Accelerating Java Workloads via GPUs

Gary Frost

JavaOne 2010 (S313888)

AMD
The future is fusion

Agenda

- The untapped supercomputer in your GPU
- GPUs: Not just for graphics anymore
- What we can offload to the GPU?
- Why we can't offload everything?
- Identifying data-parallel algorithms/workloads
- GPU and Java challenges
- Available Java APIs and bindings
  - JOCL Demo
- Aparapi
  - Aparapi Demo
- Conclusions/Summary
- Q/A

AMD
The future is fusion

The untapped supercomputer in your GPU

- 2000 ASCI RED, Sandia National Laboratories
  - World's #1 supercomputer
  - http://www.top500.org/system/ranking/4428
  - ~3,200 GFLOPS
- 2010 AMD Radeon™ HD 5970
  - ~4,700 GFLOPS

AMD
The future is fusion

GPUs: Not just for graphics anymore

- GPUs originally developed to accelerate graphic operations
- Early adopters realized they could be used for 'general compute' by performing 'unnatural acts' with GPU shader APIs
- OpenGL allows shaders/textures to be compiled and executed via extensions
- OpenCL/GLSL/CUDA standardize and formalize how to express both the GPU compute and the host programming requirements

AMD
The future is fusion

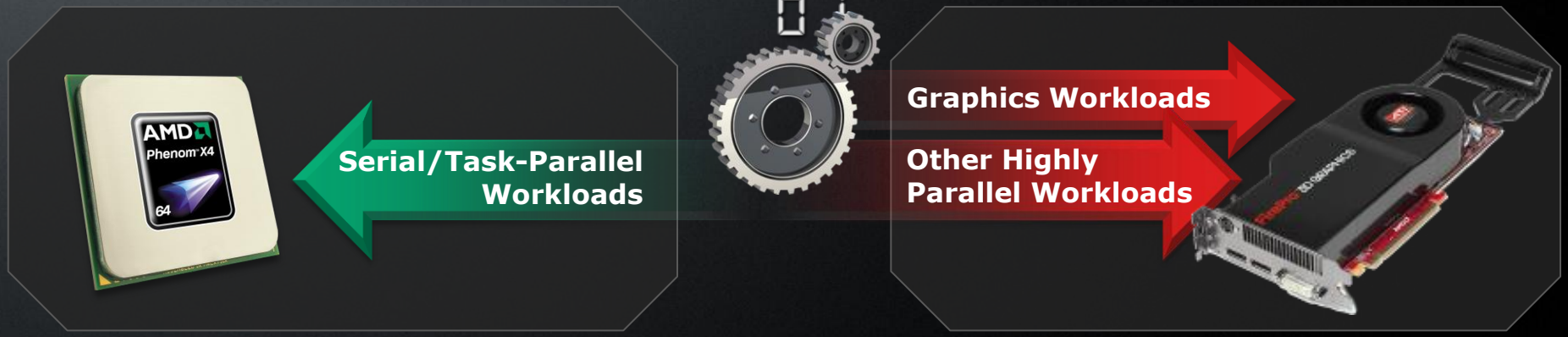
Ideally, we can target compute at the most capable device

CPU excels at sequential, branchy code, I/O interaction, system programming

Most Java applications have these characteristics and excel on the CPU

GPU excels at data-parallel tasks, image processing, data analysis, map reduce

Java is used in the above areas/domains, but does not exploit the capabilities of the GPU as a compute device



AMD
The future is fusion

Ideal data parallel algorithms/workloads

- GPU SIMDs are optimized for data-parallel operations

  Performing the same sequence of operations on different data at the same time

- The body of loops are a good place to look for data-parallel opportunities

```
for (int i=0; i< 100; i++){
      out[i] = in[i]*in[i];
}
```

- Particularly if we can loop in any order and get same result

```
for (int i=99; i>=0; i--){ // backwards
      out[i] = in[i]*in[i];
}
```


AMD
The future is fusion

Watch out for dependencies and bottlenecks

- Data dependencies can violate the 'in any order' guideline

```
for (int i=1; i< 100; i++)
      out[i] = out[i-1]+in[i];
```

- Mutating shared data can force use of atomic constructs

```
for (int i=0; i< 100; i++)
      sum += in[i];
```

- Sometimes we can refactor to expose some parallelism

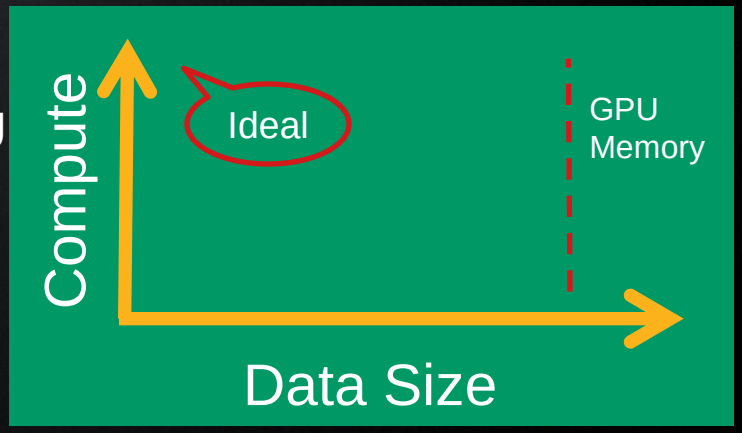
```
for (int n=0; n<10; n++)
   for (int i=0; i<10; i++)
      partial[n] += data[n*10+i];

for (int i=0; i< 10; i++)
      sum+=partial[i];
```


