GPU Programmierung mit Java

Heiko Spindler
Freelancer / Software-Architekt

heise online > News > 2017 > KW 27 > Wegen Mining-Boom: Preise von AMD- und Nvidia-Grafikkarten steigen we

Wegen Mining-Boom: Preise von AMD- und Nvidia-Grafikkarten steigen weiter

05.07.2017 15:23 Uhr – Martin Fischer vorlesen

Questions

What do you have in your Notebook or PC?

- CPU (Cores, Speed, Memory)?
- GPU (Cores, Speed, Memory)?

Demo Time

Demo Output

```
Machine contains 1 OpenCL platforms
Platform 0{
   Name    : "Apple"
   Vendor  : "Apple"
   Version : "OpenCL 1.2 (Aug 10 2016 17:16:39)"
   Platform contains 2 OpenCL devices
   Device 1{
       Type                     : GPU
       GlobalMemSize            : 1610612736
       LocalMemSize             : 65536
       MaxComputeUnits          : 40
       MaxWorkGroupSizes        : 512
       MaxWorkItemDimensions    : 3
   }
}
```

GPUs

	Radeon RX Vega 64	Radeon RX Vega 56	Radeon R9 Fury X	GeForce GTX 1080	GeForce GTX 1080Ti
GPU	Vega 10	Vega 10	Fiji	GP104	GP104
Cores	4096	3584	4096	2560	3584
GPU/Turbo (Mhz)	1247 / 1546	1156 / 1471	1050	1607 / 1733	1480 / 1582
TFlops	12,66	10,5	8,7	8,87	11,34
Memory	8 GB HBM2	8 GB HBM2	4 GB HBM1	8 GB GDDR5X	11 GB GDDR5X

Quelle: https://www.heise.de/newsticker/meldung/Ultrakompakte-Nano-Variante-der-AMD-Radeon-RX-Vega-in-Vorbereitung-3788640.html

Two Worlds: CPU vs GPU

CPU

- General-purpose capabilities
- Established technology
- Optimal for concurrent processes but not on a large scale
- Task driven
- Context switch is expensive
- Communicates to other parts of the system (IO, memory, network, GPU, …)

Two Worlds: CPU vs GPU

CPU

- General-purpose capabilities
- Established technology
- Optimal for concurrent processes but not on a large scale
- Task driven
- Context switch is expensive
- Communicates to other parts of the system (IO, memory, network, GPU, …)

GPU

- Created specifically for graphics
- Became more capable of general computations
- Data driven
- Context switch is cheap, branching is not cheap

Latency vs Throuput

CPU: Compute one thread fast. Get one answer as fast as possible.

GPU: Finsh processing for the whole set of data as fast as possible.

What is GPGPU?

- General-Purpose Computing on GPUs
- Use the GPU to speed up certain parts of an Application
- Typically used for mathematically intense applications
- A single mid to high-end GPU can accelerate a mathematically intense process

CUDA
(Compute Unified Device Architecture)

Nvidia’s low-level GPGPU framework.

Pro:

- Direct support for BLAS (basic linear algebra subroutines)
- Provides better performance on nVidia hardware
- CUDA is more mature than OpenCL

Cons:

- No direct support for mixing CPU/GPU
- Locks your application into nVidia

OpenCL

An open framework supporting CPU’s, GPU’s and other devices. Managed by the Khronos Group.

Pro:

- OpenCL supports GPU’s, CPU’s and other devices
- OpenCL has wider hardware support

Cons:

- Not optimal if you are only targeting nVidia hardware

JOCL, JavaCL & JCUDA

- JOCL, JavaCL are used for OpenCL
- JCUDA is used for CUDA

- Not object oriented
- Code executed on GPU (kernel) must be written in OpenCL or CUDA code (C-like)

LWJGL

- Light Weight Java Game Library
- Uses CUDA, OpenCL and OpenGL standards
- Hides all the JNI stuff
- Large and mature project
- Used by several popular games and products (e.g. Minecraft)

Demo Time

Aparapi: A PAR{allel} API

- Java library that supports concurrent operations
- Convert Java byte codes to OpenCL at runtime
- Aparapi offers a higher level of abstraction
- Initially developed by AMD
- Released as open source
- The API supports code running on GPUs or CPUs
- GPU operations are executed using OpenCL - Alternative: Can run a Kernel with Java threads

