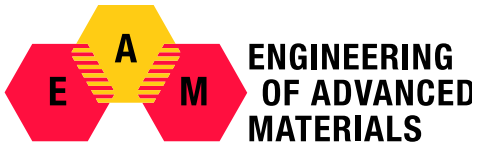
Supercomputing on Graphics Cards

Marcus Bannerman, Severin Strobl, Thorsten Pöschel

mss-opencl@cbi.uni-erlangen.de

An Introduction to OpenCL and the C++ Bindings: Part 9



Part 1: Compiler Options and Kernel "Binaries"

1. ICD
   - What is the ICD extension?
   - Why use it?
   - How to use it?
2. Compiler Options
   - Standard Options
   - NVidia's Options
   - ATI's Options
3. Saving Kernel "Binaries"
   - About Intermediate Formats
   - NVidia
   - AMD
4. A Walk Through A PTX Kernel With Severin
   - Introduction to PTX
   - Syntax and Instruction Set of PTX
   - Code Sample

Part 2: Debugging OpenCL

5 Printing To The Screen
- Availability
- Usage
- Combining With Defines

6 GDB and OpenCL
- Prerequisites
- Running GDB

Part I

Compiler Options and Kernel "Binaries"

Outline

1. ICD
   - What is the ICD extension?
   - Why use it?
   - How to use it?
2. Compiler Options
   - Standard Options
   - NVidia's Options
   - ATI's Options
3. Saving Kernel "Binaries"
   - About Intermediate Formats
   - NVidia
   - AMD
4. A Walk Through A PTX Kernel With Severin
   - Introduction to PTX
   - Syntax and Instruction Set of PTX
   - Code Sample

- There is an extension, now supported by AMD and NVidia, denoted by the name `cl_khr_icd`.
- This extension makes it possible to use multiple OpenCL platforms on one computer.
- It's important to note that you cannot "link" the platforms (buffers, kernels etc. live in `contexts`, and these cannot be moved across `platforms`).
- But you can run your program on any of the platforms available.
- For example today's example code, run on my desktop PC, gives...

- There is an extension, now supported by AMD and NVidia, denoted by the name `cl_khr_icd`.
- This extension makes it possible to use multiple OpenCL platforms on one computer.
- It's important to note that you cannot "link" the platforms (buffers, kernels etc. live in `contexts`, and these cannot be moved across `platforms`).
- But you can run your program on any of the platforms available.
- For example today's example code, run on my desktop PC, gives...

- There is an extension, now supported by AMD and NVidia, denoted by the name `cl_khr_icd`.
- This extension makes it possible to use multiple OpenCL platforms on one computer.
- It's important to note that you cannot "link" the platforms (buffers, kernels etc. live in `contexts`, and these cannot be moved across `platforms`).
- But you can run your program on any of the platforms available.
- For example today's example code, run on my desktop PC, gives...

- There is an extension, now supported by AMD and NVidia, denoted by the name `cl_khr_icd`.
- This extension makes it possible to use multiple OpenCL platforms on one computer.
- It's important to note that you cannot "link" the platforms (buffers, kernels etc. live in `contexts`, and these cannot be moved across `platforms`).
- But you can run your program on any of the platforms available.
- For example today's example code, run on my desktop PC, gives...

- There is an extension, now supported by AMD and NVidia, denoted by the name `cl_khr_icd`.
- This extension makes it possible to use multiple OpenCL platforms on one computer.
- It's important to note that you cannot "link" the platforms (buffers, kernels etc. live in `contexts`, and these cannot be moved across `platforms`).
- But you can run your program on any of the platforms available.
- For example today's example code, run on my desktop PC, gives...

```
#####################################################
OpenCL platform [0]: ATI Stream
 Extensions:
##OpenCL device [0]: Intel(R) Core(TM) i5 CPU         750  @ 2.67GHz
    Type: CPU
    Extensions: cl_khr_icd
                cl_amd_fp64
                cl_khr_global_int32_base_atomics
                cl_khr_global_int32_extended_atomics
                cl_khr_local_int32_base_atomics
                cl_khr_local_int32_extended_atomics
                cl_khr_int64_base_atomics
                cl_khr_int64_extended_atomics
                cl_khr_byte_addressable_store
                cl_khr_gl_sharing
                cl_ext_device_fission
                cl_amd_device_attribute_query
                cl_amd_printf
##OpenCL device [1]: Cypress
    Type: GPU
    Extensions: cl_khr_icd
                cl_amd_fp64
                cl_khr_global_int32_base_atomics
                cl_khr_global_int32_extended_atomics
                cl_khr_local_int32_base_atomics
                cl_khr_local_int32_extended_atomics
                cl_khr_byte_addressable_store
                cl_khr_gl_sharing
                cl_amd_device_attribute_query
                cl_amd_media_ops
```

```
#####################################################
OpenCL platform [1]: NVIDIA CUDA
 Extensions: cl_khr_byte_addressable_store
             cl_khr_icd
             cl_khr_gl_sharing
             cl_nv_compiler_options
             cl_nv_device_attribute_query
             cl_nv_pragma_unroll
##OpenCL device [0]: GeForce GTX 260
    Type: GPU
    Extensions: cl_khr_byte_addressable_store
                cl_khr_icd
                cl_khr_gl_sharing
                cl_nv_compiler_options
                cl_nv_device_attribute_query
                cl_nv_pragma_unroll
                cl_khr_global_int32_base_atomics
                cl_khr_global_int32_extended_atomics
                cl_khr_local_int32_base_atomics
                cl_khr_local_int32_extended_atomics
                cl_khr_fp64
```

