Languages and Programming Environments

Dominik Göddeke

ARCS 2008 - Architecture of Computing Systems
GPGPU and CUDA Tutorials
Dresden, Germany, February 25 2008
Summary (last things first)

- GPGPU programming
  - Is traditionally seen, well, with some prejudice
  - "Hacking the GPU"

- This is a misconception
  - Admittedly (mostly) true until 2006

- My claim
  - not (much) harder to write efficient code for GPUs than for multicores
  - Heterogeneous memory hierarchies / NUMA already present in commodity CPUs

- It’s all about algorithm design for 100s of cores
  - And languages indicating how this can be exposed to us
Overview

- "Old School": Graphics APIs
- GPGPU languages, GPU computing, stream computing
  - CAL (AMD)
  - CUDA (NVIDIA)
  - RapidMind
  - Brook, Brook+
  - Accelerator
    - Make programming GPUs easier
    - Allow to focus on the algorithm and not on implementational details
    - Integrate the GPU as a computational resource into the rest of the system
"Old School" GPGPU

• Use graphics APIs to access GPU
  – DirectX, Direct3D (Windows, vendor-independent)
  – OpenGL (platform-independent, vendor-dependent via extensions)

• Use high level shading languages to implement computation kernels
  – GLSL (OpenGL)
  – HLSL (D3D)
  – Cg (NVIDIA, both GL and D3D)

• Toolchain support
  – D3D and GL: Libraries and headers, build around C++ and C (wrappers exist for many other languages)
  – Shading languages with separate compilers (embedded into the driver and standalone)
"Old School" GPGPU

- **Cast algorithms into graphics operations**
  - Arrays = Textures
  - Need to cope with unrelated things such as viewport transformation
  - Computing = Drawing

- **Advantages**
  - Platform- and vendor-independent
  - No license required

- **Disadvantages**
  - No direct access to the hardware
  - Steep learning curve
  - Graphics-centric
AMD Compute Abstraction Layer (CAL)

- http://ati.amd.com/technology/streamcomputing/resources.html

- **Bottom-up approach to "stream computing"**
  - Allow low-level access to the hardware for those who want it
  - Provide high-level implementations to those who don’t

- **Expose relevant parts of the GPU (R600+):**
  - Command processor
  - Data parallel processors
  - Memory controller

- **Hide everything else**
  - In particular, graphics-specific features and constraints
  - Take the driver out of the loop
  - Direct communication to device
AMD Compute Abstraction Layer (CAL)

- Design goals
  - Interact with the processing cores on the lowest level if needed
  - Maintain forward compatibility
  - Device-specific code generation
  - Device management
  - Multi-device support
  - Resource management
  - Kernel loading and execution (written in AMD IL – intermediate language, assembly-like)

- CAL SDK
  - Small set of C routines and data types, e.g. to download IL code into command processor and to trigger the computation
AMD Stream Computing Software Stack

- Libraries
  - AMD ACML (BLAS, LAPACK, FFT, RNG)
  - Includes loadbalancer (suitability of a task for a particular architecture)
  - AMD COBRA (video library)

- Compilers: Brook+ and RapidMind
  - Target both GPUs and multicore CPUs

- Compute Abstraction Layer (CAL)

- Hardware: FireStream GPUs, HAL

- Currently in beta testing
  - Check webpage for updates
CUDA

• http://www.nvidia.com/cuda

• See Simon‘s talks later today
RapidMind

• http://www.rapidmind.net

• **Software development platform for both multicore and stream processors**
  – Multicore CPUs, Cell BE and GPUs

• **Embedded within ISO C++**
  – No changes to toolchain, compilers etc.

• **Portable code**
  – But exposes platform-specific functionality to allow fine-tuning if needed

Slides based on talks by Mike Houston and Stefanus Du Toit
RapidMind

- Program definition

```plaintext
Program p;
p = BEGIN {
  In<Value3f> a, b;
  Out<Value3f> c;
  IF (all(a > 0.0f)) {
    Value3f d = f(a, b);
    c = d + a * 2.0f;
  } ELSE {
    c = d - a * 2.0f;
  } ENDIF;
} END;
```

Slides based on talks by Mike Houston and Stefanus Du Toit
RapidMind

• **SPMD data parallel programming model**
  – Data parallel arrays
  – Programs return new arrays
  – Programs may have control flow, may perform random reads from other arrays
  – Subarrays, ranges

• **Collective Operations**
  – Reduce, gather, scatter, ...

