NMath Premium: GPU-Accelerated Math Libraries for .NET

CenterSpace™
Software

Introduction

CenterSpace Software’s **NMath**™ numerical library provides object-oriented components for mathematical, engineering, scientific, and financial applications on the .NET platform. The **Premium Edition** of **NMath** leverages the power of NVIDIA’s CUDA™ architecture for accelerated performance.

CUDA is a parallel computing platform and programming model developed by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU). GPU computing is a standard feature in all NVIDIA's 8-Series and later GPUs. The entire NVIDIA Tesla line supports CUDA. For a full list of supported products, see:

https://developer.nvidia.com/cuda-gpus

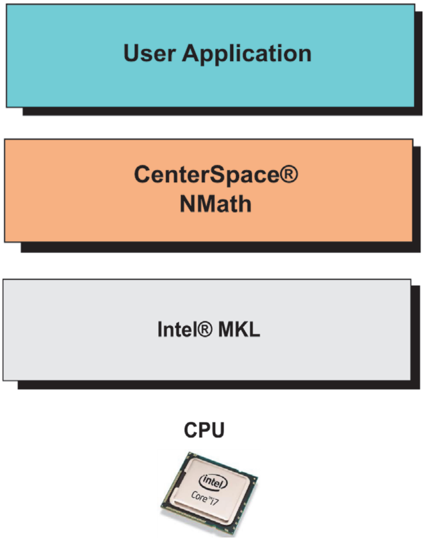
NMath and NMath Premium

For most computations, **NMath** uses the Intel® Math Kernel Library (MKL), which contains highly-optimized, extensively-threaded versions of the C and FORTRAN public domain computing packages known as the BLAS (Basic Linear Algebra Subroutines) and LAPACK (Linear Algebra PACKage).

This gives **NMath** classes performance levels comparable to C, and often results in performance an order of magnitude faster than non-platform-optimized implementations.[1] Deployment to both x86 and x64 environments is supported, with the appropriate native code loaded automatically at runtime. By using MKL, **NMath** gets the most out of today’s multicore and multiprocessor architectures, and future-proofs applications for tomorrow’s 32-, 64-, and 128-core designs.

[1] For more information on NMath performance, see:
http://www.centerspace.net/doc/NMath/whitepapers/NMath.Benchmarks.pdf

Figure 1 – NMath stack

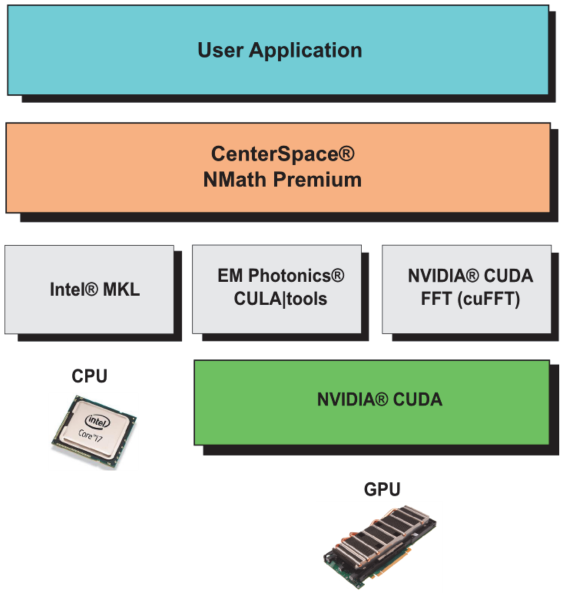


For additional acceleration, **NMath Premium** leverages the power of the GPU. **NMath Premium** automatically detects the presence of a CUDA-enabled GPU at runtime (64-bit only) and seamlessly redirects appropriate computations to it (Figure 2). The library can be configured to specify which problems should be solved by the GPU, and which by the CPU. If a GPU is not present at runtime, or on 32-bit systems, the computation automatically falls back to the CPU without error.

NMath Premium's Adaptive Bridge™ technology provides:

- Support for multiple GPUs
- Per-thread control for binding threads to GPUs
- Automatic performance tuning of individual CPU–GPU routing to insure optimal hardware usage

Figure 2 – NMath Premium stack



Automatic routing based
on problem size and available
hardware (customizable)

NMath Premium addresses the biggest barriers to adoption of GPUs:

- No GPU programming experience is required.
- No changes are required to existing **NMath** code.

Existing **NMath** developers can simply upgrade to **NMath Premium** and immediately begin to offer their users higher performance from current graphics cards, or from additional GPUs, without writing any new software.

