GPU Programming In Java Using OpenCL

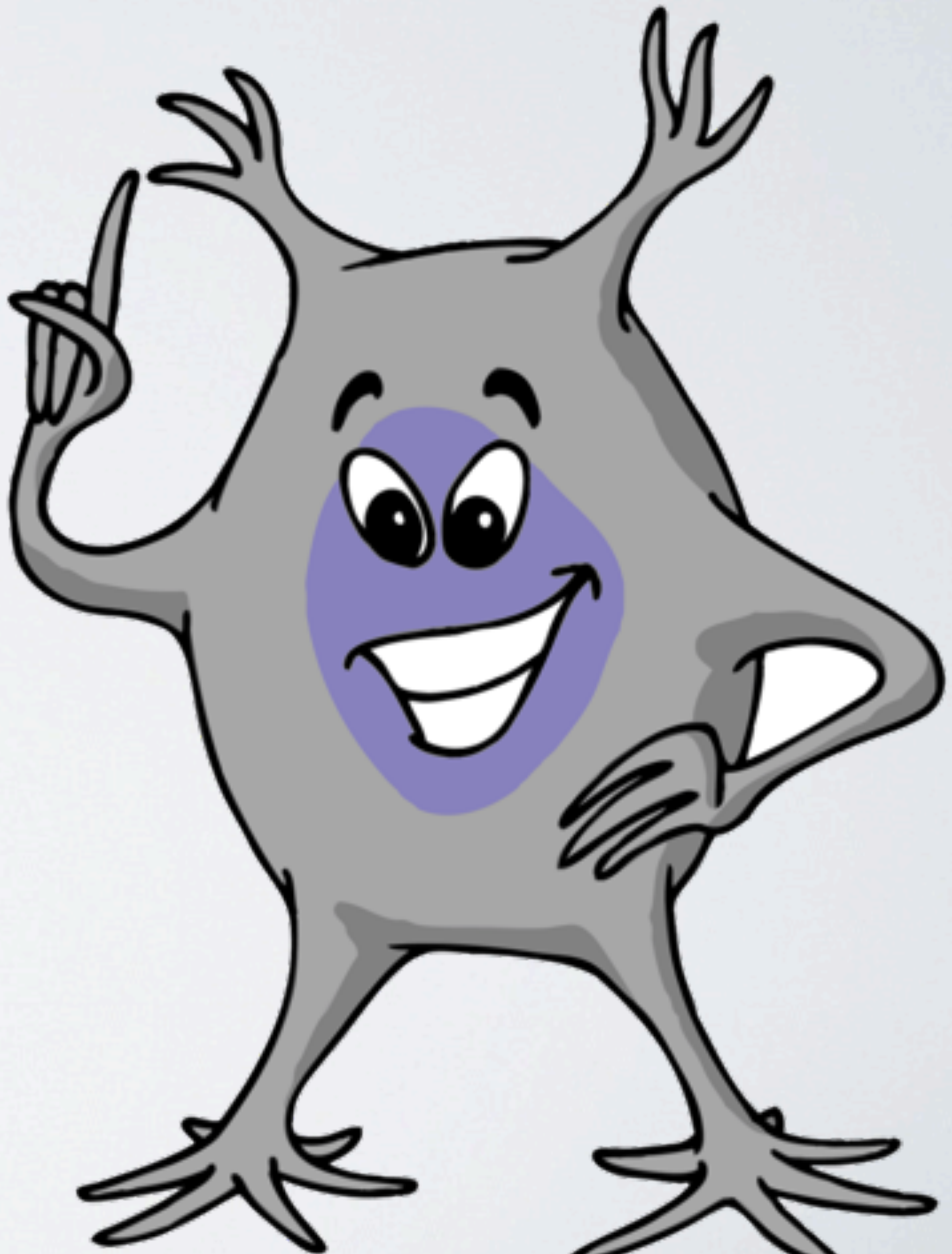
Jeff Heaton

- **EMail:** jheaton@heatonresearch.com
- **Twitter:** @jeffheaton
- **Blog:** http://www.jeffheaton.com
- **GitHub:** https://github.com/jeffheaton

ENCOG

Lead developer for Encog. Encog is an advanced machine learning framework that supports a variety of advanced algorithms, as well as support classes to normalize and process data.