• **License**
  – sales@rapidmind.net
  – Very supportive to academia, company founded out of University of Waterloo, Canada

Slides based on talks by Mike Houston and Stefanus Du Toit
RapidMind

- **Example: Step 1 - Replace types**

```c
#include <cmath>

float f;
float a[512][512][3];
float b[512][512][3];

float func(
    float r, float s
) {
    return (r + s) * f;
}

void func_arrays() {
    for (int x = 0; x<512; x++) {
        for (int y = 0; y<512; y++) {
            for (int k = 0; k<3; k++) {
                a[y][x][k] =
                    func(a[y][x][k],b[y][x][k]);
            }
        }
    }
}
```

```cpp
#include <rapidmind/platform.hpp>

Value1f f;
Array<2,Value3f> a(512,512);
Array<2,Value3f> b(512,512);

Value3f func(
    Value3f r, Value3f s
) {
    return (r + s) * f;
}
```

Slides based on talks by Mike Houston and Stefanus Du Toit
RapidMind

- Example: Step 2 - Capture computation

```cpp
#include <cmath>

float f;
float a[512][512][3];
float b[512][512][3];

float func(
    float r, float s
) {
    return (r + s) * f;
}

void func_arrays() {
    for (int x = 0; x<512; x++) {
        for (int y = 0; y<512; y++) {
            for (int k = 0; k<3; k++) {
                a[y][x][k] =
                func(a[y][x][k], b[y][x][k]);
            }
        }
    }
}
```

```cpp
#include <rapidmind/platform.hpp>

Value1f f;
Array<2,Value3f> a(512,512);
Array<2,Value3f> b(512,512);

Value3f func(
    Value3f r, Value3f s
) {
    return (r + s) * f;
}

void func_arrays() {
    Program func_prog = BEGIN {
        In<Value3f> r, s;
        Out<Value3f> q;
        q = func(r,s);
    } END;
    ...
}
```
• Example: Step 3 - Parallel execution

```cpp
#include <cmath>

float f;
float a[512][512][3];
float b[512][512][3];

float func(
    float r, float s
) {
    return (r + s) * f;
}

void func_arrays() {
    for (int x = 0; x<512; x++)
        for (int y = 0; y<512; y++)
            for (int k = 0; k<3; k++)
                a[y][x][k] =
                    func(a[y][x][k], b[y][x][k]);
}
}
```

```cpp
#include <rapidmind/platform.hpp>

Value3f f;
Array<2,Value3f> a(512,512);
Array<2,Value3f> b(512,512);

Value3f func(
    Value3f r, Value3f s
) {
    return (r + s) * f;
}

void func_arrays() {
    Program func_prog = BEGIN {
        In<Value3f> r, s;
        Out<Value3f> q;
        q = func(r,s);
    } END;
    a = func_prog(a,b);
}
```
RapidMind

• **Usage:**
  – Include platform headers
  – Link to runtime library

• **Data**
  – Value tuples
  – Data parallel arrays
  – Remote data abstraction

• **Programs**
  – Defined dynamically
  – Execute on multicores and co-processors
  – Remote procedure abstraction

```cpp
#include <rapidmind/platform.hpp>

Value3f f;
Array<2, Value3f> a(512, 512);
Array<2, Value3f> b(512, 512);

Value3f func(
    Value3f r, Value3f s
) {
    return (r + s) * f;
}

void func_arrays() {
    Program func_prog = BEGIN {
        In<Value3f> r, s;
        Out<Value3f> q;
        q = func(r,s);
    } END;
    a = func_prog(a,b);
}
```

Slides based on talks by Mike Houston and Stefanus Du Toit
Brook, Brook+

- **Developed at Stanford University**
  - SIGGRAPH 2004 paper by Buck et al.