New **NMath** developers can develop their applications without worrying about the details of multicore optimization and GPU acceleration.

For example, the following **NMath** code computes a forward, 1D, discrete fourier transform (DFT), in place, overwriting the input data:

```
var fft = new DoubleComplexForward1DFFT( 200000 );
var signal = new DoubleComplexVector( 200000,
   new RandGenUniform( -1, 1, 888 ) );
fft.FFTInPlace( signal );
```

With the **NMath** assemblies, this code uses MKL's FFT implementation, highly parallelized for multicore CPUs. With the **NMath Premium** assemblies, and in the presence of a CUDA-enabled GPU, this same code routes the computation to NVIDIA's CUDA Fast Fourier Transform (cuFFT) library.

Supported Features

Only select **NMath** classes are able to route their computations to the graphics processor. The directly supported features for GPU acceleration of linear algebra (dense systems) include:

- Singular value decomposition (SVD)
- QR decomposition
- Eigenvalue routines
- Solve Ax = B

GPU acceleration for signal processing includes:

- 1D Fast Fourier Transforms (Complex data input)[2]
- 2D Fast Fourier Transforms (Complex data input)

Of course, many higher-level **NMath** and **NMath Stats** classes make use of these functions internally, and so also benefit from GPU acceleration indirectly.

NMath

- Least squares, including weighted least squares
- Filtering, such as moving window filters and Savitsky-Golay
- Nonlinear programming (NLP)
- Ordinary differential equations (ODE)

NMath Stats

- Two-Way ANOVA, with or without repeated measures
- Factor Analysis
- Linear regression and logistic regression
- Principal component analysis (PCA)
- Partial least squares (PLS)
- Nonnegative matrix factorization (NMF)

[2] Real signals can currently be handled by filling the imaginary parts with zeros.

GPU Tuning

Math Premium is pre-tuned to work reasonably well for common hardware. Larger problems are routed to the GPU, where there's typically a speed advantage, while smaller problems are retained on the CPU. In these cases, it is not worth incurring the data transfer time to the GPU.

However, to get the most out of your available hardware, or your users' hardware, you can explicitly tune a bridge for optimal performance. The tuning process performs a series of sample timings on your hardware, models the performance, and infers an optimal routing from the model.

```
var device = BridgeManager.Instance.GetComputeDevice( 0 );
var bridge = BridgeManager.Instance.NewDefaultBridge( device );

int maxProblemSize = 1200;
bridge.Tune( BridgeFunctions.dgemm, device, maxProblemSize );
```

Once a bridge is tuned it can be persisted, redistributed, and used again.

GPU logging can be enabled to record a history of the routing process.

```
string path = @"C:\tmp\MyBridge.log";
bool append = false;
BridgeManager.Instance.EnableLogging( path, append );
```

For example, the following log shows two general matrix multiplications. The first, multiplying two `100 x 100` matrices, is below threshold and issued to the CPU to be performed by MKL. The second, multiplying two `1000 x 1000` matrices, is over threshold and issued to the GPU to be performed by CUDA libraries.

```
2014-07-10 08:33:04.795 AM 13 0 dgemm CPU Below threshold ;
  size n x m 100x100 = 10000 < 490000 (crossover 700)
2014-07-10 08:33:04.959 AM 13 0 dgemm GPU Above threshold ;
  size n x m 1000x1000 = 1000000 >= 490000 (crossover 700)
```

For more information on GPU tuning, see the chapter on **NMath Premium** in the *NMath User's Guide*.

Factors Affecting GPU Performance

NMath Premium GPU acceleration typically provides up to 5x speed-up for supported operations. With large data sets running on high-performance GPUs, the speed-up can exceed 10x.[3] Furthermore, off-loading computation to the GPU frees up the CPU for additional processing tasks, a further performance gain.

The biggest factors affecting GPU performance are:

- **Hardware**

  NVIDIA manufactures a wide range of CUDA-enabled GPUs, with varying price points and performance characteristics.

- **Data Transfer Time**

  Because computation is done in a different memory space from the CPU's memory, the data must be transferred to the GPU's local memory. Once the algorithm is complete, the results must be transferred back to the host's memory. As a result, if the computational complexity of the work done on the GPU isn't high enough, the benefits of massively parallel computation on the GPU can be swamped by the data transfer times. NVIDIA has answered this issue with hardware innovations including increased bus bandwidth, pipe-lined data transfers, and dual DMA engines to reduce data transfer overhead times, but it's still an important consideration when choosing what kind of algorithms to run on a GPU.