http://www.heatonresearch.com/encog

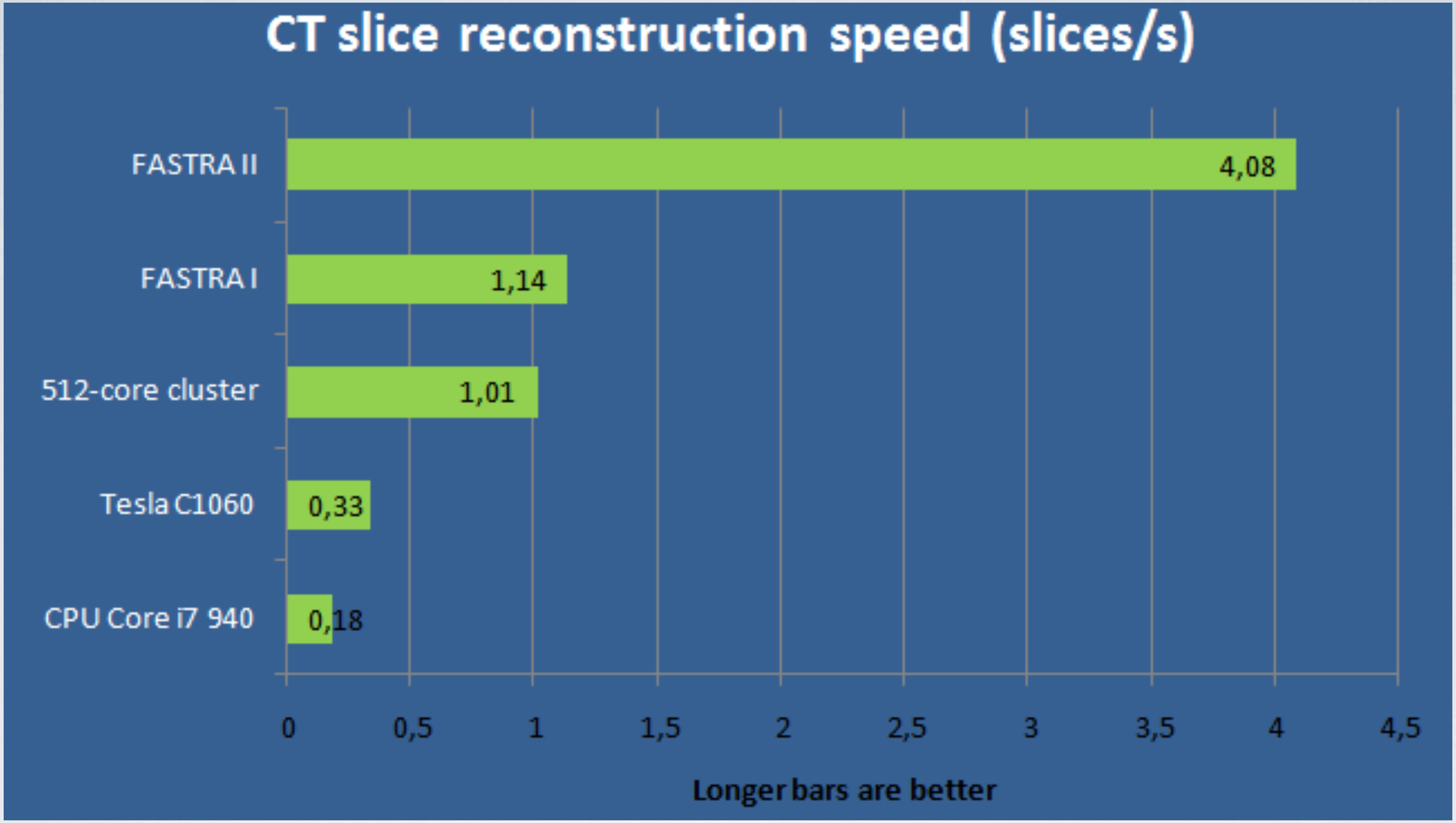


WHAT IS GPGPU?

- General-Purpose Computing on Graphics Processing Units
- Use the GPU to greatly speed up certain parts of computer programs
- Typically used for mathematically intense applications
- A single mid to high-end GPU can accelerate a mathematically intense process
- Multiple GPU's can be used to replace a grid of computers

Desktop Supercomputers

- The Fastra II Desktop Supercomputer
- Built by the University of Antwerp
- 7 GPUs
- 2,850 Watts of power
- Built with "gamer" hardware



FASTRA II PERFORMANCE

Compared to CPU Clusters

NVIDIA HARDWARE

- **GeForce** - These GPU's are the gamer class. Most work fine with OpenCL/CUDA, however optimized for game use.
- **Quadro** - These GPU's are the workstation class. They will do okay with games, but are optimized for GPGPU. Improved double-precision floating point and memory transfers.
- **Tesla** - These GPU's are the datacenter class. Usually part of an integrated hardware solution. Usually ran "headless". Available on Amazon EC2.

HOW A GAME USES A GPU

- 32-bit (float) is typically used over 64-bit (double)
- Computation is in very short-term computationally intensive bursts(fames)
- Rarely does data “return”. The frame is rendered and we move on.
- GPU holds data (textures) that is relevant between frames. Textures are transferred during initialization.
- Math is important, branching is not. The same thing is done a large number of times.

MY GTX580

GPU FRAMEWORKS

- **CUDA** - CUDA is Nvidia's low-level GPGPU framework.
- **OpenCL** - An open framework supporting CPU's, GPU's and other devices. Managed by the Khronos Group.

