How to program a GPU?
Important features from a software point of view

- Massively parallel
  - Only useful for "inner loop" style code
- High-bandwidth, high-latency memory
  - Favors data streaming rather than random access
- Separate processor
  - Not autonomous
  - Managed by host CPU

Programming paradigms

- Kernels and stream programming
- Structured programming, flow control
- Shared memory and host communication
- JIT compilation, implicit or explicit
- Single or multi-level languages
- Library, language extension, or annotations

Kernels

- Small function
- Called multiple times implicitly
  - How many times and with which arguments depends on host program
- (Mostly) independent from other kernel calls
- Data parallelism

Structured programming

- C syntax, for loops, conditionals, functions, etc.
- SIMD flow control
  - Guarded execution
  - Jump if all threads in a cluster follow the same path
Structured programming

GLSL (/ HLSL / Cg)

- Compiled by command
- Fast switching between compiled kernels
- Loading and “calling” as in shader assembly

```
uniform vec4 insideColor;
uniform sampler1D outsideColorTable;
uniform float maxIterations;

void main ()
{
  vec2 c = gl_TexCoord[0].xy;
  vec2 z = c;
  gl_FragColor = insideColor;
  for (float i = 0; i < maxIterations; i += 1.0)
  {
    z = vec2(z.x*z.x - z.y*z.y, 2.0*z.x*z.y) + c;
    if (dot(z, z) > 4.0){
      gl_FragColor = texture1D(outsideColorTable, i / maxIterations);
      break;
    }
  }
}
```

Shared memory

OpenCL (/ DirectX compute shaders)

- Shared data within a threadblock
- Explicit synchronization
- Race conditions
- Thread-driven computation
- Number of threads determined by programmer
- Explicit looping within threads

```
__local float4 *shared_pos

...intindex = get_global_id(0);
int local_id = get_local_id(0);
tile_size = get_local_size(0);
...inti, j;
for (i = 0; i < bodies; i += tile_size, tile++)
{
  size_t l_idx = (tile * tile_size + local_id);
  float4l_pos = i_pos[l_idx];
  shared_pos[local_id] = l_pos;
  barrier(CLK_LOCAL_MEM_FENCE);
  for (j = 0; j < tile_size; )
  {
    force = ComputeForce(force, shared_pos[j++], pos, softening_squared);
  }
  barrier(CLK_LOCAL_MEM_FENCE);
}
```

Implicit compilation

RapidMind (Sh)

- Macros for unoverloadable operations
- Implicit communication
- Read & write instead of transfer
- Asynchronous execution

```
Array<2,Value1f> A(m,l);
Array<2,Value1f> B(l,n);Array<2,Value1f> C(m,n);
Program mxm = BEGIN {
  In<Value2i> ind;
  Out<Value1f> c = Value1f(0.);
  Value1i k;// Computation of C(i,j)
  RM_FOR (k=0, k < Value1i(l), k++) {
    c += A[Value2i(ind(0),k)]*B[Value2i(k,ind(1))];
  } RM_ENDFOR;
} END;
C = mxm(grid(m,n));
```

Single-level language

CUDA

- Kernel is just a function
- No variables holding code
- Extension to C/C++
- Requires dedicated compiler

```
__global__ void
paradd(float *in, float *out, int size){
  const int stride = blockDim.x * gridDim.x;
  const int start  = IMUL(blockDim.x, blockIdx.x) + threadIdx.x;
  __shared__ float accum[THREADS];
  accum[threadIdx.x] = 0;
  for (int ii=start; ii < size; ii += stride)
  {
    accum[threadIdx.x] += in[ii];
  }
  __syncthreads();if(!threadIdx.x)
  {
    float res = 0;
    for (int ii = 0; ii < blockDim.x; ii++)
    {
      res += accum[ii];
    }
    out[blockIdx.x] = res;
  }
}
```

Stream programming

- Notion of data shape
- Restricts access pattern
- Can be extended to different access patterns
- Recursive neighborhood, stack, etc.
- Dependent on hardware
Stream programming
Brook(GPU)

- Gather streams for random access

Annotation
PGI Accelerator (/ CAPS HMPP)

- Inspired by HPF & OpenMP
- Just add pragmas
- Can still compile under other compilers
- Incremental upgrade path
- Compiler is not all-knowing
- Directives may need to be specific
- Manually restructure loops

Accelerator library
jacket

- All GPU code is encapsulated in library calls
- GPU memory management
- Data conversion = transfer
- Matlab toolbox
- JIT removes overhead
- Similar to RapidMind retained mode
- Data type determines CPU or GPU execution

Summary

<table>
<thead>
<tr>
<th>Struc-</th>
<th>Kernels</th>
<th>Lvls</th>
<th>Platform</th>
<th>Compilation</th>
<th>Kernel</th>
<th>Comms</th>
<th>Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD</td>
<td>✔️</td>
<td>2</td>
<td>Library</td>
<td>Explicit</td>
<td>X</td>
<td>Explicit</td>
<td></td>
</tr>
<tr>
<td>CUBA</td>
<td>✔️</td>
<td>1</td>
<td>Compiler</td>
<td>Implicit</td>
<td>X</td>
<td>Implicit</td>
<td></td>
</tr>
<tr>
<td>Jacket</td>
<td>✔️</td>
<td>1</td>
<td>Toolbox</td>
<td>Implicit</td>
<td>X</td>
<td>Implicit</td>
<td></td>
</tr>
</tbody>
</table>

Conclusion

- There are many GPU programming languages
- Some use radically different programming paradigms
- Often trading efficiency for ease of use
- Paradigm shift often restricted to GPU kernels
- But future multi-GPU and task parallel code may change that
- Programmer effort will always be required
- Cannot simply rely on compiler
- Look around before you choose a language

Questions?

- Gather streams for random access

- Inspired by HPF & OpenMP
- Just add pragmas
- Can still compile under other compilers
- Incremental upgrade path
- Compiler is not all-knowing
- Directives may need to be specific
- Manually restructure loops
Example sources

- Vendors