AMD
The future is fusion

Characteristics of an ideal GPU workload

- Looping/searching arrays of primitives
  - 32-/64-bit data types preferred
  - Order of iterations unimportant
  - Minimal data dependencies between iterations
- Each iteration contains sequential code (few branches)
- Good balance between data size (low) and compute (high)
  - Transfer of data to/from the GPU can be costly
  - Trivial compute often not worth the transfer cost
  - May still benefit, by freeing up CPU for other work



AMD
The future is fusion

Fork/Join

Traditionally, our data-parallel code wrapped in some sort of pure Java fork/join framework pattern.

```
int cores = Runtime.getRuntime().availableProcessors();
final int chunk = in.length/cores;
Thread t[] = new Thread[cores];
for(int core=0; core<cores; core++){
   final int start = core*chunk;
   t[core] = new Thread(new Runnable(){
     public void run(){
        for (int i=start;i<start+chunk;i++){
           out[i] = in[i]*in[i];
        }
     }
   });
   t[i].start();
}
for(int core=0; core<cores; core++){
   t[core].join();
}
```


AMD
The future is fusion

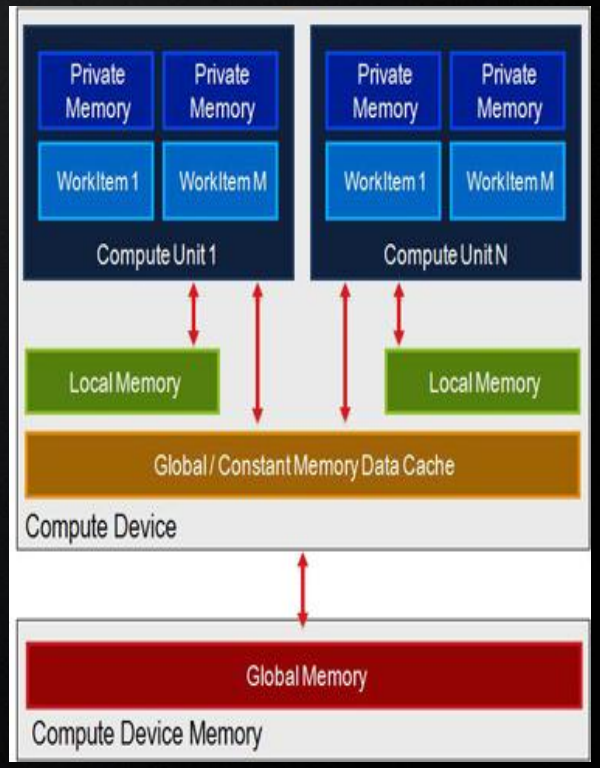
Why GPU programming is unnatural for Java developers

- GPU languages/runtimes optimized for vector types

```
float3 f = {x,y,z};

f += (float3){0,10,20);
```

- GPU languages/runtimes expose explicit memory model semantics
- Understanding how to use this information can reap performance benefits
- Moving data between host CPU and target GPU can be expensive, especially when negotiating with a garbage collector



AMD
The future is fusion

Why GPU programming is unnatural for Java developers

- Most GPU APIs require developing in a domain-specific language (OpenCL, GLSL, or CUDA)

```
__kernel void squares(__global const float *in, __global float *out){
    int gid = get_global_id(0);
    out[gid] = in[gid] * in[gid];
}
```

- As well as the 'host' CPU-based code to
  - Select/Initialize execution device
  - Compile 'Kernel' for a selected device
  - Allocate or define memory buffers for args
  - Write/Send args to device
  - Execute the kernel
  - Read results back from the device

AMD
The future is fusion

Current options available to Java developers

- Java + JNI + (OpenCL/GCGPU/CUDA)
  - Write GPU code in OpenCL/GLSL/CUDA
  - Write 'host' code in C/C++
  - Wrap host entry points in Java JNI methods
  - Write application code in Java using JNI calls

- Use an available Java binding (JOpenCL, JOCL, JavaCL/OpenCL4Java, JOGL+GLSL, JCUDA)
  - Write GPU code in OpenCL/GLSL/GLslang/CUDA
  - Write your host code and application using Java bindings

AMD
The future is fusion

JOCL A Java OpenCL Binding

- http://www.jocl.org/
- API maps very closely to the original API of OpenCL.
- Functions provided as static methods (delegate to OpenCL via JNI)
- JOCL distribution includes :-
  - JOCL<ver>.jar
  - JOCL<ver>.dll/.so
- "…semantics and signatures of methods have been kept consistent with the original library functions, except for the language-specific limitations of Java."
- Kernel implemented in OpenCL and dispatched to the GPU via Java API.