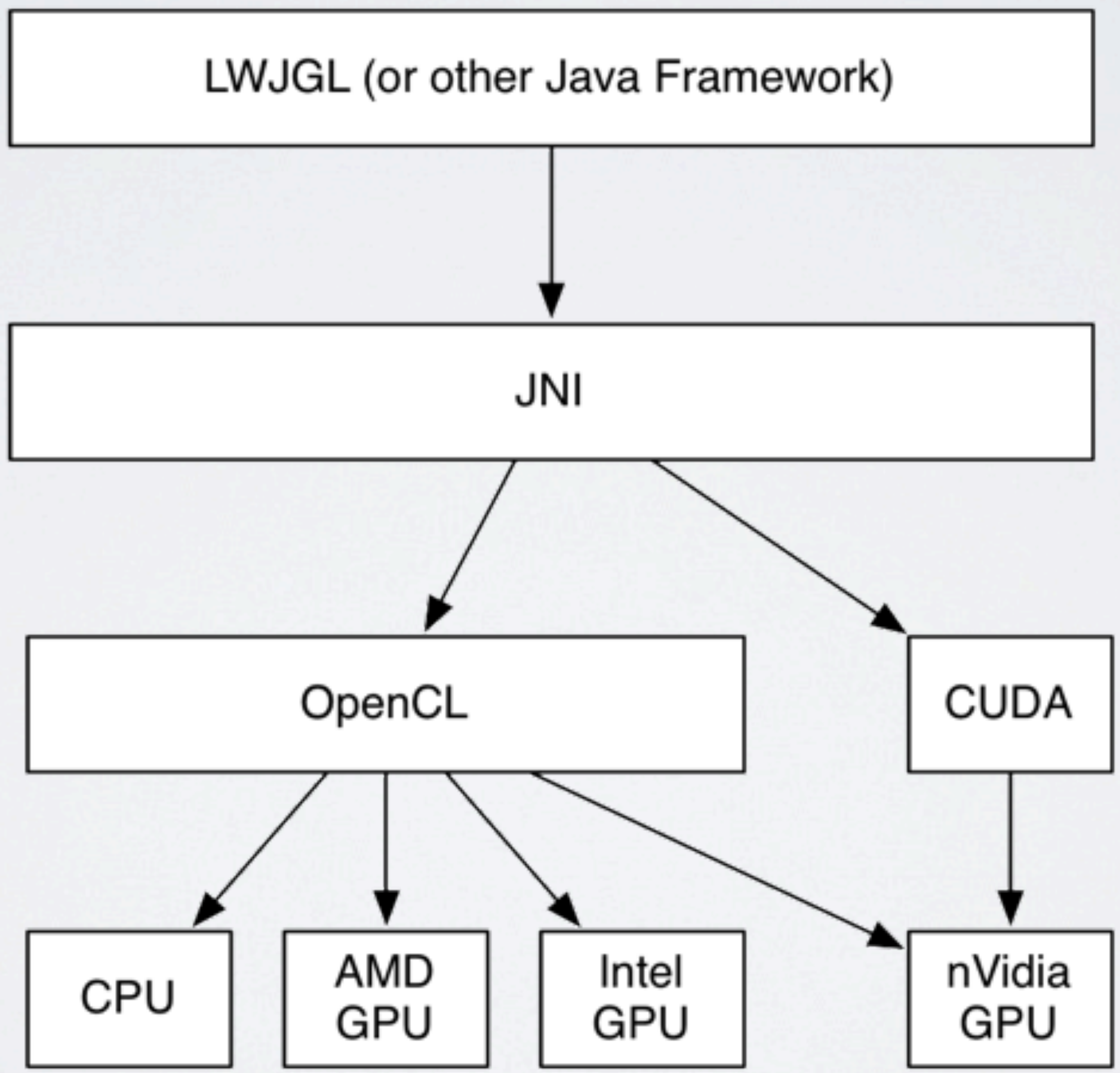
CUDA

- **Reasons to choose CUDA** (pro's)
  - Direct support for BLAS (CUBLAS)
  - Provides better performance on nVidia hardware
  - CUDA is more mature than OpenCL
- **Reasons not to choose CUDA** (con's)
  - No direct support for mixing CPU/GPU
  - Locks your application into nVidia

OPENCL

- **Reasons to choose OpenCL** (pro's)
  - OpenCL supports GPU's, CPU's and other devices
  - OpenCL has wider hardware support
- **Reasons not to choose OpenCL** (con's)
  - Not optimal if you are only targeting nVidia
  - Even with OpenCL you must tune your code for each hardware vendor

TECHNOLOGY STACK



GPU PROGRAMMING FROM JAVA

- Most GPU frameworks are implemented as API's accessible from C/C++ or Fortran
- Fortran is not dead yet! It still has its place in HPC.
- Java cannot directly access CUDA or OpenCL
- Native code must be used
- Java Native Interface (JNI) must be used to access either CUDA or OpenCL

HIGHER LEVEL JAVA GPU API'S

- These API's handle the native interface to OpenCL and CUDA. Typically distributed as a JAR that contains embedded native libraries for Windows, Linux & Mac.
- **LWJGL** (http://www.lwjgl.org/) - My 1st choice.
- **JOCL** (http://www.jocl.org) - My 2nd choice.
- **JCUDA** (http://www.jcuda.de/)
- **Aparapi** (http://code.google.com/p/aparapi/) - Interesting idea.
- **JavaCL** (http://code.google.com/p/javacl/)

LWJGL

- Light Weight Java Game Library
- Very true to OpenCL and OpenGL standards
- Does not bog you down with JNI cruft
- Largest of the projects mentioned
- Engine used by several popular games, such as Minecraft.