- **Advantage:** Using `cl_khr_icd` you can use your CPU (AMD) and your NVidia GPU, even though NVidia doesn't support the CPU in OpenCL.
- **Advantage:** You can test your code on multiple implementations, more testing means more bugs squished!
- **Advantage:** If we run on AMD's **CPU** implementation, we get access to `cl_amd_printf` and gdb debugging of our kernels (more in a bit)!
- **Disadvantage:** None! Well, each platform has it's own annoying peculiarities/errors and now you can encounter them all.

- **Advantage:** Using `cl_khr_icd` you can use your CPU (AMD) and your NVidia GPU, even though NVidia doesn't support the CPU in OpenCL.
- **Advantage:** You can test your code on multiple implementations, more testing means more bugs squished!
- **Advantage:** If we run on AMD's **CPU** implementation, we get access to `cl_amd_printf` and gdb debugging of our kernels (more in a bit)!
- **Disadvantage:** None! Well, each platform has it's own annoying peculiarities/errors and now you can encounter them all.

- **Advantage:** Using `cl_khr_icd` you can use your CPU (AMD) and your NVidia GPU, even though NVidia doesn't support the CPU in OpenCL.
- **Advantage:** You can test your code on multiple implementations, more testing means more bugs squished!
- **Advantage:** If we run on AMD's **CPU** implementation, we get access to `cl_amd_printf` and gdb debugging of our kernels (more in a bit)!
- **Disadvantage:** None! Well, each platform has it's own annoying peculiarities/errors and now you can encounter them all.

- **Advantage:** Using `cl_khr_icd` you can use your CPU (AMD) and your NVidia GPU, even though NVidia doesn't support the CPU in OpenCL.
- **Advantage:** You can test your code on multiple implementations, more testing means more bugs squished!
- **Advantage:** If we run on AMD's **CPU** implementation, we get access to `cl_amd_printf` and gdb debugging of our kernels (more in a bit)!
- **Disadvantage:** None! Well, each platform has it's own annoying peculiarities/errors and now you can encounter them all.

- To use `cl_khr_icd`, you first have to set up one OpenCL implementation as normal.
- Then you have to install the other implementation in a separate directory.
- Finally, you have to create and fill the `/etc/OpenCL/vendors/` directory with a file for each implementation (e.g. nvidia.icd, ati32.icd, ati64.icd).
- Inside this file you write the path to the OpenCL library for that implementation (e.g., nvidia.icd=libcuda.so, ati32.icd=libatiocl32.so, ati64.icd=libatiocl64.so).
- Any installed OpenCL implementation that supports `cl_khr_icd` will check this directory and find the other implementations.
- Now we have access to two platforms, lets have a look at some of the extensions!

- To use `cl_khr_icd`, you first have to set up one OpenCL implementation as normal.
- Then you have to install the other implementation in a separate directory.
- Finally, you have to create and fill the `/etc/OpenCL/vendors/` directory with a file for each implementation (e.g. nvidia.icd, ati32.icd, ati64.icd).
- Inside this file you write the path to the OpenCL library for that implementation (e.g., nvidia.icd=libcuda.so, ati32.icd=libatiocl32.so, ati64.icd=libatiocl64.so).
- Any installed OpenCL implementation that supports `cl_khr_icd` will check this directory and find the other implementations.
- Now we have access to two platforms, lets have a look at some of the extensions!

- To use `cl_khr_icd`, you first have to set up one OpenCL implementation as normal.
- Then you have to install the other implementation in a separate directory.
- Finally, you have to create and fill the `/etc/OpenCL/vendors/` directory with a file for each implementation (e.g. nvidia.icd, ati32.icd, ati64.icd).
- Inside this file you write the path to the OpenCL library for that implementation (e.g., nvidia.icd=libcuda.so, ati32.icd=libatiocl32.so, ati64.icd=libatiocl64.so).
- Any installed OpenCL implementation that supports `cl_khr_icd` will check this directory and find the other implementations.
- Now we have access to two platforms, lets have a look at some of the extensions!