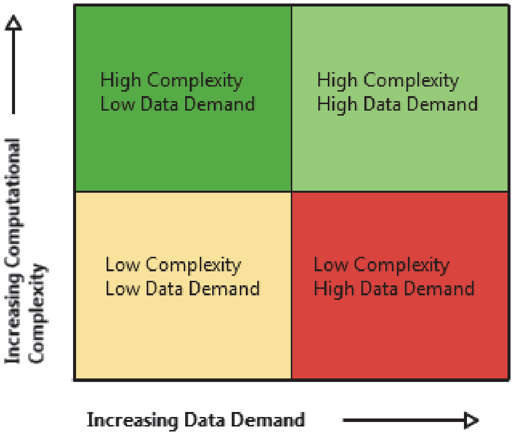
- **Computational Precision**

  Computational GPU hardware design originated from video processors, and in the early years the technology was almost exclusively driven by the gaming industry. For that reason, GPU processors have been optimized for many years for single precision workloads. Until recently, GPU single precision computation would far out perform double precision computation. The recent K20/K20x from NVIDIA, however, is optimized for double precision workloads.

As a general guide, the best algorithms to target for GPU computation are those with high computational complexity and relatively low data demand. Within that group, the very best performance will be realized by algorithms which best leverage the fine-grain parallelism offered by GPUs.

[3] Note that this is in addition to the greater than 10x speedup already provided by MKL, as compared to straight C# implementations.

Figure 3 – Data Transfer Cost vs Computational Complexity



Because data transfer is an important component of realized GPU performance, in this document *all reported GFLOP performance numbers take into account the data transfer time in both directions*. This is done to give an accurate picture of attainable performance.

Linear Algebra Performance

Four linear algebra algorithms were tested: SVD, QR, and Eigenvalue Decomposition, and Solve, for solving a linear system of equations, Ax = b.

Benchmarks were run on three different NVIDIA GPUs (Table 1), as well as a quadcore 2.0 Ghz Intel i7 CPU.

Table 1 – NVIDIA GPU Tested for Linear Algebra

GPU	Peak GFLOP (single / double)	Summary
Tesla K20	3510 / 1170	Optimized for applications requiring double precision performance such as computational physics, biochemistry simulations, and computational finance.
Tesla K10	2288/ 95	Optimized for applications requiring single precision performance such as seismic and video or image processing. The K10 features two GPUs on one board; if both GPU cores are maximally utilized these GFLOP numbers would double.
Tesla 2090	1331/ 655	A single core GPU with a more balanced single and double precision performance.

Single Precision Linear Algebra

Figure 4 through Figure 7 illustrate the GFLOP single precision performance. Each algorithm was run with four random dense matrices of size `500 x 500`, `3000 x 3000`, `7000 x 7000` and `10000 x 10000`. Thus, the largest matrix tested had 100 million elements and occupied about 380 MBytes.