JOCL & JCUDA

- JOCL is used for OpenCL
- JCUDA is used for CUDA
- Both attempt reassembly faithful implementations of the OpenCL C-based API
- Not object oriented
- Code executed on GPU (kernel) must be written in C-like OpenCL or CUDA code.

JAVACL

- JavaCL is used for OpenCL
- JavaCL provides a higher-level "object-based" interface to OpenCL
- Code executed on GPU (kernel) must be written in C-like OpenCL code.

APARAPI

- Aparapi is provided by AMD.
- Aparapi is object oriented
- Code executed on GPU (kernel) is generated from Java bytecode
- No need to learn special C-like OpenCL language that kernels are typically written in
- Not possible to do some of the optimizations typically done in hand-crafted kernels
- Seen (by AMD FAQ) as a gateway technology to more advanced OpenCL based techniques.

GPU KERNELS

- The code actually executed on a GPU is a kernel
- Kernels execute totally within the GPU
- Data is typically copied directly to the GPU, processed, and then copied back to the host PC
- Kernels have no access to the operating system services
- Both CUDA and OpenCL kernels are written in C-like language
- Some of the latest CUDA GPU's have some ability to directly access main host memory

EXAMPLE KERNEL CODE

```
__kernel void fft1D_1024 (__global float2 *in, __global float2 *out,
                         __local float *sMemx, __local float *sMemy) {
  int tid = get_local_id(0);
  int blockIdx = get_group_id(0) * 1024 + tid;
  float2 data[16];

  // starting index of data to/from global memory
  in = in + blockIdx;  out = out + blockIdx;

  globalLoads(data, in, 64); // coalesced global reads
  fftRadix16Pass(data);      // in-place radix-16 pass
  twiddleFactorMul(data, tid, 1024, 0);

  // local shuffle using local memory
  localShuffle(data, sMemx, sMemy, tid, (((tid & 15) * 65) + (tid >> 4)));
  fftRadix16Pass(data);               // in-place radix-16 pass
  twiddleFactorMul(data, tid, 64, 4); // twiddle factor multiplication
  localShuffle(data, sMemx, sMemy, tid, (((tid >> 4) * 64) + (tid & 15)));
  // four radix-4 function calls
  fftRadix4Pass(data);      // radix-4 function number 1
  fftRadix4Pass(data + 4);  // radix-4 function number 2
  fftRadix4Pass(data + 8);  // radix-4 function number 3
  fftRadix4Pass(data + 12); // radix-4 function number 4

 globalStores(data, out, 64);
}
```

DESIGNING KERNELS

- Kernels typically require a great deal of "tweaking" to get optimal performance
- Memory transfers are expensive
- Minimal branching
- Memory optimization
- Massively Parallel
- Multiple GPU's

MINIMIZE MEMORY TRANSFERS

- Transfers between host and GPU are expensive
- Design kernels to keep data on the GPU
- CUDA provides pinned memory to speed up frequent memory transfers
- Data can be transferred between the GPU and ho between multiple GPU's

MINIMIZE BRANCHING

- Thread Warps execute the same instruction, only different data
- If-statements that execute different blocks of code, in different threads, break this model.
- Think of the threads as blades on a plow moving forward together

MEMORY OPTIMIZATIONS

- OpenCL and CUDA provide several classes of memory
- Each memory class provides different levels of performance
- The patterns that you use to access memory can also greatly impact performance
- Global memory is the largest, but slow
- Register memory is the smallest, but fast
- Floats perform considerably faster than doubles