AMD
The future is fusion

Using JOCL OpenCL Java Bindings

Calculate an array of square values

Create in and out array to hold data

```
float in[] = new float[size];
float out[] = new float[size];
for (int i=0; i<size; i++) {
   in[i] = i;
}
```

Perform the parallel equivalent of

```
for (int i=0; i<size; i++) {
   out[i] = in[i]*in[i];
}
```

Print the results

```
for (float f: out) {
   System.out.printf("%5.2f,", f);
}
```

```
import static org.jocl.CL.*;
import org.jocl.*;

public class Sample {
   public static void main(String args[]) {
      // Create input- and output data
      int size = 10;
      float inArr[] = new float[size];
      float outArray[] = new float[size];
      for (int i=0; i<size; i++) {
         inArr[i] = i;
      }

      Pointer in = Pointer.to(inArr);
      Pointer out = Pointer.to(outArray);

      // Obtain the platform IDs and initialize the context properties
      cl_platform_id platforms[] = new cl_platform_id[1];
      clGetPlatformIDs(1, platforms, null);
      cl_context_properties contextProperties = new cl_context_properties();
      contextProperties.addProperty(CL_CONTEXT_PLATFORM, platforms[0]);

      // Create an OpenCL context on a GPU device
      cl_context context = clCreateContextFromType(contextProperties,
            CL_DEVICE_TYPE_CPU, null, null, null);

      // Obtain the cl_device_id for the first device
      cl_device_id devices[] = new cl_device_id[1];
      clGetContextInfo(context, CL_CONTEXT_DEVICES,
            Sizeof.cl_device_id,  Pointer.to(devices), null);

      // Create a command-queue
      cl_command_queue commandQueue =
         clCreateCommandQueue(context, devices[0], 0, null);

      // Allocate the memory objects for the input- and output data
      cl_mem inMem = clCreateBuffer(context, CL_MEM_READ_ONLY |
            CL_MEM_COPY_HOST_PTR,
            Sizeof.cl_float * size, in, null);
      cl_mem outMem = clCreateBuffer(context, CL_MEM_READ_WRITE,
            Sizeof.cl_float * size, null, null);

      // Create the program from the source code
      cl_program program = clCreateProgramWithSource(context, 1, new String[]{
         "__kernel void sampleKernel("+
         "    __global const float *in,"+
         "    __global float *out){"+
         "    int gid = get_global_id(0);"+
         "    out[gid] = in[gid] * in[gid];"+
         "}"
      }, null, null);

      // Build the program
      clBuildProgram(program, 0, null, null, null, null);

      // Create and extract a reference to the kernel
      cl_kernel kernel = clCreateKernel(program, "sampleKernel", null);

      // Set the arguments for the kernel
      clSetKernelArg(kernel, 0, Sizeof.cl_mem, Pointer.to(inMem));
      clSetKernelArg(kernel, 1, Sizeof.cl_mem, Pointer.to(outMem));

      // Execute the kernel
      clEnqueueNDRangeKernel(commandQueue, kernel,
            1, null,  new long[]{inArray.length}, null, 0, null, null);

      // Read the output data
      clEnqueueReadBuffer(commandQueue, outMem, CL_TRUE, 0,
            outArray.length * Sizeof.cl_float, out, 0, null, null);

      // Release kernel, program, and memory objects
      clReleaseMemObject(inMem);
      clReleaseMemObject(outMem);
      clReleaseKernel(kernel);
      clReleaseProgram(program);
      clReleaseCommandQueue(commandQueue);
      clReleaseContext(context);

      for (float f:outArray){
         System.out.printf("%5.2f, ", f);
      }

   }
}
```


AMD
The future is fusion

An example using JOCL

```
Pointer in = Pointer.to(inArr);
Pointer out = Pointer.to(outArray);

// Get platform IDs and initialize the context
cl_platform_id platforms[] = new cl_platform_id[1];
clGetPlatformIDs(1, platforms, null);
cl_context_properties contextProperties =
    new cl_context_properties();
contextProperties.addProperty(CL_CONTEXT_PLATFORM, platforms[0]);

// Create an OpenCL context on a GPU device
cl_context context = clCreateContextFromType(contextProperties,
    CL_DEVICE_TYPE_CPU, null, null, null);

// Obtain the cl_device_id for the first device
cl_device_id devices[] = new cl_device_id[1];
clGetContextInfo(context, CL_CONTEXT_DEVICES,
    Sizeof.cl_device_id,  Pointer.to(devices), null);
```

```
import static org.jocl.CL.*;
import org.jocl.*;

public class Sample {
   public static void main(String args[]) {
      // Create input- and output data
      int size = 10;
      float inArr[] = new float[size];
      float outArray[] = new float[size];
      for (int i=0; i<size; i++) {
         inArr[i] = i;
      }

      Pointer in = Pointer.to(inArr);
      Pointer out = Pointer.to(outArray);

      // Obtain the platform IDs and initialize the context properties
      cl_platform_id platforms[] = new cl_platform_id[1];
      clGetPlatformIDs(1, platforms, null);
      cl_context_properties contextProperties = new cl_context_properties();
      contextProperties.addProperty(CL_CONTEXT_PLATFORM, platforms[0]);

      // Create an OpenCL context on a GPU device
      cl_context context = clCreateContextFromType(contextProperties,
            CL_DEVICE_TYPE_CPU, null, null, null);

      // Obtain the cl_device_id for the first device
      cl_device_id devices[] = new cl_device_id[1];
      clGetContextInfo(context, CL_CONTEXT_DEVICES,
            Sizeof.cl_device_id,  Pointer.to(devices), null);

      // Create a command-queue
      cl_command_queue commandQueue =
         clCreateCommandQueue(context, devices[0], 0, null);

      // Allocate the memory objects for the input- and output data
      cl_mem inMem = clCreateBuffer(context, CL_MEM_READ_ONLY |
            CL_MEM_COPY_HOST_PTR,
            Sizeof.cl_float * size, in, null);
      cl_mem outMem = clCreateBuffer(context, CL_MEM_READ_WRITE,
            Sizeof.cl_float * size, null, null);

      // Create the program from the source code
      cl_program program = clCreateProgramWithSource(context, 1, new String[]{
         "__kernel void sampleKernel("+
         "    __global const float *in,"+
         "    __global float *out){"+
         "      int gid = get_global_id(0);"+
         "      out[gid] = in[gid] * in[gid];"+
         "}"
      }, null, null);

      // Build the program
      clBuildProgram(program, 0, null, null, null, null);

      // Create and extract a reference to the kernel
      cl_kernel kernel = clCreateKernel(program, "sampleKernel", null);

      // Set the arguments for the kernel
      clSetKernelArg(kernel, 0, Sizeof.cl_mem, Pointer.to(inMem));
      clSetKernelArg(kernel, 1, Sizeof.cl_mem, Pointer.to(outMem));

      // Execute the kernel
      clEnqueueNDRangeKernel(commandQueue, kernel,
            1, null,  new long[]{inArray.length}, null, 0, null, null);

      // Read the output data
      clEnqueueReadBuffer(commandQueue, outMem, CL_TRUE, 0,
            outArray.length * Sizeof.cl_float, out, 0, null, null);

      // Release kernel, program, and memory objects
      clReleaseMemObject(inMem);
      clReleaseMemObject(outMem);
      clReleaseKernel(kernel);
      clReleaseProgram(program);
      clReleaseCommandQueue(commandQueue);
      clReleaseContext(context);

      for (float f:outArray){
         System.out.printf("%5.2f, ", f);
      }

   }
}
```