- To use `cl_khr_icd`, you first have to set up one OpenCL implementation as normal.
- Then you have to install the other implementation in a separate directory.
- Finally, you have to create and fill the `/etc/OpenCL/vendors/` directory with a file for each implementation (e.g. nvidia.icd, ati32.icd, ati64.icd).
- Inside this file you write the path to the OpenCL library for that implementation (e.g., nvidia.icd=libcuda.so, ati32.icd=libatiocl32.so, ati64.icd=libatiocl64.so).
- Any installed OpenCL implementation that supports `cl_khr_icd` will check this directory and find the other implementations.
- Now we have access to two platforms, lets have a look at some of the extensions!

- To use `cl_khr_icd`, you first have to set up one OpenCL implementation as normal.
- Then you have to install the other implementation in a separate directory.
- Finally, you have to create and fill the `/etc/OpenCL/vendors/` directory with a file for each implementation (e.g. nvidia.icd, ati32.icd, ati64.icd).
- Inside this file you write the path to the OpenCL library for that implementation (e.g., nvidia.icd=libcuda.so, ati32.icd=libatiocl32.so, ati64.icd=libatiocl64.so).
- Any installed OpenCL implementation that supports `cl_khr_icd` will check this directory and find the other implementations.
- Now we have access to two platforms, lets have a look at some of the extensions!

- To use `cl_khr_icd`, you first have to set up one OpenCL implementation as normal.
- Then you have to install the other implementation in a separate directory.
- Finally, you have to create and fill the `/etc/OpenCL/vendors/` directory with a file for each implementation (e.g. nvidia.icd, ati32.icd, ati64.icd).
- Inside this file you write the path to the OpenCL library for that implementation (e.g., nvidia.icd=libcuda.so, ati32.icd=libatiocl32.so, ati64.icd=libatiocl64.so).
- Any installed OpenCL implementation that supports `cl_khr_icd` will check this directory and find the other implementations.
- Now we have access to two platforms, lets have a look at some of the extensions!

Outline

1. ICD
   - What is the ICD extension?
   - Why use it?
   - How to use it?
2. Compiler Options
   - Standard Options
   - NVidia's Options
   - ATI's Options
3. Saving Kernel "Binaries"
   - About Intermediate Formats
   - NVidia
   - AMD
4. A Walk Through A PTX Kernel With Severin
   - Introduction to PTX
   - Syntax and Instruction Set of PTX
   - Code Sample

- When we compile our kernel sources into a program binary we can specify build options to be passed to the compiler...

```
std::string buildOptions;
...
try {
  program.build(devices, buildOptions.c_str());
} catch(cl::Error& err) {
  ...
}
```

- But what can we actually pass to the compiler? What are it's arguments?
- The OpenCL standard requires some options to always be available...

- When we compile our kernel sources into a program binary we can specify build options to be passed to the compiler...

```
std::string buildOptions;
...
try {
  program.build(devices, buildOptions.c_str());
} catch(cl::Error& err) {
  ...
}
```

- But what can we actually pass to the compiler? What are it's arguments?
- The OpenCL standard requires some options to always be available...

- When we compile our kernel sources into a program binary we can specify build options to be passed to the compiler...

```
std::string buildOptions;
...
try {
  program.build(devices, buildOptions.c_str());
} catch(cl::Error& err) {
  ...
}
```

- But what can we actually pass to the compiler? What are it's arguments?
- The OpenCL standard requires some options to always be available...

- First, we can define preprocessor macros.

```
std::string buildOptions;
buildOptions += " -DCrazyOptimizations";
buildOptions += " -DMyConstantValue=3.0";
```

- These options create preprocessor defines...

```
//The following lines are equivalent to the build options above
#define CrazyOptimizations
#define MyConstantValue=3.0

__kernel void myKernel() {
  ...
#ifndef CrazyOptimizations
  A = invsqrt(B);
#else
  union {
          float f;
          int i;
} tmp;
  tmp.f = x;
  tmp.i = 0x5f3759df - (tmp.i >> 1);
  float y = tmp.f;

  A = y * (1.5f - 0.5f * x * y * y)
#endif

 A+= MyConstantValue;
}
```

- First, we can define preprocessor macros.

```
std::string buildOptions;
buildOptions += " -DCrazyOptimizations";
buildOptions += " -DMyConstantValue=3.0";
```

- These options create preprocessor defines...

```
//The following lines are equivalent to the build options above
#define CrazyOptimizations
#define MyConstantValue=3.0

__kernel void myKernel() {
  ...
#ifndef CrazyOptimizations
  A = invsqrt(B);
#else
  union {
          float f;
          int i;
} tmp;
  tmp.f = x;
  tmp.i = 0x5f3759df - (tmp.i >> 1);
  float y = tmp.f;

  A = y * (1.5f - 0.5f * x * y * y)
#endif

 A+= MyConstantValue;
}
```

- We can also suppress all warnings

```
std::string buildOptions;
buildOptions += " -w";
```

- Or turn them all into errors

```
std::string buildOptions;
buildOptions += " -Werror";
```

- But some of the more cool/dangerous options are listed in section 5.4.3.2/3 of the OpenCL standard...