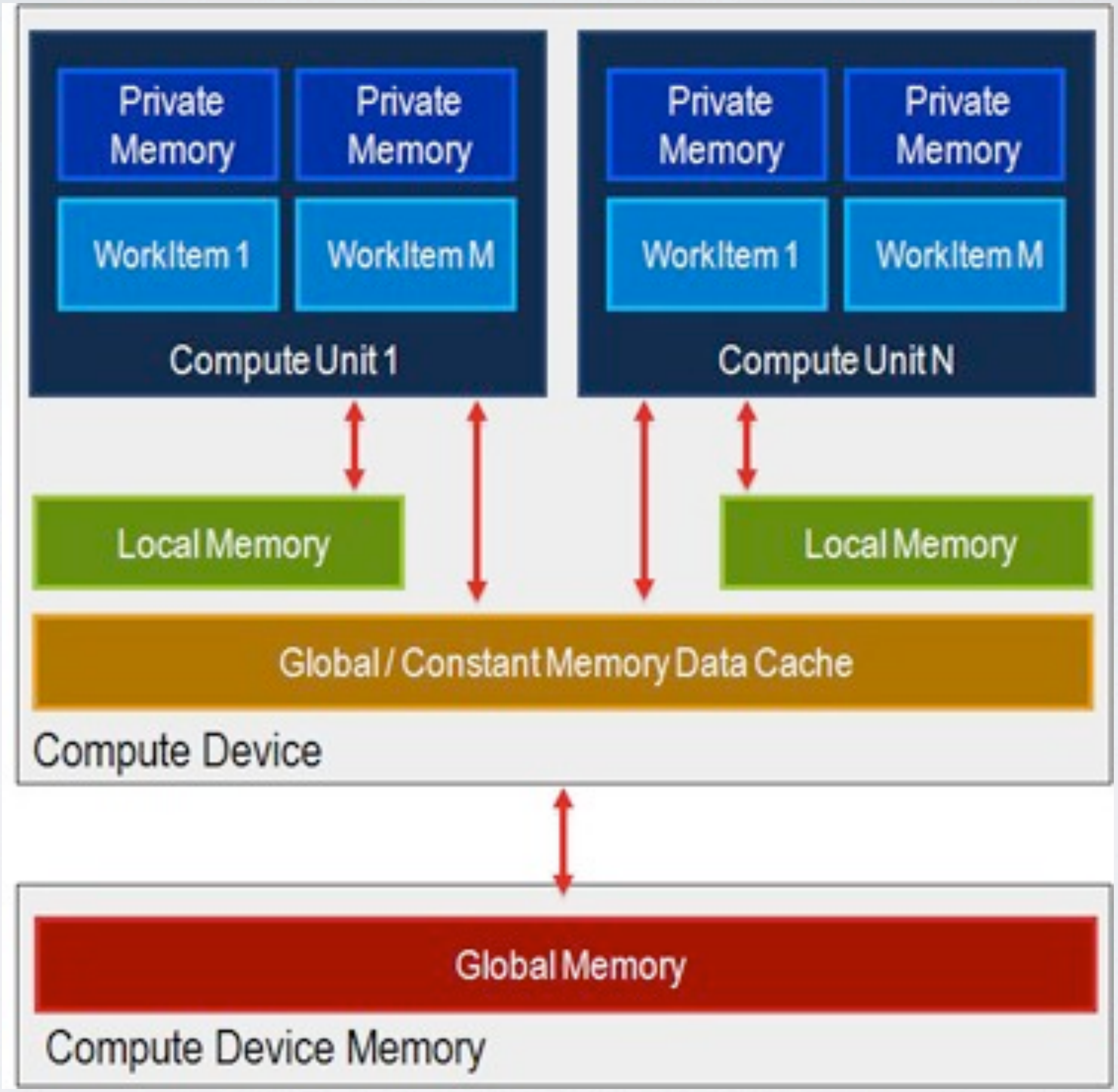
BEST PRACTICES

- Reuse your local variables. These are typically the registers. They are fast! Don't reuse.... inline it!
- You typically have 32k of local memory. Use it as a scratch pad. Reuse/redefine it as your kernel progresses.
- 32-bit (floats) are faster than 64-bit (doubles) they also take half the memory. So you can put twice as many in faster memory. <u>That is a double win!</u>
- Minimize branching. If you can accomplish the branch "in math", do it.
- **Coding standard? Whats a coding standard?**