API

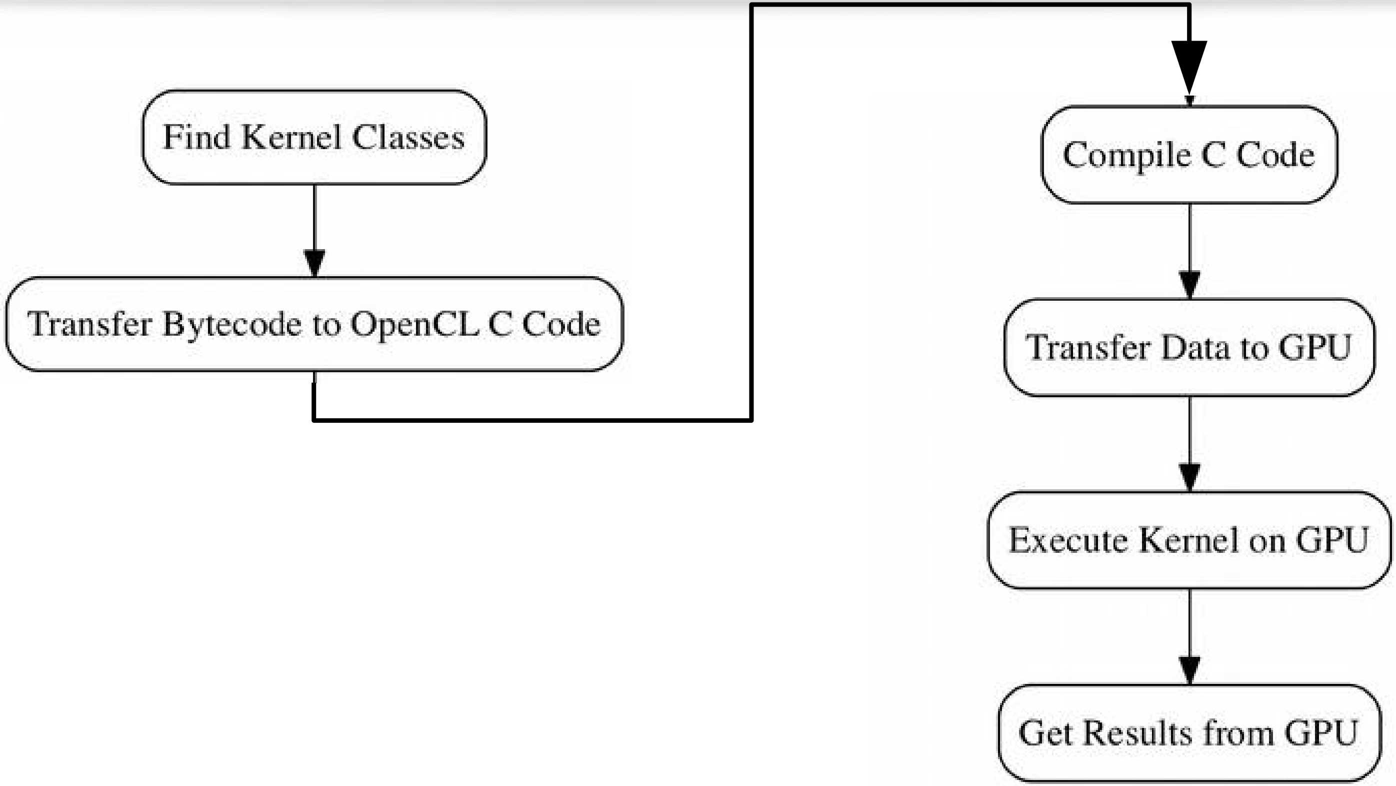
- Aparapi code is located in a class derived from the Kernel class
- Its `execute` method will start the operations
- This will result in an internal call to a `run` method
- The `run` method is executed multiple times on different cores