- We can also suppress all warnings

```
std::string buildOptions;
buildOptions += " -w";
```

- Or turn them all into errors

```
std::string buildOptions;
buildOptions += " -Werror";
```

- But some of the more cool/dangerous options are listed in section 5.4.3.2/3 of the OpenCL standard...

- We can also suppress all warnings

```
std::string buildOptions;
buildOptions += " -w";
```

- Or turn them all into errors

```
std::string buildOptions;
buildOptions += " -Werror";
```

- But some of the more cool/dangerous options are listed in section 5.4.3.2/3 of the OpenCL standard...

- We can treat all constants as single precision values (probably safe and sensible for all code where people don't define "difficult" constants like π or e, and always safe for single precision).

```
std::string buildOptions;
buildOptions += " -cl-single-precision-constant";
```

- A bit more dodgy, we can have denormalized numbers flush to zero (numbers where the mantissa is less than one). This is a suggestion that can help the FPU's out, but beware! $x - y == 0$ no longer implies $x == y$!

```
std::string buildOptions;
buildOptions += " -cl-denorms-are-zero";
```

- We can even turn on fused MAD (multiply-add) instructions

```
std::string buildOptions;
buildOptions += " -cl-mad-enable";
```

which replace $a * b + c$ with a single, possibly-lower-precision instruction (NVidia's 64bit unit is completely MAD, it can only do fused multiply-adds).

- We can treat all constants as single precision values (probably safe and sensible for all code where people don't define "difficult" constants like π or e, and always safe for single precision).

```
std::string buildOptions;
buildOptions += " -cl-single-precision-constant";
```

- A bit more dodgy, we can have denormalized numbers flush to zero (numbers where the mantissa is less than one). This is a suggestion that can help the FPU's out, but beware! $x - y == 0$ no longer implies $x == y$!

```
std::string buildOptions;
buildOptions += " -cl-denorms-are-zero";
```

- We can even turn on fused MAD (multiply-add) instructions

```
std::string buildOptions;
buildOptions += " -cl-mad-enable";
```

which replace $a * b + c$ with a single, possibly-lower-precision instruction (NVidia's 64bit unit is completely MAD, it can only do fused multiply-adds).

- We can treat all constants as single precision values (probably safe and sensible for all code where people don't define "difficult" constants like π or e, and always safe for single precision).

```
std::string buildOptions;
buildOptions += " -cl-single-precision-constant";
```

- A bit more dodgy, we can have denormalized numbers flush to zero (numbers where the mantissa is less than one). This is a suggestion that can help the FPU's out, but beware! $x - y == 0$ no longer implies $x == y$!

```
std::string buildOptions;
buildOptions += " -cl-denorms-are-zero";
```

- We can even turn on fused MAD (multiply-add) instructions

```
std::string buildOptions;
buildOptions += " -cl-mad-enable";
```

which replace $a * b + c$ with a single, possibly-lower-precision instruction (NVidia's 64bit unit is completely MAD, it can only do fused multiply-adds).

- We can also disable all optimizations (always useful when debugging).

```
std::string buildOptions;
buildOptions += " -cl-opt-disable";
```

- But this doesn't always work (AMD).
- From here on the options get more dodgy, we have `-cl-unsafe-math-optimizations` that trade accuracy for speed and should only be used if you know what you're doing.
- You can read about the rest in Section 5.4.3.3 if you really want to.
- But each vendor has some interesting options too, let's take a look at NVidia's first...

- We can also disable all optimizations (always useful when debugging).

```
std::string buildOptions;
buildOptions += " -cl-opt-disable";
```

- But this doesn't always work (AMD).
- From here on the options get more dodgy, we have `-cl-unsafe-math-optimizations` that trade accuracy for speed and should only be used if you know what you're doing.
- You can read about the rest in Section 5.4.3.3 if you really want to.
- But each vendor has some interesting options too, let's take a look at NVidia's first...

- We can also disable all optimizations (always useful when debugging).

```
std::string buildOptions;
buildOptions += " -cl-opt-disable";
```

- But this doesn't always work (AMD).
- From here on the options get more dodgy, we have `-cl-unsafe-math-optimizations` that trade accuracy for speed and should only be used if you know what you're doing.
- You can read about the rest in Section 5.4.3.3 if you really want to.
- But each vendor has some interesting options too, let's take a look at NVidia's first...