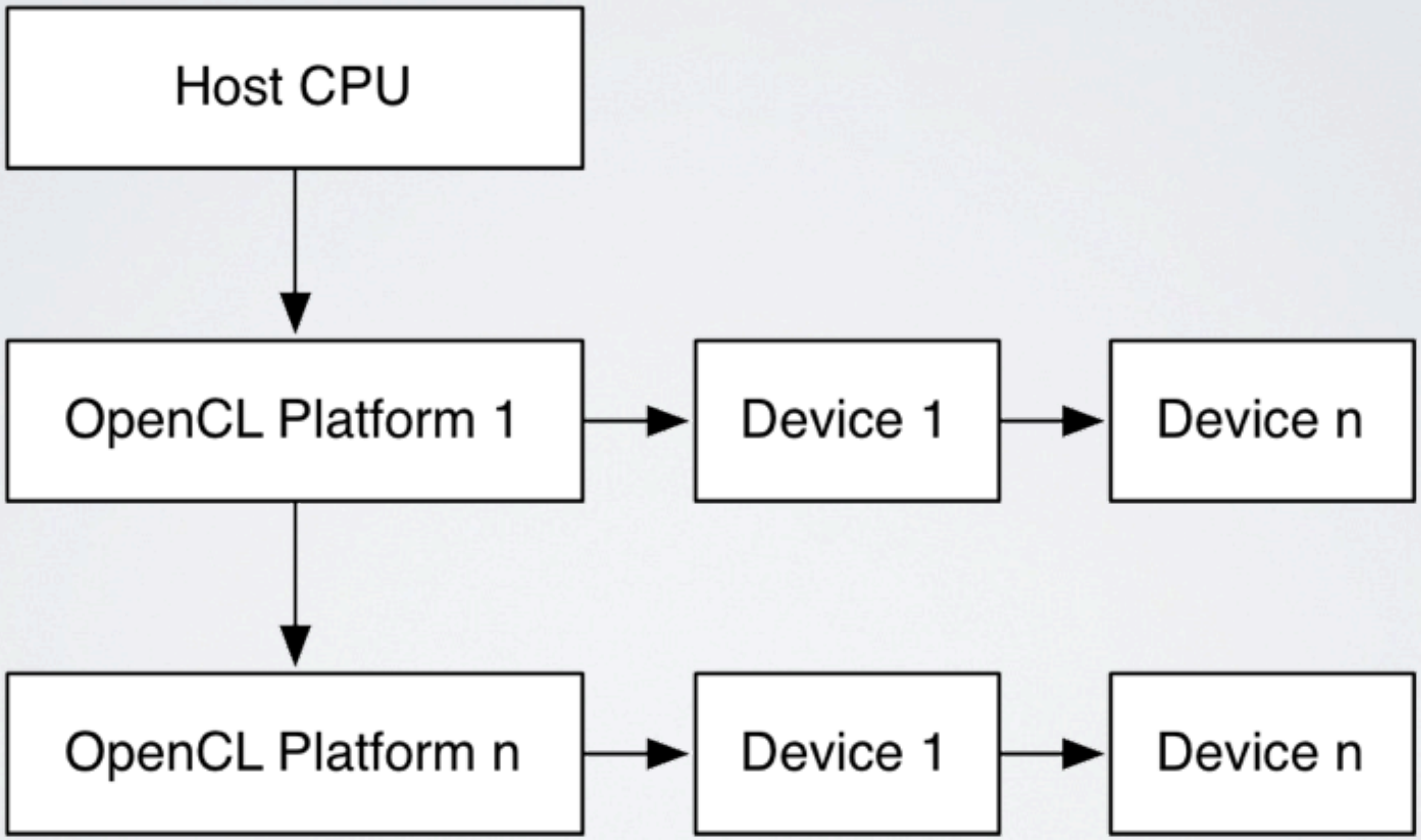
MASSIVELY PARALLEL

- It was hard enough programming for quadcores, now you may have over 512 processors
- Software must be specifically designed to map and reduce over this large array of processors
- Support for multiple GPU's is not automatic, you must separate your job over multiple GPU's
- Don't waste the CPU.

OPENCL MEMORY SPACE



OPENCL STRUCTURE



HELLO WORLD

- https://github.com/jeffheaton/opencl-hello-world
- Good starting point
- Uses LWJGL with Gradle/Gradle Wrapper
- Polls GPU/CPU for basic stats

EXAMPLE OUTPUT

```
Platform #0:Apple
Device #0(CPU):Intel(R) Core(TM) i7-3820QM CPU @ 2.70GHz
    Compute Units: 8 @ 2700 mghtz
    Local memory: 32 KB
    Global memory: 16 GB
Device #1(GPU):GeForce GT 650M
    Compute Units: 2 @ 900 mghtz
    Local memory: 48 KB
    Global memory: 1 GB
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0
+
9.0 8.0 7.0 6.0 5.0 4.0 3.0 2.0 1.0 0.0
=
10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0
```