AMD
The future is fusion

An example using JOCL

```
// Create a command-queue
cl_command_queue commandQueue =
   clCreateCommandQueue(context, devices[0], 0, null);

// Allocate the memory objects for the input and output data
cl_mem inMem = clCreateBuffer(context,
      CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
      Sizeof.cl_float * size, in, null);

cl_mem outMem = clCreateBuffer(context, CL_MEM_READ_WRITE,
      Sizeof.cl_float * size, null, null);
```

```
import static org.jocl.CL.*;
import org.jocl.*;

public class Sample {
   public static void main(String args[]) {
      // Create input- and output data
      int size = 10;
      float inArr[] = new float[size];
      float outArray[] = new float[size];
      for (int i=0; i<size; i++) {
         inArr[i] = i;
      }

      Pointer in = Pointer.to(inArr);
      Pointer out = Pointer.to(outArray);

      // Obtain the platform IDs and initialize the context properties
      cl_platform_id platforms[] = new cl_platform_id[1];
      clGetPlatformIDs(1, platforms, null);
      cl_context_properties contextProperties = new cl_context_properties();
      contextProperties.addProperty(CL_CONTEXT_PLATFORM, platforms[0]);

      // Create an OpenCL context on a GPU device
      cl_context context = clCreateContextFromType(contextProperties,
            CL_DEVICE_TYPE_CPU, null, null, null);

      // Obtain the cl_device_id for the first device
      cl_device_id devices[] = new cl_device_id[1];
      clGetContextInfo(context, CL_CONTEXT_DEVICES,
            Sizeof.cl_device_id,  Pointer.to(devices), null);

      // Create a command-queue
      cl_command_queue commandQueue =
         clCreateCommandQueue(context, devices[0], 0, null);

      // Allocate the memory objects for the input- and output data
      cl_mem inMem = clCreateBuffer(context, CL_MEM_READ_ONLY |
            CL_MEM_COPY_HOST_PTR,
            Sizeof.cl_float * size, in, null);
      cl_mem outMem = clCreateBuffer(context, CL_MEM_READ_WRITE,
            Sizeof.cl_float * size, null, null);

      // Create the program from the source code
      cl_program program = clCreateProgramWithSource(context, 1, new String[]{
         "__kernel void sampleKernel("+
         "   __global const float *in,"+
         "   __global float *out){"+
         "     int gid = get_global_id(0);"+
         "     out[gid] = in[gid] * in[gid];"+
         "}"
      }, null, null);

      // Build the program
      clBuildProgram(program, 0, null, null, null, null);

      // Create and extract a reference to the kernel
      cl_kernel kernel = clCreateKernel(program, "sampleKernel", null);

      // Set the arguments for the kernel
      clSetKernelArg(kernel, 0, Sizeof.cl_mem, Pointer.to(inMem));
      clSetKernelArg(kernel, 1, Sizeof.cl_mem, Pointer.to(outMem));

      // Execute the kernel
      clEnqueueNDRangeKernel(commandQueue, kernel,
            1, null,  new long[]{inArray.length}, null, 0, null, null);

      // Read the output data
      clEnqueueReadBuffer(commandQueue, outMem, CL_TRUE, 0,
            outArray.length * Sizeof.cl_float, out, 0, null, null);

      // Release kernel, program, and memory objects
      clReleaseMemObject(inMem);
      clReleaseMemObject(outMem);
      clReleaseKernel(kernel);
      clReleaseProgram(program);
      clReleaseCommandQueue(commandQueue);
      clReleaseContext(context);

      for (float f:outArray){
         System.out.printf("%5.2f, ", f);
      }

   }
}
```