- We can also disable all optimizations (always useful when debugging).

```
std::string buildOptions;
buildOptions += " -cl-opt-disable";
```

- But this doesn't always work (AMD).
- From here on the options get more dodgy, we have `-cl-unsafe-math-optimizations` that trade accuracy for speed and should only be used if you know what you're doing.
- You can read about the rest in Section 5.4.3.3 if you really want to.
- But each vendor has some interesting options too, let's take a look at NVidia's first...

- We can also disable all optimizations (always useful when debugging).

```
std::string buildOptions;
buildOptions += " -cl-opt-disable";
```

- But this doesn't always work (AMD).
- From here on the options get more dodgy, we have `-cl-unsafe-math-optimizations` that trade accuracy for speed and should only be used if you know what you're doing.
- You can read about the rest in Section 5.4.3.3 if you really want to.
- But each vendor has some interesting options too, let's take a look at NVidia's first...

- NVidia has several options that can be used, and even define an OpenCL extension for them (`cl_nv_compiler_options`).
- There are some really cool options. First, we can get verbose information about the kernels memory usage...

```
std::string buildOptions;
buildOptions += " -cl-nv-verbose";
```

- This prints out statistics on the private memory usage (registers), local memory (smem), and constant memory (cmem).

```
ptxas info: Compiling entry function 'reductionVector' for 'sm_13'
ptxas info: Used 12 registers, 16+16 bytes smem, 236 bytes cmem[0],
 24 bytes cmem[1]
ptxas info: Compiling entry function 'reductionNVIDIA' for 'sm_13'
ptxas info: Used 6 registers, 16+16 bytes smem, 236 bytes cmem[0],
 12 bytes cmem[1]
ptxas info: Compiling entry function 'reduction' for 'sm_13'
ptxas info: Used 7 registers, 16+16 bytes smem, 236 bytes cmem[0],
 28 bytes cmem[1]
```

- NVidia has several options that can be used, and even define an OpenCL extension for them (`cl_nv_compiler_options`).
- There are some really cool options. First, we can get verbose information about the kernels memory usage...

```
std::string buildOptions;
buildOptions += " -cl-nv-verbose";
```

- This prints out statistics on the private memory usage (registers), local memory (smem), and constant memory (cmem).

```
ptxas info: Compiling entry function 'reductionVector' for 'sm_13'
ptxas info: Used 12 registers, 16+16 bytes smem, 236 bytes cmem[0],
 24 bytes cmem[1]
ptxas info: Compiling entry function 'reductionNVIDIA' for 'sm_13'
ptxas info: Used 6 registers, 16+16 bytes smem, 236 bytes cmem[0],
 12 bytes cmem[1]
ptxas info: Compiling entry function 'reduction' for 'sm_13'
ptxas info: Used 7 registers, 16+16 bytes smem, 236 bytes cmem[0],
 28 bytes cmem[1]
```

- NVidia has several options that can be used, and even define an OpenCL extension for them (`cl_nv_compiler_options`).
- There are some really cool options. First, we can get verbose information about the kernels memory usage...

```
std::string buildOptions;
buildOptions += " -cl-nv-verbose";
```

- This prints out statistics on the private memory usage (registers), local memory (smem), and constant memory (cmem).

```
ptxas info: Compiling entry function 'reductionVector' for 'sm_13'
ptxas info: Used 12 registers, 16+16 bytes smem, 236 bytes cmem[0],
 24 bytes cmem[1]
ptxas info: Compiling entry function 'reductionNVIDIA' for 'sm_13'
ptxas info: Used 6 registers, 16+16 bytes smem, 236 bytes cmem[0],
 12 bytes cmem[1]
ptxas info: Compiling entry function 'reduction' for 'sm_13'
ptxas info: Used 7 registers, 16+16 bytes smem, 236 bytes cmem[0],
 28 bytes cmem[1]
```

- Another cool option is that you can set an upper limit on the number of registers the compiler is allowed to use per kernel

```
std::string buildOptions;
buildOptions += " -cl-nv-maxrregcount=8";
```

- **Note:** There are two r's in the name, it's not a typo.
- This either reduces the number registers by increasing the number of loads from memory (less unchanged values are cached)...

```
#Before (-cl-nv-maxrregcount unset or > 12)
ptxas info: Compiling entry function 'reductionVector' for 'sm_13'
ptxas info: Used 12 registers, 16+16 bytes smem, 236 bytes cmem[0],
 24 bytes cmem[1]

#After (-cl-nv-maxrregcount=8)
ptxas info: Compiling entry function 'reductionVector' for 'sm_13'
ptxas info: Used 10 registers, 16+16 bytes smem, 236 bytes cmem[0],
24 bytes cmem[1]
```

- Another cool option is that you can set an upper limit on the number of registers the compiler is allowed to use per kernel

```
std::string buildOptions;
buildOptions += " -cl-nv-maxrregcount=8";
```