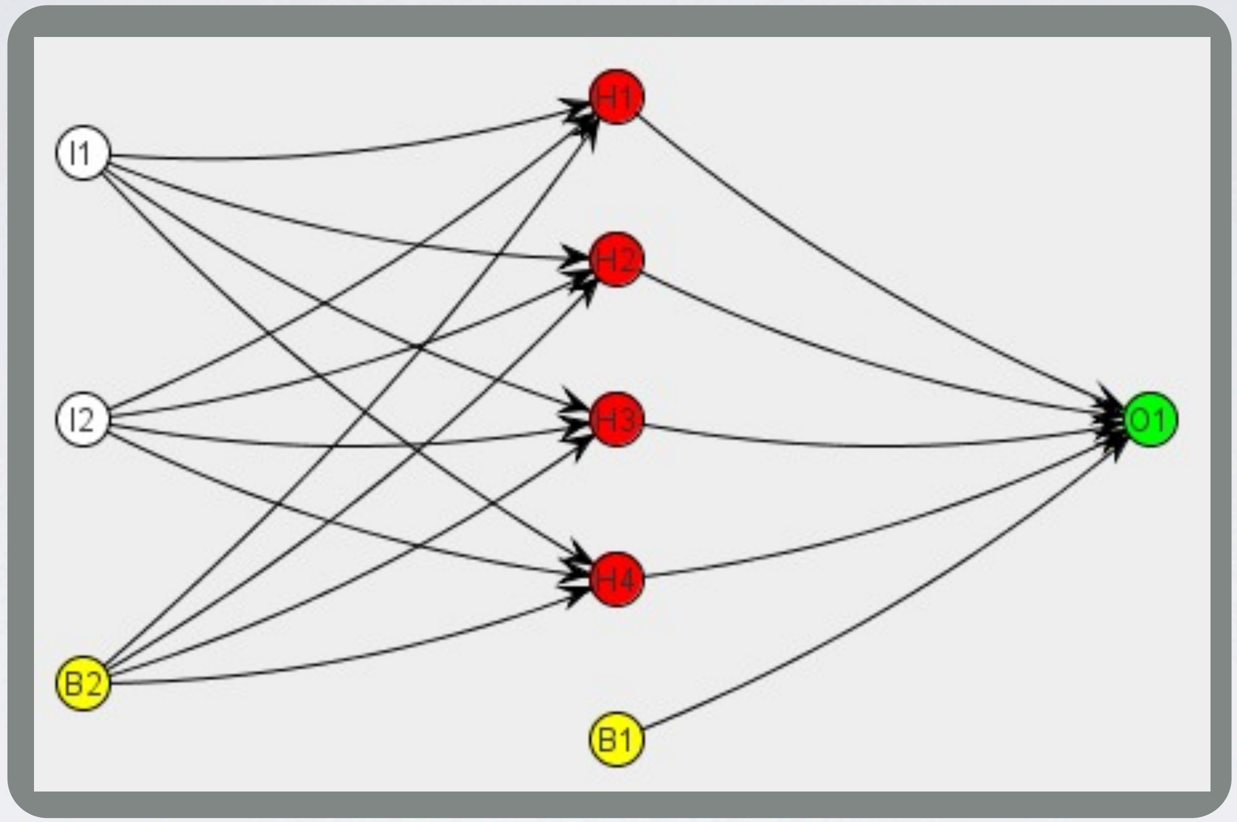
MACHINE LEARNING

$$y_i = f(x_{ij}, \beta_j, \tau_i) + \varepsilon_i$$

Where

- y_i The expected output.
- x_{ij} The input variables.
- β_j Machine learning paramaters (i.e. dimensions, weights or coefficients)
- τ_i Internal state (optional)
- ε_i Error for training case i. (optional)