- **Brook: General purpose streaming language**
  - Compiler and runtime
  - C with stream extensions
  - Integrates seamlessly into C/C++ toolchains

- **Cross-platform**
  - Windows and Linux
  - Backends for OpenGL and DirectX, running on ATI and NVIDIA
Brook, Brook+

• Actively being developed
  – SVN tree *much* more up to date than downloadable tarballs
  – http://www.sourceforge.net/projects/brook

• Open Source
  – Compiler: GPL
  – Runtime: BSD

• AMD’s brook+
  – Added backend and compiler support for IL/CAL
  – Currently betatexting, will be released open source
• **Streams**
  – Collection of records requiring similar computation
  – Particle positions, FEM cells, voxels ...

  ```cpp
  float3 velocityfield<100,100,100>;
  ```

  – Similar to arrays
  – No index operations
  – Explicit "memcpy" via `streamRead()`, `streamWrite()` from standard C/C++ arrays

Slides courtesy of Mike Houston
• **Kernels**
  – Functions applied to streams
  – Similar to `for_all`
  – No dependencies between stream elements

```c
void foo (float* a, float* b, float* c, int N) {
    for (int i=0; i<N; i++)
        c[i] = a[i] + b[i]
}

int N=100;
float* a; float* b, float* c;
foo(a,b,c,N);
```

```c
kernel void foo (float a<>, float b<>,
                out float result<> ) {
    result = a + b;
}

float a<100>;
float b<100>;
float c<100>;
foo(a,b,c);
```

Slides courtesy of Mike Houston
• **Kernel arguments**
  
  – Input / output streams (different shape resolved by repeat and stride)

```c
kernel void foo (float a<>, float b<>, out float result) {
    result = a + b;
}
```
• **Kernel arguments**
  - Input / output streams (different shape resolved by repeat and stride)
  - Gather streams

```cpp
kernel void foo (float array[[],
                 out float result) {
    result = array[i];
}
```
• **Kernel arguments**
  – Input / output streams (different shape resolved by repeat and stride)
  – Gather streams
  – Iterator streams

```c
kernel void foo (float a<>,
    iter float n<>,
    out float result) {
    result = a+n;
}
```
Brook, Brook+

- **Kernel arguments**
  - Input / output streams (different shape resolved by repeat and stride)
  - Gather streams
  - Iterator streams
  - Constant parameters

```c
kernel void foo (float a<>,
    float c,
    out float result) {
    result = a+c;
}
```
• Reductions
  – Compute a single value from a stream
  – Associative operations only

```cpp
float a<100>;
float r;
reduce void sum (float a<>,
  reduce float r<> ) {
  r += a;
}
sum(a,r);
```
Accelerator

• **Microsoft Research**
  
  
  
  – Binaries available for noncommercial use

• **Data parallel array library**
  
  – including a just-in-time compiler that generates pixel shader code
  
  – runs on top of .NET, C#

• **Explicit conversion to data parallel arrays triggers computation**
  
  – Functional programming: Each operation creates a new data parallel array
Accelerator

• **Available operations**
  – Array creation, explicit conversions
  – Element-wise arithmetic and boolean operations
  – Reductions: max, min, sum, product
  – Transformations: expand, pad, shift, gather, scatter
  – Basic linear algebra

• **Unsupported operations:**
  – no aliasing, pointer arithmetic, access to individual elements
**Example: 2D convolution**

```csharp
using Microsoft.Research.DataParallelArrays;

static float[,] Blur(float[,] array, float[] kernel)
{
    float[,] result;
    DFPA parallelArray = new DFPA(array);

    FPA resultX = new FPA(0f, parallelArray.Shape);
    for (int i = 0; i < kernel.Length; i++) {
        int[] shiftDir = new int[] { 0, i};
        resultX += PA.Shift(parallelArray, shiftDir) * kernel[i];
    }

    FPA resultY = new FPA(0f, parallelArray.Shape);
    for (int i = 0; i < kernel.Length; i++) {
        int[] shiftDir = new int[] { i, 0};
        resultY += PA.Shift(resultX, shiftDir) * kernel[i];
    }

    PA.ToArray(resultY, out result);
    parallelArray.Dispose();
    return result;
}
```

Taken from Tarditi et al.: Accelerator: Using data parallelism to program GPUs for general purpose uses
Accelerator

- Example: 2D convolution

```csharp
using Microsoft.Research.DataParallelArrays;

static float[,] Blur(float[,] array, float[] kernel)
{
    float[,] result;
    DFPA parallelArray = new DFPA(array);

    FPA resultX = new FPA(0f, parallelArray.Shape);
    for (int i = 0; i < kernel.Length; i++) {
        int[] shiftDir = new int[] { 0, i};
        resultX += PA.Shift(parallelArray, shiftDir) * kernel[i];
    }

    FPA resultY = new FPA(0f, parallelArray.Shape);
    for (int i = 0; i < kernel.Length; i++) {
        int[] shiftDir = new int[] { i, 0};
        resultY += PA.Shift(resultX, shiftDir) * kernel[i];
    }

    PA.ToArray(resultY, out result);
    parallelArray.Dispose();
    return result;
}
```

Taken from Tarditi et al.: Accelerator: Using data parallelism to program GPUs for general purpose uses
Accelerator

- **Example: 2D convolution**

```csharp
using Microsoft.Research.DataParallelArrays;

static float[,] Blur(float[,] array, float[] kernel)
{
    float[,] result;
    DFPA parallelArray = new DFPA(array);

    FPA resultX = new FPA(0f, parallelArray.Shape);
    for (int i = 0; i < kernel.Length; i++) {
        int[] shiftDir = new int[] { 0, i};
        resultX += PA.Shift(parallelArray, shiftDir) * kernel[i];
    }

    FPA resultY = new FPA(0f, parallelArray.Shape);
    for (int i = 0; i < kernel.Length; i++) {
        int[] shiftDir = new int[] { i, 0};
        resultY += PA.Shift(resultX, shiftDir) * kernel[i];
    }

    PA.ToArray(resultY, out result);
    parallelArray.Dispose();
    return result;
}
```

Compute blur by shifting the entire original image by i pixels and multiplying with the appropriate weight.

Taken from Tarditi et al.: Accelerator: Using data parallelism to program GPUs for general purpose uses
Accelerator

• Example: 2D convolution

```csharp
using Microsoft.Research.DataParallelArrays;

static float[,] Blur(float[,] array, float[] kernel)
{
    float[,] result;
    DFPA parallelArray = new DFPA(array);

    FPA resultX = new FPA(0f, parallelArray.Shape);
    for (int i = 0; i < kernel.Length; i++) {
        int[] shiftDir = new int[] { 0, i);
        resultX += PA.Shift(parallelArray, shiftDir) * kernel[i];
    }

    FPA resultY = new FPA(0f, parallelArray.Shape);
    for (int i = 0; i < kernel.Length; i++) {
        int[] shiftDir = new int[] { i, 0);
        resultY += PA.Shift(resultX, shiftDir) * kernel[i];
    }

    PA.ToArray(resultY, out result);
    parallelArray.Dispose();
    return result;
}
```

Taken from Tarditi et al.: Accelerator: Using data parallelism to program GPUs for general purpose uses
• **Example: 2D convolution**

```csharp
using Microsoft.Research.DataParallelArrays;

static float[,] Blur(float[,] array, float[] kernel)
{
    float[,] result;
    DFPA parallelArray = new DFPA(array);

    FPA resultX = new FPA(0f, parallelArray.Shape);
    for (int i = 0; i < kernel.Length; i++) {
        int[] shiftDir = new int[] { 0, i };
        resultX += PA.Shift(parallelArray, shiftDir) * kernel[i];
    }

    FPA resultY = new FPA(0f, parallelArray.Shape);
    for (int i = 0; i < kernel.Length; i++) {
        int[] shiftDir = new int[] { i, 0 };
        resultY += PA.Shift(resultX, shiftDir) * kernel[i];
    }

    PA.ToArray(resultY, out result);
    parallelArray.Dispose();
    return result;
}
```

Taken from Tarditi et al.: Accelerator: Using data parallelism to program GPUs for general purpose uses
Acknowledgements

• **Mike Houston, Ian Buck**
  – inspired by previous talks on the topic

• **Stefanus Du Toit**
  – RapidMind examples