- **Note:** There are two r's in the name, it's not a typo.
- This either reduces the number registers by increasing the number of loads from memory (less unchanged values are cached)...

```
#Before (-cl-nv-maxrregcount unset or > 12)
ptxas info: Compiling entry function 'reductionVector' for 'sm_13'
ptxas info: Used 12 registers, 16+16 bytes smem, 236 bytes cmem[0],
 24 bytes cmem[1]

#After (-cl-nv-maxrregcount=8)
ptxas info: Compiling entry function 'reductionVector' for 'sm_13'
ptxas info: Used 10 registers, 16+16 bytes smem, 236 bytes cmem[0],
24 bytes cmem[1]
```

- Another cool option is that you can set an upper limit on the number of registers the compiler is allowed to use per kernel

```
std::string buildOptions;
buildOptions += " -cl-nv-maxrregcount=8";
```

- **Note:** There are two r's in the name, it's not a typo.
- This either reduces the number registers by increasing the number of loads from memory (less unchanged values are cached)...

```
#Before (-cl-nv-maxrregcount unset or > 12)
ptxas info: Compiling entry function 'reductionVector' for 'sm_13'
ptxas info: Used 12 registers, 16+16 bytes smem, 236 bytes cmem[0],
 24 bytes cmem[1]

#After (-cl-nv-maxrregcount=8)
ptxas info: Compiling entry function 'reductionVector' for 'sm_13'
ptxas info: Used 10 registers, 16+16 bytes smem, 236 bytes cmem[0],
24 bytes cmem[1]
```

- Or it "spills" **required** registers into lmem or "CUDA local" memory (not OpenCL *local* memory, it's slower like global memory).

```
#Before (--cl-nv-maxrregcount unset or > 12)
ptxas info: Compiling entry function 'reductionVector' for 'sm_13'
ptxas info: Used 12 registers, 16+16 bytes smem, 236 bytes cmem[0],
 24 bytes cmem[1]

#After (--cl-nv-maxrregcount=4)
ptxas info    : Compiling entry function 'reductionVector' for 'sm_13'
ptxas info    : Used 4 registers, 8+0 bytes lmem, 16+16 bytes smem,
 236 bytes cmem[0], 24 bytes cmem[1]
```

- If you want to know what the compiler has actually done, you have to look at the PTX code (more on this later).
- NVidia kernels have **at least** 10 registers per thread (recent cards have 16 and Fermi has even more) so setting `maxrreg` lower than this is crazy.
- But *sometimes*, when you're using a lot of registers (40-120!), this option *might* help by increasing occupancy (ratio of actual running work-items to the maximum possible).
- Usually the programmer can do a better job than the compiler when reducing register usage.

- Or it "spills" **required** registers into lmem or "CUDA local" memory (not OpenCL *local* memory, it's slower like global memory).

```
#Before (--cl-nv-maxrregcount unset or > 12)
ptxas info: Compiling entry function 'reductionVector' for 'sm_13'
ptxas info: Used 12 registers, 16+16 bytes smem, 236 bytes cmem[0],
 24 bytes cmem[1]

#After (--cl-nv-maxrregcount=4)
ptxas info    : Compiling entry function 'reductionVector' for 'sm_13'
ptxas info    : Used 4 registers, 8+0 bytes lmem, 16+16 bytes smem,
 236 bytes cmem[0], 24 bytes cmem[1]
```

- If you want to know what the compiler has actually done, you have to look at the PTX code (more on this later).
- NVidia kernels have **at least** 10 registers per thread (recent cards have 16 and Fermi has even more) so setting `maxrreg` lower than this is crazy.
- But *sometimes*, when you're using a lot of registers (40-120!), this option *might* help by increasing occupancy (ratio of actual running work-items to the maximum possible).
- Usually the programmer can do a better job than the compiler when reducing register usage.

- Or it "spills" **required** registers into lmem or "CUDA local" memory (not OpenCL *local* memory, it's slower like global memory).

```
#Before (--cl-nv-maxrregcount unset or > 12)
ptxas info: Compiling entry function 'reductionVector' for 'sm_13'
ptxas info: Used 12 registers, 16+16 bytes smem, 236 bytes cmem[0],
 24 bytes cmem[1]

#After (--cl-nv-maxrregcount=4)
ptxas info    : Compiling entry function 'reductionVector' for 'sm_13'
ptxas info    : Used 4 registers, 8+0 bytes lmem, 16+16 bytes smem,
 236 bytes cmem[0], 24 bytes cmem[1]
```

- If you want to know what the compiler has actually done, you have to look at the PTX code (more on this later).
- NVidia kernels have **at least** 10 registers per thread (recent cards have 16 and Fermi has even more) so setting `maxrreg` lower than this is crazy.
- But *sometimes*, when you're using a lot of registers (40-120!), this option *might* help by increasing occupancy (ratio of actual running work-items to the maximum possible).
- Usually the programmer can do a better job than the compiler when reducing register usage.