Figure 4 – Single Precision Solve Ax = b

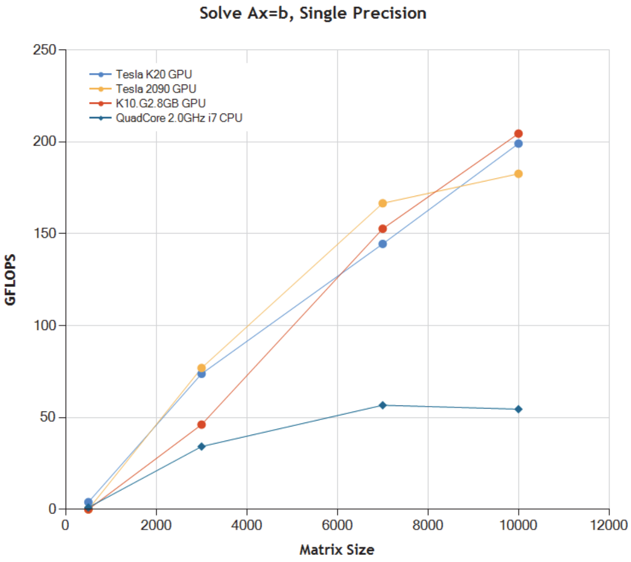


Figure 5 – Single Precision Eigenvalue Decomposition

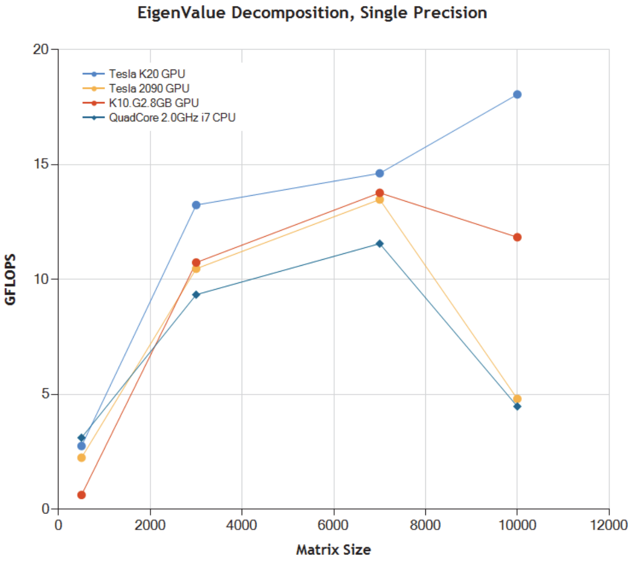


Figure 6 – Single Precision QR Decomposition

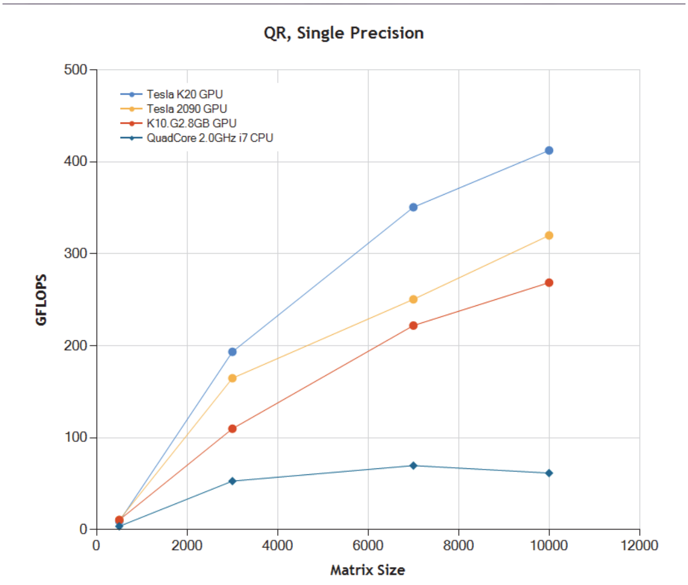
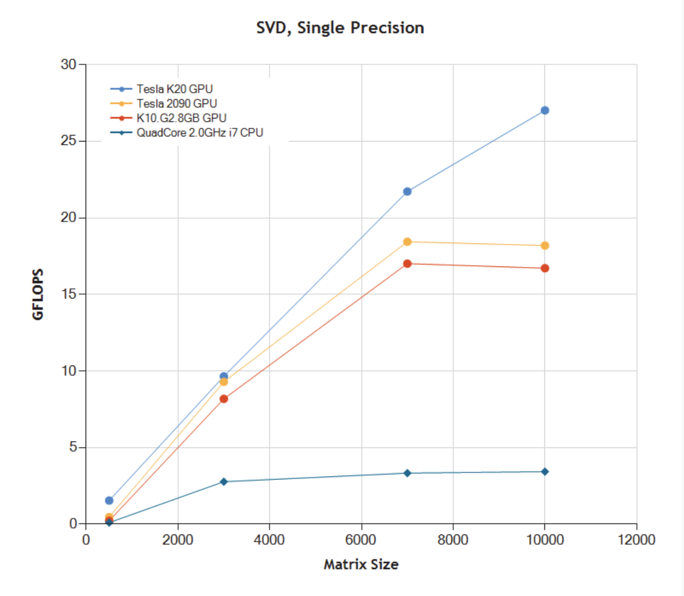


Figure 7 – Single Precision SVD



The GPU-enabled routines outperform the CPU-bound algorithms by several multiples over a large range of problem sizes.

Table 2 summarizes the GPU performance increase as a multiple of CPU performance, averaged over the three largest matrix sizes. Clearly the larger the problem, the more advantageous using the GPU becomes. The `10000 x 10000` QR decomposition is more than 10 times faster on the K20 GPU than on the quadcore i7 CPU, exceeding 400 GFLOPS.