Simple Kernel

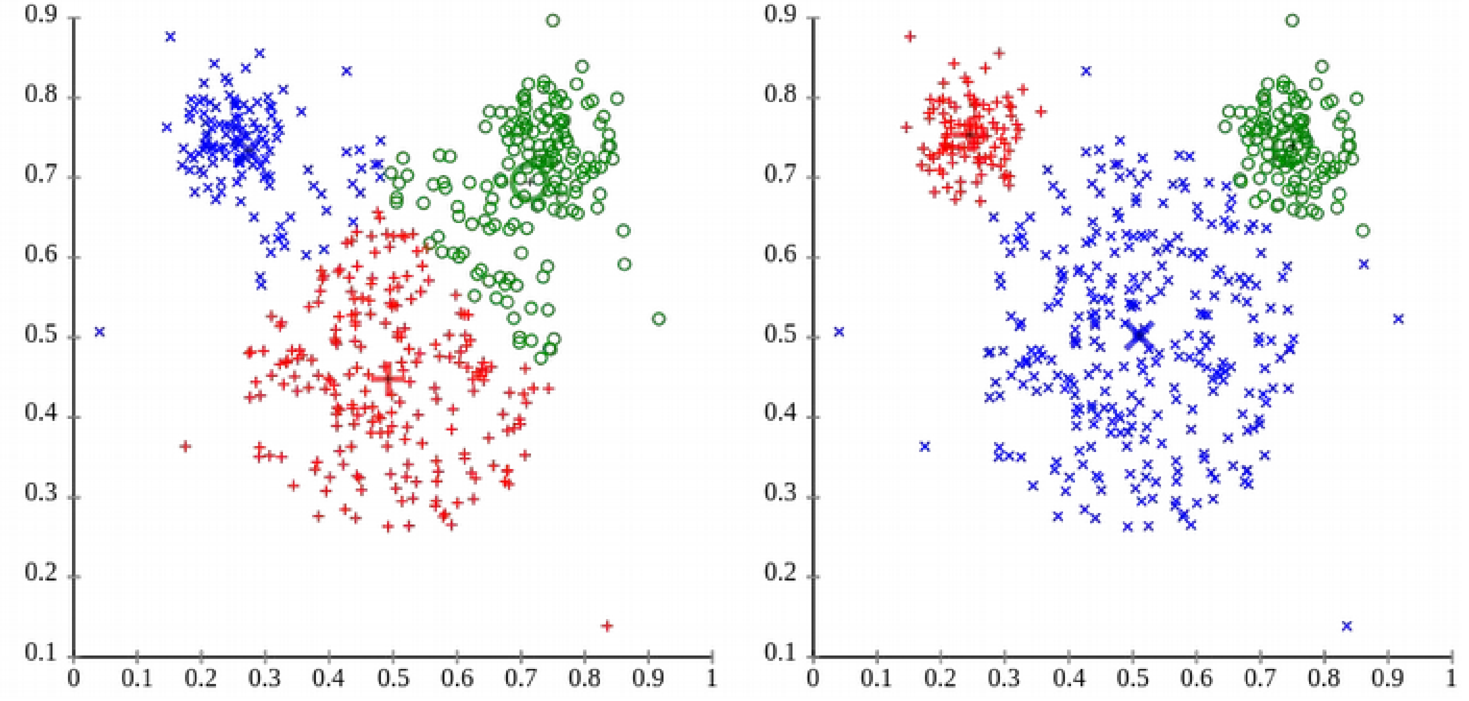
```
final int size = 512;
final float[] input = new float[size];
final float[] squares = new float[size];

for (int i = 0; i < size; i++) {
   values[i] = i; (1)
}
Kernel kernel = new Kernel(){
   @Override public void run() {
      int gid = getGlobalId(); (2)
      squares[gid] = input[gid] * input[gid]; (3)
   }
};
kernel.execute(Range.create(512)); (4)
```

OpenCL Steps



Cluster Sample



Demo Time

Cluster Sample

```
@Override public void run() {
    doRun(getGlobalId());
}

public void doRun(int index) {
    float centerX = x[index];
    float centerY = y[index];
    float sumOfDistance = 0.0f;

    for (int idx = 1; idx < numberOfPoints; idx++) {
        float xx = x[idx] - centerX;
        float yy = y[idx] - centerY;
        sumOfDistance += (sqrt(((xx * xx) + (yy * yy))));
    }
    out[index] = (sumOfDistance * sumOfDistance) / numberOfPoints;
}
```

Kernel base structure

```
public class MultiplicationKernel extends Kernel {
    float[] inputMatrix;
    float outputMatrix [];
    public MultiplicationKernel(
      float inputMatrix[]) {
        ...
    }
    @Override
    public void run() {
        ...
    }
    public void displayResult() {
        ...
    }
  }
```

Implementation run()

```
public void run() {
    int globalID = this.getGlobalId();
    outputMatrix[globalID] = 2.0f * inputMatrix[globalID];
  }
```