AMD
The future is fusion

An example using JOCL

```
cl_program program =
    clCreateProgramWithSource(context, 1, new String[]{
         "__kernel void sampleKernel("+
         "   __global const float *in,"+
         "   __global float *out){"+
         "     int gid = get_global_id(0);"+
         "     out[gid] = in[gid] * in[gid];"+
         "}"
      }, null, null);

// Build the program
clBuildProgram(program, 0, null, null, null, null);

// Create and extract a reference to the kernel
cl_kernel kernel = clCreateKernel(program, "sampleKernel", null);
```

```
import static org.jocl.CL.*;
import org.jocl.*;

public class Sample {
   public static void main(String args[]) {
      // Create input- and output data
      int size = 10;
      float inArr[] = new float[size];
      float outArray[] = new float[size];
      for (int i=0; i<size; i++) {
         inArr[i] = i;
      }

      Pointer in = Pointer.to(inArr);
      Pointer out = Pointer.to(outArray);

      // Obtain the platform IDs and initialize the context properties
      cl_platform_id platforms[] = new cl_platform_id[1];
      clGetPlatformIDs(1, platforms, null);
      cl_context_properties contextProperties = new cl_context_properties();
      contextProperties.addProperty(CL_CONTEXT_PLATFORM, platforms[0]);

      // Create an OpenCL context on a GPU device
      cl_context context = clCreateContextFromType(contextProperties,
            CL_DEVICE_TYPE_CPU, null, null, null);

      // Obtain the cl_device_id for the first device
      cl_device_id devices[] = new cl_device_id[1];
      clGetContextInfo(context, CL_CONTEXT_DEVICES,
            Sizeof.cl_device_id,  Pointer.to(devices), null);

      // Create a command-queue
      cl_command_queue commandQueue =
         clCreateCommandQueue(context, devices[0], 0, null);

      // Allocate the memory objects for the input- and output data
      cl_mem inMem = clCreateBuffer(context, CL_MEM_READ_ONLY |
            CL_MEM_COPY_HOST_PTR,
            Sizeof.cl_float * size, in, null);
      cl_mem outMem = clCreateBuffer(context, CL_MEM_READ_WRITE,
            Sizeof.cl_float * size, null, null);

      // Create the program from the source code
      cl_program program = clCreateProgramWithSource(context, 1, new String[]{
         "__kernel void sampleKernel("+
         "   __global const float *in,"+
         "   __global float *out){"+
         "     int gid = get_global_id(0);"+
         "     out[gid] = in[gid] * in[gid];"+
         "}"
      }, null, null);

      // Build the program
      clBuildProgram(program, 0, null, null, null, null);

      // Create and extract a reference to the kernel
      cl_kernel kernel = clCreateKernel(program, "sampleKernel", null);

      // Set the arguments for the kernel
      clSetKernelArg(kernel, 0, Sizeof.cl_mem, Pointer.to(inMem));
      clSetKernelArg(kernel, 1, Sizeof.cl_mem, Pointer.to(outMem));

      // Execute the kernel
      clEnqueueNDRangeKernel(commandQueue, kernel,
            1, null,  new long[]{inArray.length}, null, 0, null, null);

      // Read the output data
      clEnqueueReadBuffer(commandQueue, outMem, CL_TRUE, 0,
            outArray.length * Sizeof.cl_float, out, 0, null, null);

      // Release kernel, program, and memory objects
      clReleaseMemObject(inMem);
      clReleaseMemObject(outMem);
      clReleaseKernel(kernel);
      clReleaseProgram(program);
      clReleaseCommandQueue(commandQueue);
      clReleaseContext(context);

      for (float f:outArray){
         System.out.printf("%5.2f, ", f);
      }

   }
}
```