Table 2 – GPU Performance Increase as a Multiple of CPU Performance

Algorithm	Speed-up over CPU		
	K20	**K10**	**Tesla 2090**
Solve	2.9x	2.8x	2.9x
EigenValue Decomp	1.8x	1.4x	1.1x
QR	5.2x	3.2x	4.0x
SVD	6.6x	4.7x	5.0x

With small data sets, when performance is largely bound by the data transfer time, little difference is seen between the tested GPUs, because they all have roughly the same PCIe bandwidth.

Double Precision Linear Algebra

Figure 8 through Figure 11 illustrate the GFLOP double precision performance. Again, each algorithm was run with four random dense matrices of size `500 x 500`, `3000 x 3000`, `7000 x 7000` and `10000 x 10000`.

Double precision GPU performance is typically half the performance of single precision, and **NMath Premium** performs near this optimum for many of the GPU/algorithm pairs. As expected, on the NVIDIA K10, the double precision performance falls substantially from its single precision performance. However, the Tesla 2090 and K20 performance exceeds the CPU-bound performance by up to 7x.

Figure 8 – Double Precision Solve Ax = b

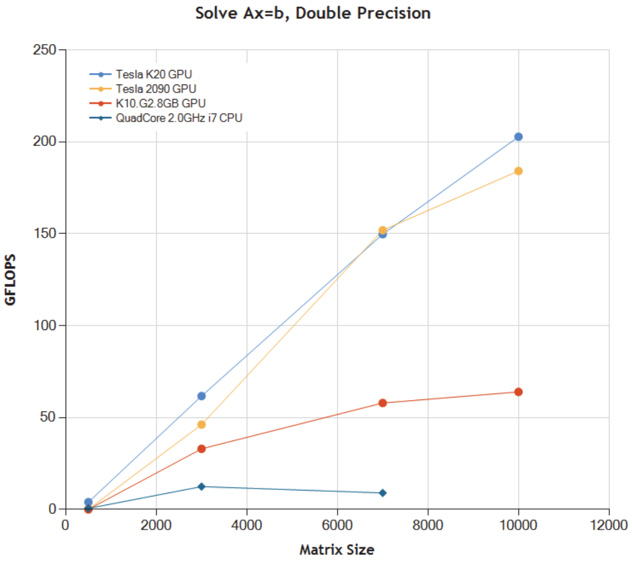


Figure 9 – Double Precision Eigenvalue Decomposition

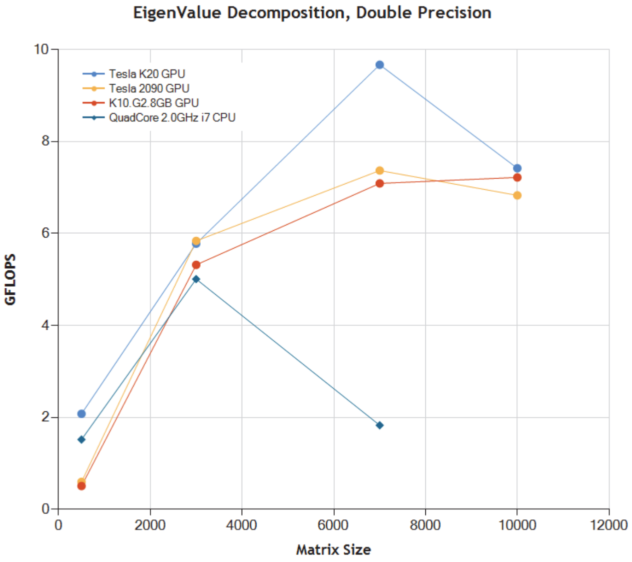


Figure 10 – Double Precision QR Decomposition

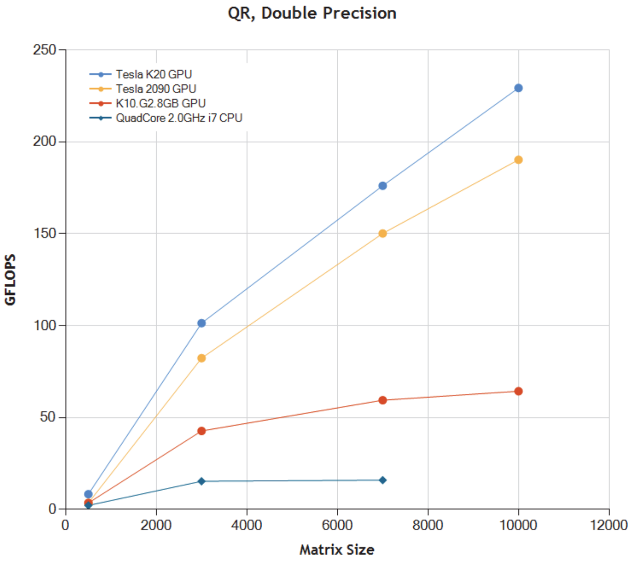
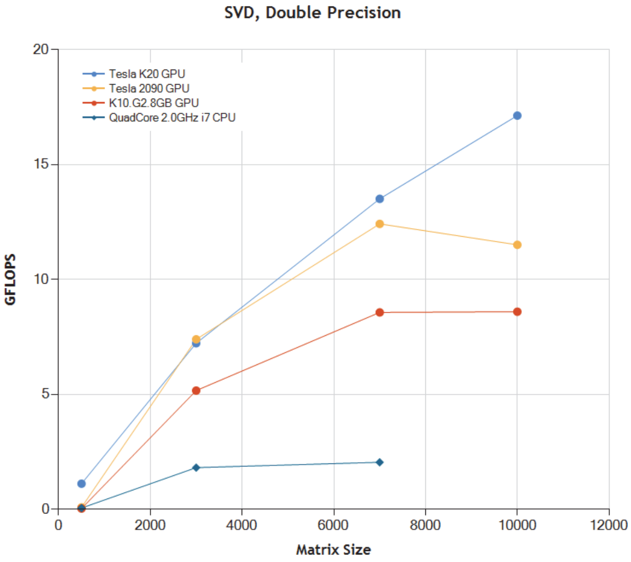


Figure 11 – Double Precision SVD



Signal Processing Performance