NEURAL NETWORK

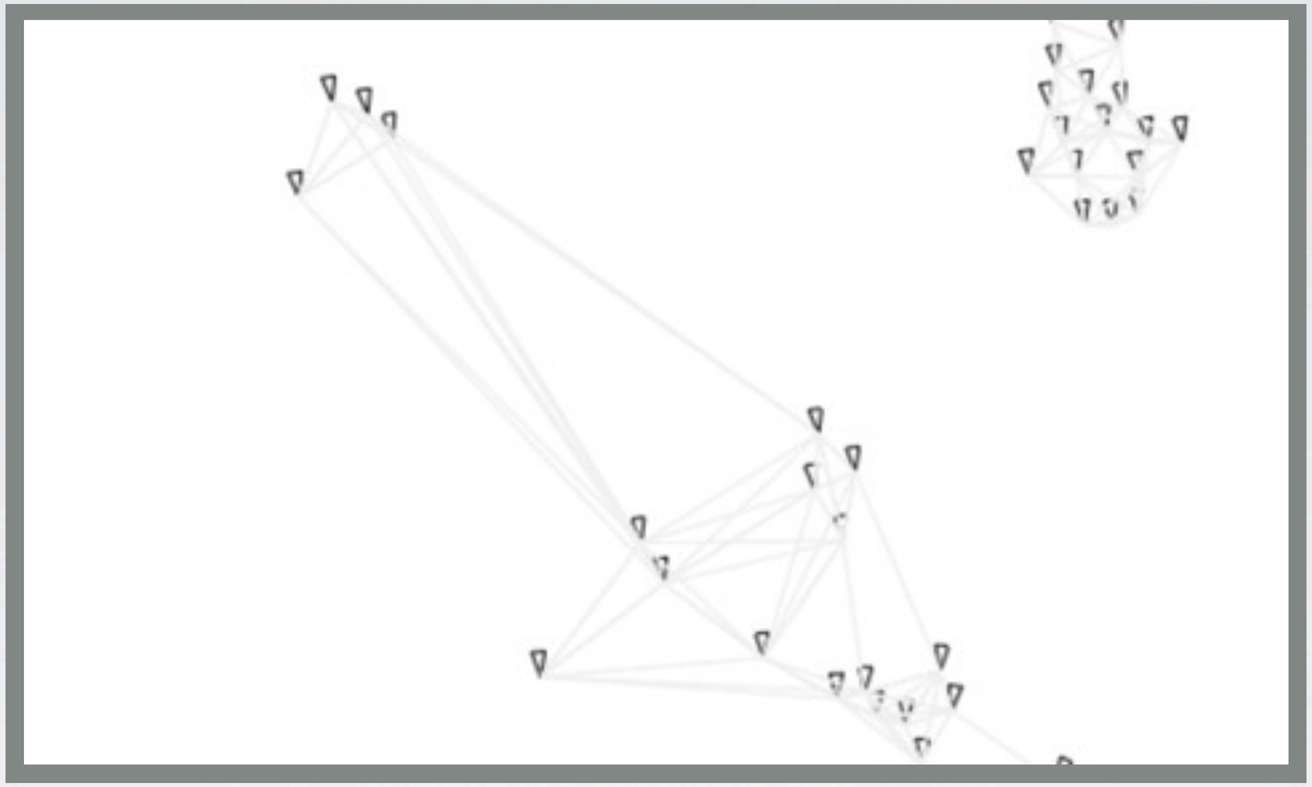


$$y = A\left(\sum_{i=1}^{n} x_i \cdot w_{iy} + b_y\right)$$

Where

- y - Is the output of the neuron.
- $A()$ - Is the Activation Function
- x_i - Is the input to this neuron.
- b_y - Is the bias weight to this neuron.
- w_{iy} - Is the weight between this neuron and each input.

PARTICLE SWARM



- Particles "fly" to optimal solution
- High-dimension space
- One dimension per learned parameter
- Mimics natural swarm/flock/herd/school/etc

PSO CALCULATION

```
v[] = v[] + c1 * rand() * (pbest[] - weight[]) + c2 *
rand() * (gbest[] - weight[])
```

- v[] - The current velocity. It is assigned a new value in the above equation.
- weight[] - The weight, or coordinate, that corresponds the the velocity of the same array index.
- pbest[] - The best weight array found by this particle.
- gbest[] - The best weight array found by any of the particles.
- c1 - The learning rate for the particle to converge to its own best. (def: 2)
- c2 - The learning rate for the particle to converge to the overall best particle. (def: 2)
- rand() - A random number between 0 and 1.

PREDICTIVE MODELING

- https://github.com/jeffheaton/java-export
- Export minute bars of EURUSD since 2000
- Java will create binary training data and randomized neural network
- Encog CUDA trainer will use PSO to train the network balancing GPU and CPU

INPUT DATA

```
<TICKER>,<DTYYYYMMDD>,<TIME>,<OPEN>,<HIGH>,<LOW>,<CLOSE>,<VOL>
EURUSD,20010102,230100,0.9507,0.9507,0.9507,0.9507,4
EURUSD,20010102,230200,0.9506,0.9506,0.9505,0.9505,4
EURUSD,20010102,230300,0.9505,0.9507,0.9505,0.9506,4
EURUSD,20010102,230400,0.9506,0.9506,0.9506,0.9506,4
EURUSD,20010102,230500,0.9506,0.9506,0.9506,0.9506,4
EURUSD,20010102,230600,0.9506,0.9506,0.9506,0.9506,4
EURUSD,20010102,230700,0.9505,0.9507,0.9505,0.9507,4
EURUSD,20010102,230800,0.9507,0.9507,0.9507,0.9507,4
EURUSD,20010102,230900,0.9507,0.9507,0.9507,0.9507,4
EURUSD,20010102,231000,0.9507,0.9507,0.9507,0.9507,4
EURUSD,20010102,231100,0.9507,0.9507,0.9506,0.9507,4
EURUSD,20010102,231200,0.9507,0.9507,0.9507,0.9507,4
EURUSD,20010102,231300,0.9507,0.9507,0.9507,0.9507,4
EURUSD,20010102,231400,0.9507,0.9507,0.9507,0.9507,4
EURUSD,20010102,231500,0.9507,0.9507,0.9507,0.9507,4
EURUSD,20010102,231600,0.9507,0.9507,0.9506,0.9506,4
EURUSD,20010102,232000,0.9507,0.9507,0.9507,0.9507,4
EURUSD,20010102,232100,0.9507,0.9507,0.9507,0.9507,4
EURUSD,20010102,232200,0.9507,0.9507,0.9507,0.9507,4
```

TIME SERIES

I-1	I-2	I-3	I-4	I-5	O-1
1	2	3	4	5	6
2	3	4	5	6	7
3	4	5	6	7	8
4	5	6	7	8	9
5	6	7	8	9	10
6	7	8	9	10	11
7	8	9	10	11	12
8	9	10	11	12	13
9	10	11	12	13	14
10	11	12	13	14	15

CONCLUSIONS

- GPU hardware is designed to perform certain operations very fast.
- GPU's can be used from Java using JNI.
- Not all applications scale to GPU's well.
- GPU's may be a glimpse into the future of CPU's.
- GPU and CPU programming each have their own set of performance rules.