AMD
The future is fusion

An example using JOCL

```
// Set the arguments for the kernel
clSetKernelArg(kernel, 0, Sizeof.cl_mem, Pointer.to(inMem));
clSetKernelArg(kernel, 1, Sizeof.cl_mem, Pointer.to(outMem));

// Execute the kernel
clEnqueueNDRangeKernel(commandQueue, kernel,
   1, null,  new long[]{inArray.length}, null, 0, null, null);

// Read the output data back into outArr
clEnqueueReadBuffer(commandQueue, outMem, CL_TRUE, 0,
   outArray.length * Sizeof.cl_float, out, 0, null, null);

// Release kernel, program, and memory objects
clReleaseMemObject(inMem);
clReleaseMemObject(outMem);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(commandQueue);
clReleaseContext(context);

// finally print out the results
for (float f:outArray){
   System.out.printf("%5.2f, ", f);
}
```

```
import static org.jocl.CL.*;
import org.jocl.*;

public class Sample {
   public static void main(String args[]) {
      // Create input- and output data
      int size = 10;
      float inArr[] = new float[size];
      float outArray[] = new float[size];
      for (int i=0; i<size; i++) {
         inArr[i] = i;
      }

      Pointer in = Pointer.to(inArr);
      Pointer out = Pointer.to(outArray);

      // Obtain the platform IDs and initialize the context properties
      cl_platform_id platforms[] = new cl_platform_id[1];
      clGetPlatformIDs(1, platforms, null);
      cl_context_properties contextProperties = new cl_context_properties();
      contextProperties.addProperty(CL_CONTEXT_PLATFORM, platforms[0]);

      // Create an OpenCL context on a GPU device
      cl_context context = clCreateContextFromType(contextProperties,
            CL_DEVICE_TYPE_CPU, null, null, null);

      // Obtain the cl_device_id for the first device
      cl_device_id devices[] = new cl_device_id[1];
      clGetContextInfo(context, CL_CONTEXT_DEVICES,
            Sizeof.cl_device_id,  Pointer.to(devices), null);

      // Create a command-queue
      cl_command_queue commandQueue =
         clCreateCommandQueue(context, devices[0], 0, null);

      // Allocate the memory objects for the input- and output data
      cl_mem inMem = clCreateBuffer(context, CL_MEM_READ_ONLY |
            CL_MEM_COPY_HOST_PTR,
            Sizeof.cl_float * size, in, null);
      cl_mem outMem = clCreateBuffer(context, CL_MEM_READ_WRITE,
            Sizeof.cl_float * size, null, null);

      // Create the program from the source code
      cl_program program = clCreateProgramWithSource(context, 1, new String[]{
         "__kernel void sampleKernel("+
         "   __global const float *in,"+
         "   __global float *out){"+
         "     int gid = get_global_id(0);"+
         "     out[gid] = in[gid] * in[gid];"+
         "}"
      }, null, null);

      // Build the program
      clBuildProgram(program, 0, null, null, null, null);

      // Create and extract a reference to the kernel
      cl_kernel kernel = clCreateKernel(program, "sampleKernel", null);

      // Set the arguments for the kernel
      clSetKernelArg(kernel, 0, Sizeof.cl_mem, Pointer.to(inMem));
      clSetKernelArg(kernel, 1, Sizeof.cl_mem, Pointer.to(outMem));

      // Execute the kernel
      clEnqueueNDRangeKernel(commandQueue, kernel,
            1, null,  new long[]{inArray.length}, null, 0, null, null);

      // Read the output data
      clEnqueueReadBuffer(commandQueue, outMem, CL_TRUE, 0,
            outArray.length * Sizeof.cl_float, out, 0, null, null);

      // Release kernel, program, and memory objects
      clReleaseMemObject(inMem);
      clReleaseMemObject(outMem);
      clReleaseKernel(kernel);
      clReleaseProgram(program);
      clReleaseCommandQueue(commandQueue);
      clReleaseContext(context);

      for (float f:outArray){
         System.out.printf("%5.2f, ", f);
      }

   }
}
```


AMD
The future is fusion

JOCL demo

AMD
The future is fusion

Aparapi (A PARallel API)

- Extending 'write once/run anywhere' to include the GPU
- A Java API for expressing data-parallel workloads
  - No C/C++/OpenCL/GLSL/CUDA coding required
  - Enables/simplifies coding of data-parallel algorithms in Java
  - Code developed using preferred Java tools/IDEs
- A runtime environment capable of
  - Executing using a Java Pool if necessary
  - Optionally offloading to GPU by converting Java bytecode to OpenCL on the fly
- Alpha available 9/20/2010
  - http://developer.amd.com/aparapi

AMD
The future is fusion

Aparapi advantages

- Development
  - Write code once in Java
  - Developer codes against Aparapi API (extend Kernel class)
  - Compiles using standard Java compiler (javac)/IDE of choice
  - Test/debug logic using Java tools (using the thread pool implementation)
- Runtime
  - Execute using standard Java JVM

```
if (Platform supports OpenCL and workload can be converted)
    Convert to OpenCL and execute on the GPU
else
    Fallback to using the pure Java implementation
```


AMD
The future is fusion

Aparapi implementation of the Square example

```
import com.amd.aparapi.Kernel;
…
final float[] in = new float[1024];
final float[] out = new float[in.length];
// populate in[1..in.length] omitted

Kernel kernel = new Kernel(){
  @Override public void run() {
    int gid = getGlobalId();
    out[gid] = in[gid]*in[gid];
  }
};

kernel.execute(in.length);
for (float f:out){
   System.out.println(f);
}
```


AMD
The future is fusion

Aparapi

- Developer extends ‘com.amd.aparapi.Kernel’ class
  - Overrides Kernel.run() to implement data parallel algorithm
  - Kernel is a template for execution
    - Cloned as needed for Java execution
    - Bytecode reified into OpenCL equivalent for dispatch to GPU
  - Kernel.execute(int size) initiates execution
    - On first run, Kernel determines ‘how’ to run
      - OpenCL vs. Java thread pool
      - Decision ‘cached’ for future invocations
    - Kernel.run() called once per ‘work item’ with globalID set to 0..size
    - Blocks until execution is complete
    - Results available in Kernel fields (or captured fields) after Kernel.execute() returns

AMD
The future is fusion

Aparapi: Converting bytecode to OpenCL

- Like Jode/Mocha/Jad, except generates OpenCL
  - Parse/Analyze the bytecode of the Kernel.run() and all methods reachable from Kernel.run()
  - Create IR of reachable methods
  - Bail (as fast as possible) if code contains artifacts that can't be represented in OpenCL
    - `System.out.println(), try/catch`
  - Identify control flow and basic blocks
    - `if(){}/if(){}else{}/while(){}/for(){}/(exp1)?exp2:exp3`
  - Generate OpenCL from the IR
    - Via a Visitor pattern which traverses the IR tree and produces OpenCL source