FFT benchmarks were run on four different NVIDIA GPUs (Table 3), and a quadcore 2.0 Ghz Intel i7 CPU for comparison. The GPUs represent the current range of performance available from NVIDIA—from the widely-installed, consumer-grade GeForce GTX 525 to NVIDIA's fasted double precision GPU, the Tesla K20.

Table 3 – NVIDIA GPU Tested for FFT

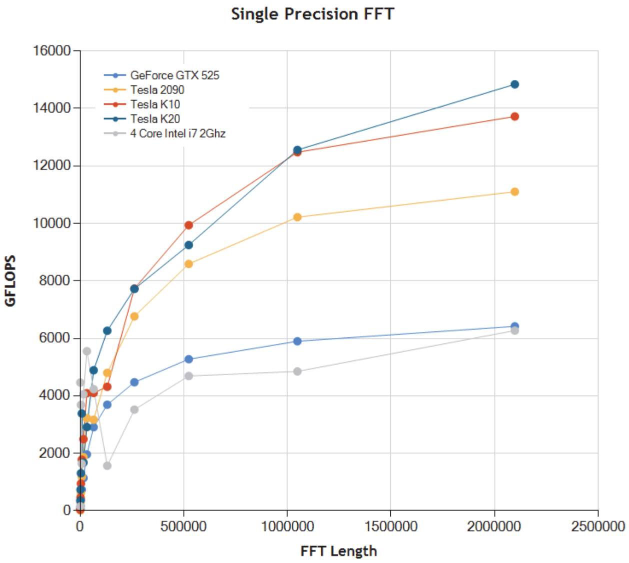
GPU	Peak GFLOP (single / double)	Summary
Tesla K20	3510 / 1170	Optimized for applications requiring double precision performance such as computational physics, biochemistry simulations, and computational finance.
Tesla K10	2288/ 95	Optimized for applications requiring single precision performance such as seismic and video or image processing. The K10 features two GPUs on one board; if both GPU cores are maximally utilized these GFLOP numbers would double.
Tesla 2090	1331/ 655	A single core GPU with a more balanced single and double precision performance.
GeForce 525	230 / -	A single core consumer GPU found in many gaming computers.

Modern FFT implementations are hybridized algorithms which switch between algorithmic approaches and processing kernels depending on the available hardware, FFT type, and FFT length. For instance, an FFT library may use the classical Cooly-Tukey algorithm for a short, power-of-two FFT, but switch to Bluestein's algorithm for odd-length FFTs. Furthermore, depending on the factors of the FFT length, different combinations of processing kernels may be used. Thus, there is no single 'FFT algorithm', and no easy expression for FLOPS completed per FFT computed.

Therefore, when analyzing the performance of FFT libraries, performance is often reported relative to the Cooly-Tukey implementation, with the FLOPs estimated at `5 * N * log( N )`. This relative performance is used here. For example, a reported performance of 10 GFLOPs means you'd need a 10 GFLOP-capable machine running an implementation of the Cooly-Tukey algorithm to match the performance.

Figure 12 through Figure 15 illustrate the performance of various power-of-two length, complex-to-complex forward 1D and 2D FFTs.[4] Again, copy time overhead is included in all reported performance numbers to give an accurate picture of attainable performance.

Figure 12 – Single Precision 1D FFT



[4] All NMath products also compute non-power-of-two length FFTs but their performance is not benchmarked here.

Figure 13 – Double Precision 1D FFT

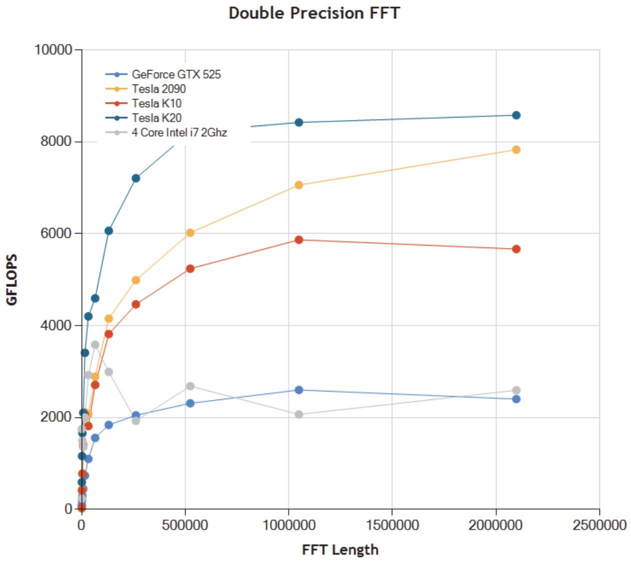


Figure 14 – Single Precision 2D FFT

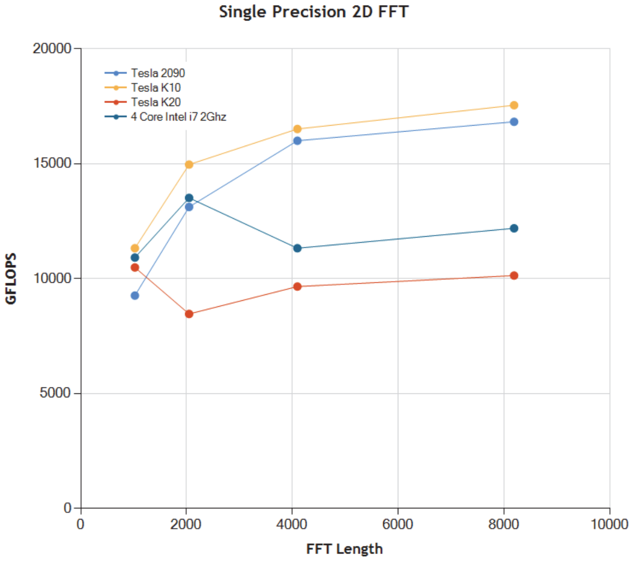
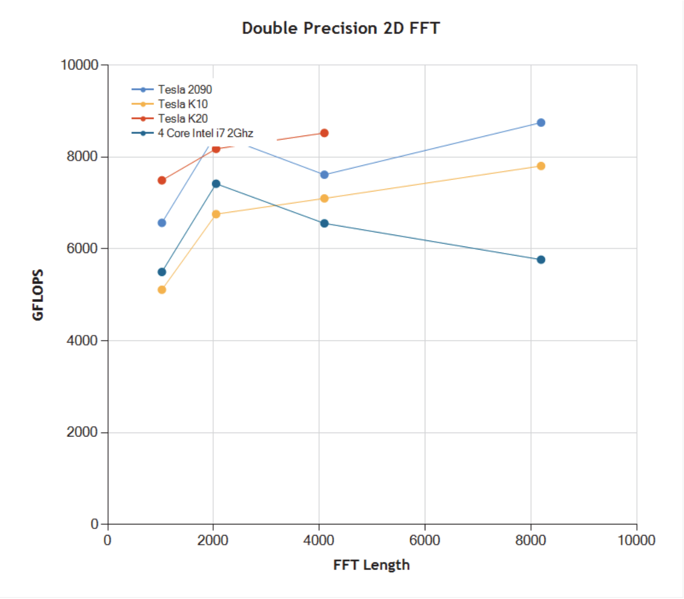