Invoke the Kernel

```
float inputMatrix[] = {3, 4, 5, 6, 7, 8, 9};
  int size = inputMatrix.length;
  ...
  MultiplicationKernel kernel =
        new MultiplicationKernel(inputMatrix);
  kernel.execute(size);
  kernel.displayResult();
  kernel.dispose();
```

Sample Generated OpenCL Code 1

```
typedef struct This_s{
   __global float *val$sum;
   __global float *val$a;
   __global float *val$b;
   int passid;
}This;
int get_pass_id(This *this){
   return this->passid;
}
```

Sample Generated OpenCL Code 2

```
__kernel void run(
   __global float *val$sum,
   __global float *val$a,
   __global float *val$b,
   int passid
){
   This thisStruct;
   This* this=&thisStruct;
   this->val$sum = val$sum;
   this->val$a = val$a;
   this->val$b = val$b;
   this->passid = passid;
   {
      int gid = get_global_id(0);
      this->val$sum[gid]  = this->val$a[gid] + this->val$b[gid];
      return;
   }
}
```

Limitations

- No inheritance or method overloading
- Debugging is not easy
- In addition, it does not like println in the run method
- Supports one-dimensional arrays
  (Higher dimension arrays have to be flattened)
- The support for double values depends on the OpenCL version and GPU configuration

Aparapi Features

- Memory Management
- Plain OpenCL Interface
- Run Kernel on GPU or CPU or both

Explicit Buffer Handling

```
final int[] hugeArray = new int[HUGE];

final int[] done = new int[]{0};
Kernel kernel= new Kernel(){
   ... // reads/writes hugeArray and writes to done[0] when complete
};
kernel.setExplicit(true);
done[0]=0;
kernel.put(done);
kernel.put(hugeArray);
while (done[0] ==0)){
   kernel.execute(HUGE);
   kernel.get(done);
}
kernel.get(hugeArray);
```

Explicit Buffer Handling 2

```
// this is global accessable to all work items.
final int[] buffer = new int[1024];

// this is a constant buffer
final int[] buffer_$constant$ = new int[]{1,2,3,4,5,6,7,8,9}
@Constant int[] constantBuffer = new int[]{1,2,3,4,5,6,7,8,9}

        Kernel k = new Kernel(){
            public void run(){
                // access buffer
                // access buffer_$constant$
                // ....
            }
        }
```

Define a array as “local”

```
// This is global accessable to all work items.
final int[] buffer = new int[1024];
// A local buffer 1024 int's shared across all work item's in a group
final int[] buffer_$local$ = new int[1024];

Kernel k = new Kernel(){
   @Local int[] localBuffer = new int[1024];
   public void run(){
      // access buffer
      // access buffer_$local$
      localBarrier();
      // all writes to buffer_$local$ to be synchronized
      // across all work items in this group
      // ....
            }
         }
```

Aparapi Upcomming Features

- Multi-dimensional Arrays
- Lambda Expressions
- Use of simple objects as data structures
- Multible entry points

Using Java Lambdas

Experimental: Aparapi Lambda Brach:
http://aparapi.com/documentation/hsa-enabled-lambda.html

```
int in[] = ..//
int out[] = .../
Device.hsa().forEach(in.length, (i)->{
  out[i] = in[i]*in[i];
});
```

IBM Java SDK 8/9

Provides a feature to run Java Lambda-Expressions on the GPU:

```
public static void main(String[] args) {
    int[] toSort = { 5, 17, 12, 32, 2, 6, 3, 20 };
    try {
        Maths.sortArray(toSort);
    } catch (GPUSortException e) {
        e.printStackTrace();
    } catch (GPUConfigurationException e) {
        e.printStackTrace();
    }
}
```

Links

Samples and Slides: https://github.com/brainbrix/aparapi-test

Introduction GPU Programming:
http://courses.cms.caltech.edu/cs179/2017_lectures/cs179_2017_lec01.pdf

Aparapi:
https://github.com/Syncleus/aparapi
https://github.com/Syncleus/aparapi-examples

LWJGL: http://wiki.lwjgl.org/wiki/OpenCL_in_LWJGL.html

IBM Java SDK 9:
https://www.ibm.com/support/knowledgecenter/en/SSYKE2_9.0.0/com.ibm.java.multiplatform.90.doc/user/gpu_developing.html

Khronos Group: https://www.khronos.org/opencl/

Heiko Spindler
hs@heikospindler.de