AMD
The future is fusion

Aparapi: No need for host code

- On first Kernel.execute() call
  - Convert bytecode to OpenCL
  - Create OpenCL context for GPU device
  - Create args and buffers for passing to generated Kernel
    - We can determine whether run() call chain reads/writes to arrays by looking at the bytecode so we can deduce read, read+write, and write-only buffer accesses
- For all Kernel.execute() calls
  - Pin all accessed arrays (so that GC doesn't move them)
  - Enqueue required buffer writes from Java primitive arrays
  - Execute Kernel
  - Enqueue required buffer reads back into Java arrays
  - Unpins arrays

AMD
The future is fusion

Aparapi demo

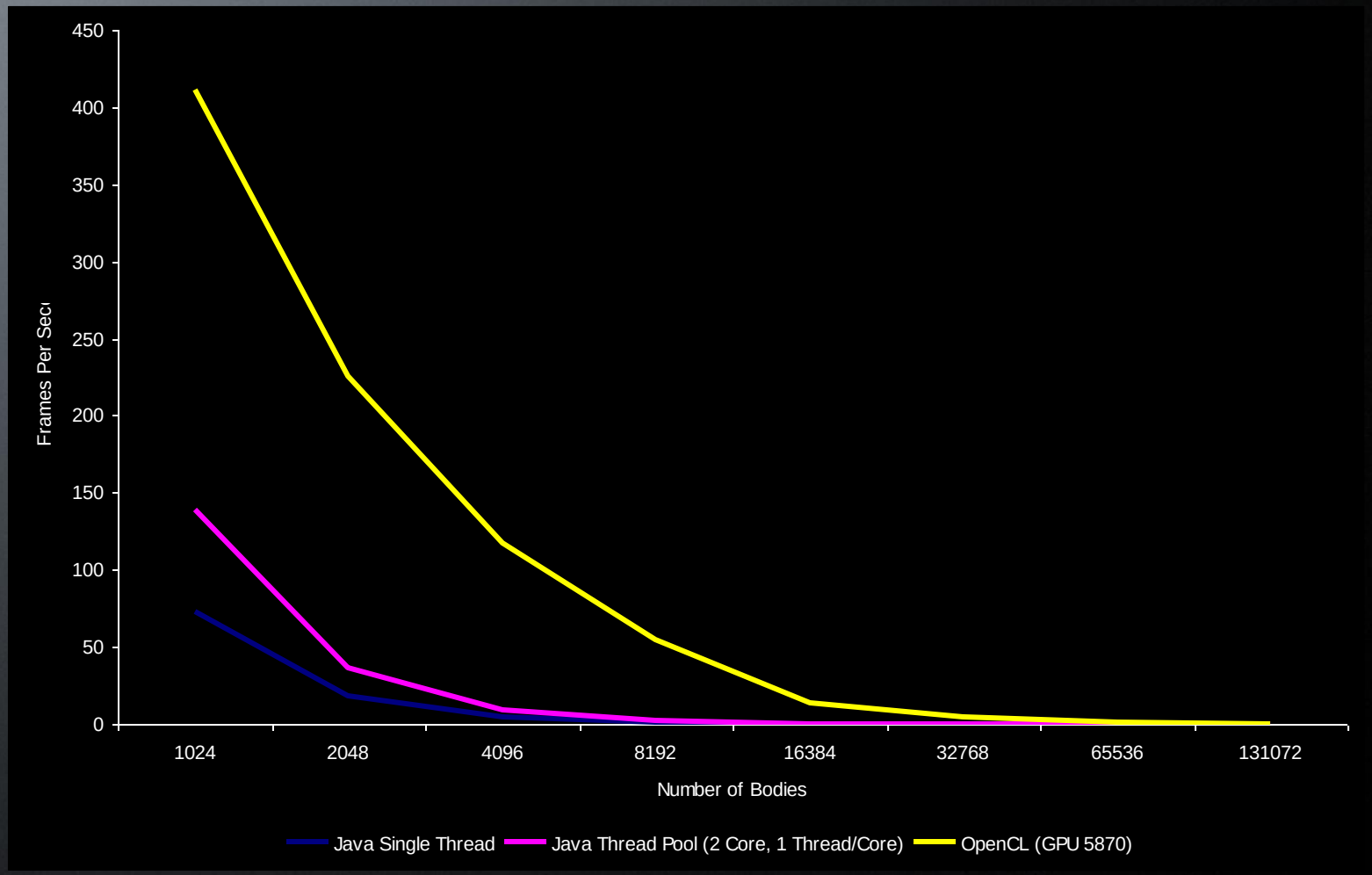
AMD
The future is fusion

Aparapi: NBody performance

- NBody is a common OpenCL/CUDA benchmark/demo
  - Determine the positions of N bodies, calculating the gravitational effect that each body has on every other body
  - C++/C version shipped with AMD Stream SDK
  - Essentially a N^2 space problem
    - If we double the number of bodies, we perform four times the positional calculations
- Following chart compares
  - Naïve Java version (single loop) (blue)
  - Aparapi version using Java Thread Pool (magenta)
  - Aparapi version offloading via OpenCL to ATI Radeon™ HD 5870 (yellow)