Figure 15 – Double Precision 2D FFT



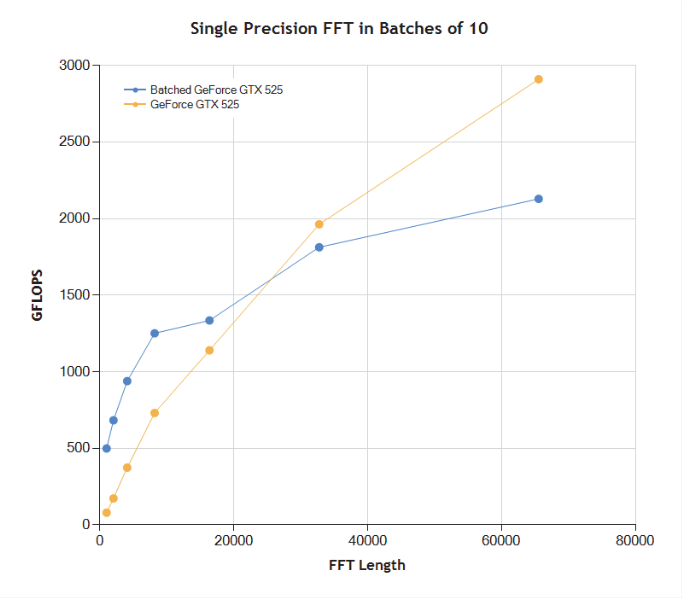
The performance of the CPU-bound 1D FFT outperformed all of the GPUs for relatively short FFT lengths. This is expected because the superior performance of the GPUs cannot be enjoyed due to the data transfer overhead. Once the computational complexity of the 1D FFT is high enough, however, the data transfer overhead is outweighed by the efficient parallel nature of the GPUs, and they start to overtake the CPU-bound 1D FFTs. This crossover point occurs when the FFT reaches a length near 65,536. The exception is the consumer level GeForce GTX 525, where the GPU and CPU FFT performance roughly track each other.

The 2D FFT case is different than 1D FFT, because of the higher computational demand. In single precision workload, we see the inferiority of the NVIDIA K20, which is designed primarily as a double precision computation engine. Here, the CPU-bound outperforms the K20 for all image sizes. However, the K10 and 2090 are extremely fast (including the data transfer time) and outperform the CPU-bound 2D FFT by approximately 60-70%.

In the double precision 2D FFT case, the K20 outperforms all other processors in nearly all cases measured. The tested K20 was memory limited in the `8192 x 8192` test case and couldn't complete the computation.

To amortize the cost of data transfer to and from the GPU, **NMath Premium** can also run FFTs in batches of signal arrays.

Figure 16 – Batch FFT



For the smaller FFT sizes, the batch processing nearly doubles the performance of the FFT on the GPU. As the length of the FFT increases, the advantage of batch processing decreases because the full array signals can no longer be loaded into the GPU.

In summary, as the complexity of the FFT increases, either due to an increase in length or problem dimension, the GPU-leveraged FFT performance overtakes the CPU-bound version. The advantage of the GPU 1D FFT grows substantially as the FFT length grows beyond ~100,000 samples. Batch processing of signals arranged in rows in a matrix can be used to mitigate the data transfer overhead to the GPU.

Note that there are times where it may be advantageous to offload the processing of FFTs onto the GPU *even when CPU-bound performance is greater*, because this frees CPU cycles for other activities. Using the GPU tuning API described above, the **NMath Premium** developer can easily push all FFT processing to the GPU, completely offloading this work from the CPU.

Conclusions

NMath Premium dynamically maximizes the use of available hardware, whether multiple CPU cores or GPU hardware. By leveraging the NVIDIA CUDA architecture, **NMath Premium** typically provides up to 5x speed-up for supported operations. With large data sets running on high-performance GPUs, the speed-up can exceed 10x. No GPU programming experience is required, and no changes are required to existing **NMath** code.

NMath Premium: GPU-Accelerated Math Libraries for .NET

The correct bibliographic reference for this document is:
NMath Premium: GPU-Accelerated Math Libraries for .NET, CenterSpace Software, Corvallis, OR.

Printed in the United States.
Printing Date: March, 2016

CenterSpace Software

Address:	622 NW 32nd St., Corvallis, OR 97330 USA
Phone:	(541) 896-1301
Web:	http://www.centerspace.net
Technical Support:	support@centerspace.net