- Or it "spills" **required** registers into lmem or "CUDA local" memory (not OpenCL *local* memory, it's slower like global memory).

```
#Before (-cl-nv-maxrregcount unset or > 12)
ptxas info: Compiling entry function 'reductionVector' for 'sm_13'
ptxas info: Used 12 registers, 16+16 bytes smem, 236 bytes cmem[0],
 24 bytes cmem[1]

#After (-cl-nv-maxrregcount=4)
ptxas info    : Compiling entry function 'reductionVector' for 'sm_13'
ptxas info    : Used 4 registers, 8+0 bytes lmem, 16+16 bytes smem,
 236 bytes cmem[0], 24 bytes cmem[1]
```

- If you want to know what the compiler has actually done, you have to look at the PTX code (more on this later).
- NVidia kernels have **at least** 10 registers per thread (recent cards have 16 and Fermi has even more) so setting `maxrreg` lower than this is crazy.
- But *sometimes*, when you're using a lot of registers (40-120!), this option *might* help by increasing occupancy (ratio of actual running work-items to the maximum possible).
- Usually the programmer can do a better job than the compiler when reducing register usage.

- Or it "spills" **required** registers into lmem or "CUDA local" memory (not OpenCL *local* memory, it's slower like global memory).

```
#Before (-cl-nv-maxrregcount unset or > 12)
ptxas info: Compiling entry function 'reductionVector' for 'sm_13'
ptxas info: Used 12 registers, 16+16 bytes smem, 236 bytes cmem[0],
 24 bytes cmem[1]

#After (-cl-nv-maxrregcount=4)
ptxas info    : Compiling entry function 'reductionVector' for 'sm_13'
ptxas info    : Used 4 registers, 8+0 bytes lmem, 16+16 bytes smem,
 236 bytes cmem[0], 24 bytes cmem[1]
```

- If you want to know what the compiler has actually done, you have to look at the PTX code (more on this later).
- NVidia kernels have **at least** 10 registers per thread (recent cards have 16 and Fermi has even more) so setting `maxrreg` lower than this is crazy.
- But *sometimes*, when you're using a lot of registers (40-120!), this option *might* help by increasing occupancy (ratio of actual running work-items to the maximum possible).
- Usually the programmer can do a better job than the compiler when reducing register usage.

- ATI does not have as many options, but one is particularly useful

```
std::string buildOptions;
buildOptions += " -g";
```

- This adds debug symbols to the generated binaries when compiling for the CPU, useful for debugging (next major part of the talk)!

- ATI does not have as many options, but one is particularly useful

```
std::string buildOptions;
buildOptions += " -g";
```

- This adds debug symbols to the generated binaries when compiling for the CPU, useful for debugging (next major part of the talk)!

Outline

1. ICD
   - What is the ICD extension?
   - Why use it?
   - How to use it?
2. Compiler Options
   - Standard Options
   - NVidia's Options
   - ATI's Options
3. Saving Kernel "Binaries"
   - About Intermediate Formats
   - NVidia
   - AMD
4. A Walk Through A PTX Kernel With Severin
   - Introduction to PTX
   - Syntax and Instruction Set of PTX
   - Code Sample

- The last section of this part of the talk deals with the intermediate (compiled) form of the kernels.
- AMD and NVidia have an intermediate language which the OpenCL C kernels are compiled into, called CAL and PTX respectively.
- Once the OpenCL compiler has been invoked, it is actually possible to obtain these compiled kernels, and this helps you figure out what exactly the compiler is doing.
- Sometimes this is helpful as the compiler can do strange things....

- The last section of this part of the talk deals with the intermediate (compiled) form of the kernels.
- AMD and NVidia have an intermediate language which the OpenCL C kernels are compiled into, called CAL and PTX respectively.
- Once the OpenCL compiler has been invoked, it is actually possible to obtain these compiled kernels, and this helps you figure out what exactly the compiler is doing.
- Sometimes this is helpful as the compiler can do strange things....

- The last section of this part of the talk deals with the intermediate (compiled) form of the kernels.
- AMD and NVidia have an intermediate language which the OpenCL C kernels are compiled into, called CAL and PTX respectively.
- Once the OpenCL compiler has been invoked, it is actually possible to obtain these compiled kernels, and this helps you figure out what exactly the compiler is doing.
- Sometimes this is helpful as the compiler can do strange things....

- The last section of this part of the talk deals with the intermediate (compiled) form of the kernels.
- AMD and NVidia have an intermediate language which the OpenCL C kernels are compiled into, called CAL and PTX respectively.
- Once the OpenCL compiler has been invoked, it is actually possible to obtain these compiled kernels, and this helps you figure out what exactly the compiler is doing.
- Sometimes this is helpful as the compiler can do strange things....