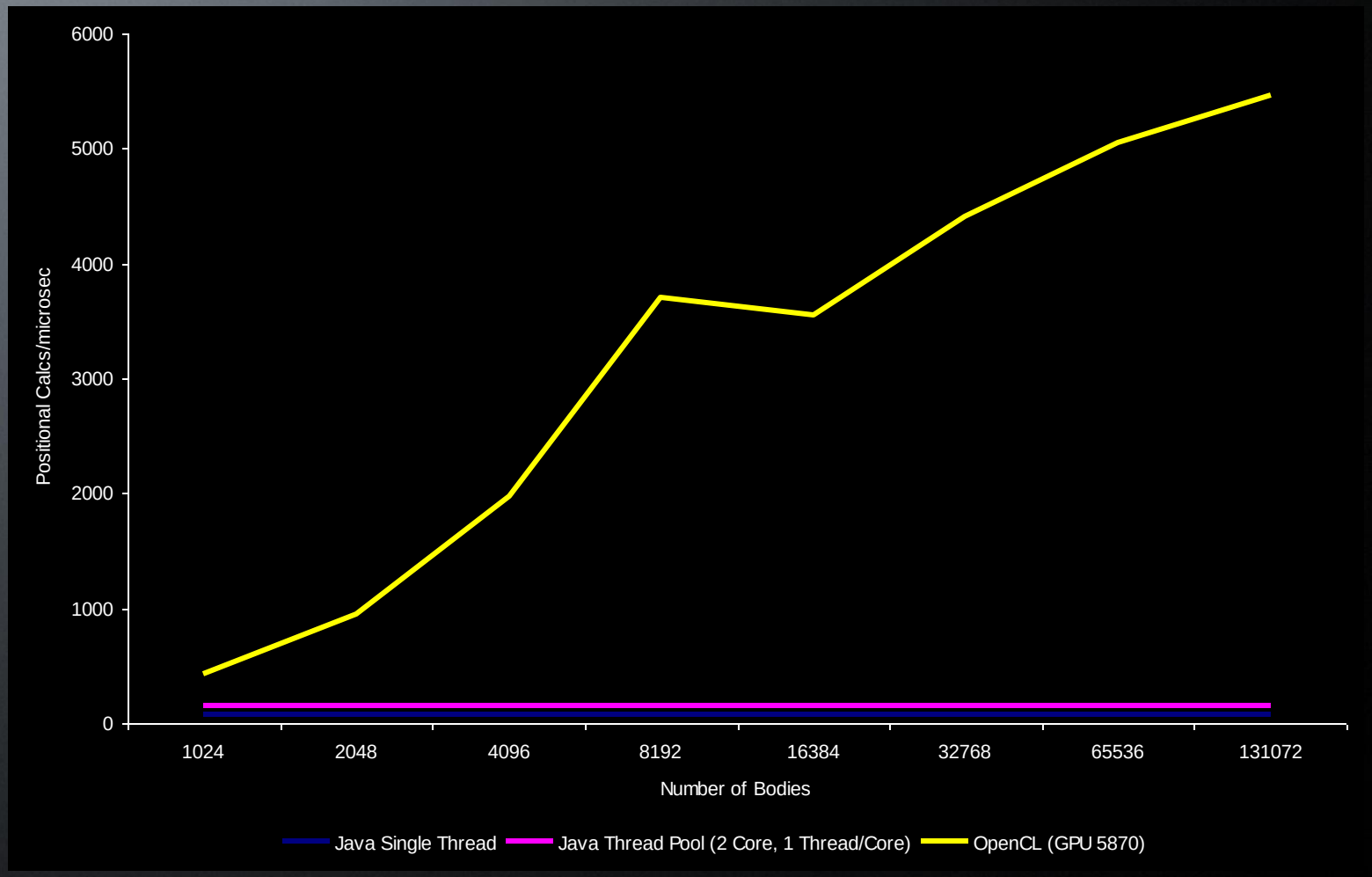
AMD
The future is fusion

Aparapi: NBody FPS for various 'body counts'



AMD
The future is fusion

Aparapi: NBody calcs/microsec for various 'body counts'



AMD
The future is fusion

Summary

- GPUs offer unprecedented performance for the appropriate workload
- Don’t assume everything can/should execute on the GPU
- Look for ‘Islands of parallel in a sea of sequential’
- Consider using one of the available Java bindings for OpenCL or CUDA
- Aparapi provides an ideal framework for executing data-parallel code on the GPU
- If you are comfortable with JNI and with C programming, consider learning OpenCL or CUDA and writing custom Kernels to perform numeric intensive tasks

AMD
The future is fusion

Links/Info

- AMD OpenCL Zone

  http://developer.amd.com/OpenCLZone

- Aparapi 'alpha' downloads

  http://developer.amd.com/aparapi

- Data-parallel papers/info

  http://groups.csail.mit.edu/mac/users/gjs/6.945/readings/MITApril2009Steele.pdf

  http://cva.stanford.edu/classes/cs99s/papers/hillis-steele-data-parallel-algorithms.pdf

AMD
The future is fusion

Java Bindings

- Jgpu
  - https://jgpu.dev.java.net/
- Jopencl
  - http://sourceforge.net/projects/jopencl/
- Javacl
  - http://code.google.com/p/javacl
- Jcuda
  - http://www.jcuda.org
- ScalaCl
  - http://code.google.com/p/scalacl/
- JOCL
  - http://www.jocl.org/

AMD
The future is fusion

Disclaimer & Attribution

DISCLAIMER

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

ATTRIBUTION

© 2010 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, ATI, the ATI Logo, Radeon, FirePro, FireStream and combinations thereof are trademarks of Advanced Micro Devices, Inc. Microsoft, Windows, Windows Vista, and DirectX are registered trademarks of Microsoft Corporation in the United States and/or other jurisdictions. Other names are for informational purposes only and may be trademarks of their respective owners.

OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos.