described after an overview of the hardware platforms used for the benchmarking.

28.4.1 Hardware Platforms

The computer platforms used were two identical Colfax CXT2000i's based on an Asus Supercomputer P6T7 motherboard with an Intel Core i7–975 CPU (12 GB DDR3 memory). The computers were equipped with the NVIDIA GPUs shown in Table 28.1: one computer had one Quadro 4000 and three Quadro FX3800's, and the other had one Quadro FX3800 and a Tesla C1060, Tesla C2050 or GeForce GTX580. Both computers used the Microsoft Windows 7 x64 Enterprise operating system (ver. 6.1 build 7400), NVIDIA driver 263.06, MATLAB 7.11.0.584 (R2010b), Jacket 1.6.0 (build 9686), and CUDA Toolkit 3.1. Both computers have a change in the Windows Registry to set the "TDR Delay" to

Table 28.1 Estimated Theoretical Performance of the Used NVIDIA GPUs. Data Based on GPUReview.com and NVIDIA.com. "C. Cap." Denotes "Compute Capability"

GPU type	Single pr.[†] [GFlops]	Double pr. [GFlops]	Cores [–]	RAM [MB]	C. Cap.
Core i7–975	110.8	55.4	4	—	—
FX3800	462.3	57.8	192	1024	1.3
GTX580	1572.9	393.2	512	1534	2.0
Q4000	486.4	243.2	256	2048	2.0
C1060	622.1	77.8	240	4096	1.3
C2050	1030.4	515.2	448	3072	2.0

[†] *As the floating point performance is measured from matrix multiplications only two floating point operations per cycle have been used for calculation of theoretical single precision performance.*

30 seconds.[1] For all results, MATLAB was running with the maximum number of available host threads (8), and the MATLAB process was running with High Priority set in the Windows Task Manager. For the Tesla C2050, ECC was enabled unless otherwise noted. All results shown in the following were fully reproducible.

28.4.2 Memory Access

Since MATLAB is hosted on the CPU and Jacket works on the GPU, minimizing the transfer of data the between the CPU's and the GPU's memories is critical for the performance of Jacket. On the test platforms the overhead in both transfer directions was measured to about 85 μs with a linear relation between transfer time and array size. Once data has been moved to GPU memory, Jacket generally has much faster access to GPU memory than MATLAB has access to CPU memory. For all tested GPUs the transfer rate from host (computer) to device (GPU) was around 3.8 GB/s, and from device to host the transfer rate was around 2.6 GB/s when the array size was larger than approximately 5 MB. Due to this, data should be transferred in large chunks whenever possible in order to minimize the total transfer time.

28.4.3 Floating Point Operations

The purpose of the floating point benchmark is to see what Jacket can achieve in terms of floating point arithmetic operations per second with respect to the size of the matrices involved. Jacket includes a highly tuned version of generalized matrix multiply (not NVIDIA CUBLAS's GEMM, at present) that provides a good candidate for measuring performance. The following high-arithmetic intensity operation is performed at the core of matrix multiplication:

$$\mathbf{R} := \alpha\mathbf{A}\mathbf{B} + \delta\mathbf{R}, \qquad \alpha,\delta \in \mathbb{R}; \quad \mathbf{A},\mathbf{B},\mathbf{R} \in \mathbb{R}^{N\times N} \tag{1}$$

where the number of floating point operations is $2N^3 + N^2$; see [7]. The GPUs listed in Table 28.1 perform as shown in Figures 28.2 and 28.3. The performance of an Intel Core i7–975 CPU is shown for comparison.

As seen from Figure 28.2 for single precision performance, there is the expected clear progression toward better performance for the more powerful GPUs. The GeForce GTX580 clearly outperforms everything else with a peak performance of 849 GFlops. The Tesla C2050 follows as the next with a peak performance of 550 GFlops, and more than 350 GFlops for matrix sizes above approximately 1200×1200. In single precision, all measured GPUs deliver 53–58% of theoretical peak performance. Disabling ECC on the Tesla C2050 increased the peak performance by 0.4 %.

As seen for the double precision results in Figure 28.3, the C2050 has the highest peak performance of 249 GFlops where the GTX580 delivers 193 GFlops. For double precision, the GeForce GTX580, Tesla C2050, and Quadro 4000 deliver 48–52% of theoretical peak performance where the older Quadro FX3800 and Tesla C1060 deliver 94–96% of theoretical peak performance. Disabling ECC on the Tesla C2050 increased the peak performance by 4.6%.

---

[1] See http://www.microsoft.com/whdc/device/display/wddm_timeout.mspx for more information on the WDDM Timeout Detection and Recovery mechanism.

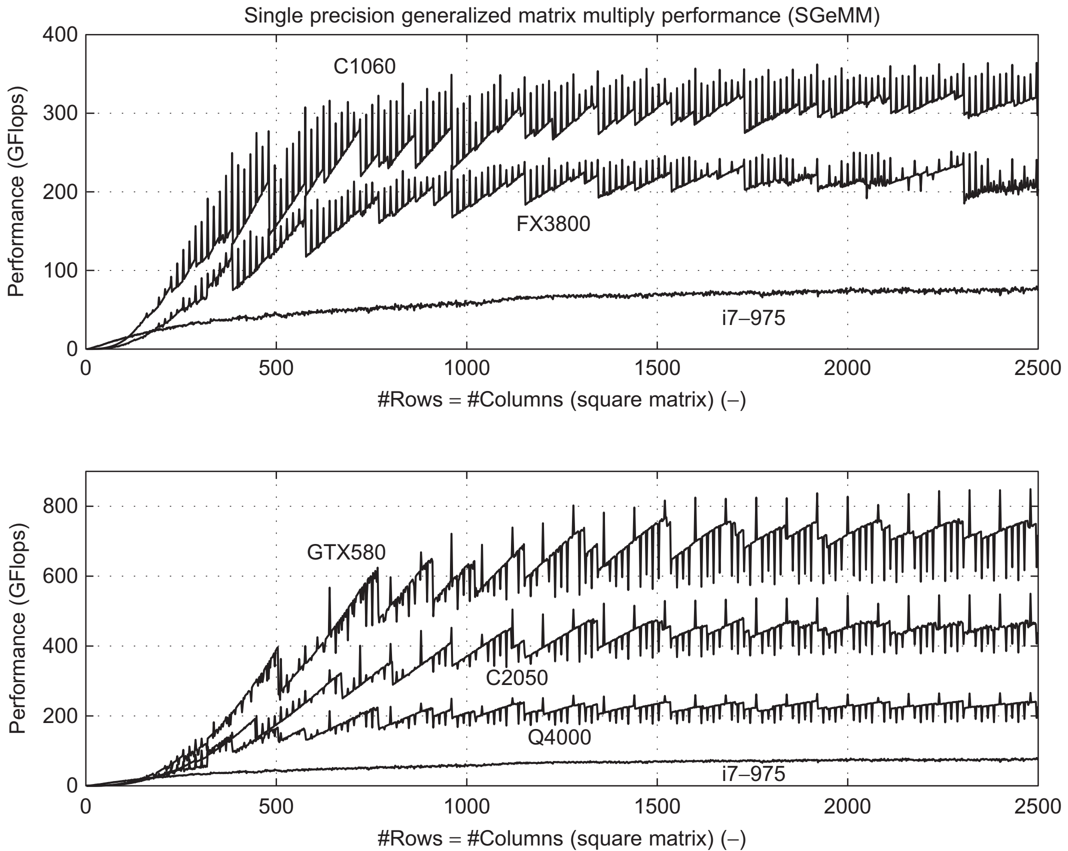

FIGURE 28.2

Measured single precision floating point performance versus square matrix size for different GPUs and an Intel Core i7–975 CPU.

28.4.4 Functions

The performance of `Jacket` has also been measured for a number of different functions for vector/matrix input and single/double precision; see Table 28.2. The reference used in all the benchmarks is again an Intel Core i7–975 CPU. When observing the results for a number of different functions some general observations can be made. Some functions are extremely fast on the GPU (e.g., `power`, `interp2`, and trigonometric functions) whereas others we refer to as “glue” functions (such as `subsasgn`). These are functions that are not ideally suited to GPU execution, but for which it can be advantageous to run them on the GPU anyway in conjunction with other GPU operations to avoid transferring data back and forth between CPU memory and GPU memory.

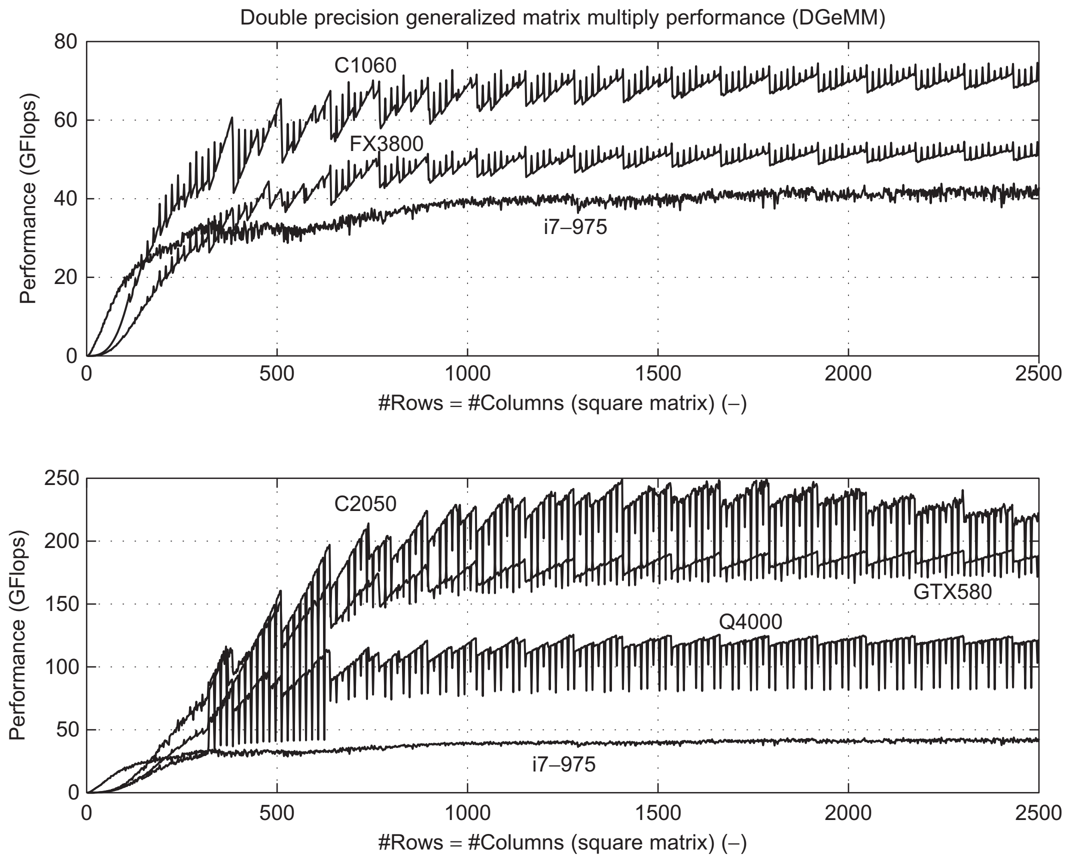

FIGURE 28.3

Measured double precision floating point performance versus square matrix size for different GPUs and an Intel Core i7–975 CPU.

Benchmarks are published regularly [8], and functions are generally improved when necessary. Other functions that work well on the GPU are `grand` and `grandn`, which generate random numbers directly on the GPU. These are important functions as they are used in virtually all scientific areas.

28.5 FUTURE DIRECTIONS

`Jacket` efficiently and transparently brings GPU computing to the popular `MATLAB` tool for computations, programming, and visualizations, delivering performance near to that of native `CUDA C` programs while maintaining the ease of programming provided by `MATLAB`'s M-language.

Table 28.2 Measured `Jacket` Performance as GPU Speed-Up Relative to a CPU. Matrix Size was 2048×2048, and Vector Size was $2^{22} \times 1$

Measured Speed-up for Tesla C2050 vs. Core i7–975				
Function	**Matrix**		**Vector**	
	Single	**Double**	**Single**	**Double**
all	4.64	5.04	7.26	6.68
any†	4.62	5.15	7.37	6.85
asinh	44.34	12.72	44.12	12.60
atan2	296.95	93.49	297.38	93.71
atan	35.81	8.05	35.83	7.91
chol	29.30	1.73	—	—
conv2	2.49	—	—	—
cos	25.10	11.30	24.88	11.38
det	2.06	1.76	—	—
exp	41.95	15.58	41.54	15.68
fft	17.87	8.90	37.56	24.84
find	17.73	16.92	17.77	16.94
ifft	10.79	5.47	29.37	16.22
interp1	—	—	169.11	138.44
interp2	426.19	—	—	—
inv	—	1.56	—	—
log	31.09	13.37	31.15	13.44
lu	2.44	2.01	0.38	0.40
max	1.43	2.58	1.80	2.59
min	1.43	2.58	1.80	2.53
minus	17.53	9.31	17.55	9.62
mldivide	3.12	2.16	—	—
norm	0.64	0.92	2.44	25.60
plus	17.43	9.25	17.45	9.35
power	40.74	13.48	40.69	13.57
rand	32.21	32.75	32.16	32.72
randn	24.20	25.15	24.22	25.14
rdivide	10.50	5.96	10.61	5.98
sort	5.96	4.73	7.95	2.26
subsasgn	0.08	0.08	0.02	0.01
sum	1.42	2.60	1.85	2.57
times	21.36	9.40	21.35	9.37
trapz	2.34	20.13	0.74	0.84

† *The speed-up of the* `any` *function depends significantly on the density of the vector/matrix. The result shown is for the most typical case of a sparse vector/matrix.*

The near future of Jacket is focused on adding support for sparse arrays and different optimization techniques. Furthermore, the Jacket core is being segregated and repackaged as libJacket, which facilitates MATLAB–independent Jacket computing directly from the users' own C/C++ programs.

References

[1] AccelerEyes, Addr.: 800 W Peachtree St NW, Atlanta, GA 30308, USA. URL: http://www.accelereyes.com.
[2] The MathWorks, Inc., MATLAB. Addr.: The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760-2098, USA. URL: http://www.mathworks.com.
[3] A. Webb, MATLABs racing jacket, Autom. Trader 16 (1) (2010) 54–61.
[4] G. Sharma, J. Martin, MATLAB: a language for parallel computing, Springer Int. J. Parallel Program. 37 (1) (2008) 3–36.
[5] AccelerEyes, Jacket v1.5: Getting Started Guide: URL: http://www.accelereyes.com/services/documentation.
[6] M. Harris, Optimizing CUDA, SuperComputing 2007 Tutorial, Reno, NV, 2007.
[7] V. Volkov, J.W. Demmel, Benchmarking GPUs to Tune Dense Linear Algebra, in: SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, Austin, Texas, 2008, pp. 1–11.
[8] Torben's Corner, Available from the Wiki of AccelerEyes at: URL: http://www.accelereyes.com/wiki/index.php?title=Torben's_Corner.