- NVidia allow you access to the PTX form of the kernels using the whole binary kernel section of OpenCL (see Section 5.4.1 and 5.4.5)
- This part of the standard is supposed to let you ship pre-compiled binaries with your code, if you didn't want to ship the kernel source code in plain text.
- This is a bit of a brain-dead thing to do:
  - You would have to ship the binaries for all target architectures and their variants.
  - Without pre-compiled kernels, your application will run on all future OpenCL compatible devices!
  - The intermediate code is still readable.
  - Good kernels are simple, which makes the intermediate code even more readable!
  - The loading of pre-compiled binaries rarely works anyway!
- Regardless, the interface is at least a standard way for obtaining the PTX code.

- NVidia allow you access to the PTX form of the kernels using the whole binary kernel section of OpenCL (see Section 5.4.1 and 5.4.5)
- This part of the standard is supposed to let you ship pre-compiled binaries with your code, if you didn't want to ship the kernel source code in plain text.
- This is a bit of a brain-dead thing to do:
  - You would have to ship the binaries for all target architectures and their variants.
  - Without pre-compiled kernels, your application will run on all future OpenCL compatible devices!
  - The intermediate code is still readable.
  - Good kernels are simple, which makes the intermediate code even more readable!
  - The loading of pre-compiled binaries rarely works anyway!
- Regardless, the interface is at least a standard way for obtaining the PTX code.

- NVidia allow you access to the PTX form of the kernels using the whole binary kernel section of OpenCL (see Section 5.4.1 and 5.4.5)
- This part of the standard is supposed to let you ship pre-compiled binaries with your code, if you didn't want to ship the kernel source code in plain text.
- This is a bit of a brain-dead thing to do:
  - You would have to ship the binaries for all target architectures and their variants.
  - Without pre-compiled kernels, your application will run on all future OpenCL compatible devices!
  - The intermediate code is still readable.
  - Good kernels are simple, which makes the intermediate code even more readable!
  - The loading of pre-compiled binaries rarely works anyway!
- Regardless, the interface is at least a standard way for obtaining the PTX code.

- NVidia allow you access to the PTX form of the kernels using the whole binary kernel section of OpenCL (see Section 5.4.1 and 5.4.5)
- This part of the standard is supposed to let you ship pre-compiled binaries with your code, if you didn't want to ship the kernel source code in plain text.
- This is a bit of a brain-dead thing to do:
  - You would have to ship the binaries for all target architectures and their variants.
  - Without pre-compiled kernels, your application will run on all future OpenCL compatible devices!
  - The intermediate code is still readable.
  - Good kernels are simple, which makes the intermediate code even more readable!
  - The loading of pre-compiled binaries rarely works anyway!
- Regardless, the interface is at least a standard way for obtaining the PTX code.

- NVidia allow you access to the PTX form of the kernels using the whole binary kernel section of OpenCL (see Section 5.4.1 and 5.4.5)
- This part of the standard is supposed to let you ship pre-compiled binaries with your code, if you didn't want to ship the kernel source code in plain text.
- This is a bit of a brain-dead thing to do:
  - You would have to ship the binaries for all target architectures and their variants.
  - Without pre-compiled kernels, your application will run on all future OpenCL compatible devices!
  - The intermediate code is still readable.
  - Good kernels are simple, which makes the intermediate code even more readable!
  - The loading of pre-compiled binaries rarely works anyway!
- Regardless, the interface is at least a standard way for obtaining the PTX code.

- NVidia allow you access to the PTX form of the kernels using the whole binary kernel section of OpenCL (see Section 5.4.1 and 5.4.5)
- This part of the standard is supposed to let you ship pre-compiled binaries with your code, if you didn't want to ship the kernel source code in plain text.
- This is a bit of a brain-dead thing to do:
  - You would have to ship the binaries for all target architectures and their variants.
  - Without pre-compiled kernels, your application will run on all future OpenCL compatible devices!
  - The intermediate code is still readable.
  - Good kernels are simple, which makes the intermediate code even more readable!
  - The loading of pre-compiled binaries rarely works anyway!
- Regardless, the interface is at least a standard way for obtaining the PTX code.

- NVidia allow you access to the PTX form of the kernels using the whole binary kernel section of OpenCL (see Section 5.4.1 and 5.4.5)
- This part of the standard is supposed to let you ship pre-compiled binaries with your code, if you didn't want to ship the kernel source code in plain text.
- This is a bit of a brain-dead thing to do:
  - You would have to ship the binaries for all target architectures and their variants.
  - Without pre-compiled kernels, your application will run on all future OpenCL compatible devices!
  - The intermediate code is still readable.
  - Good kernels are simple, which makes the intermediate code even more readable!
  - The loading of pre-compiled binaries rarely works anyway!
- Regardless, the interface is at least a standard way for obtaining the PTX code.

- NVidia allow you access to the PTX form of the kernels using the whole binary kernel section of OpenCL (see Section 5.4.1 and 5.4.5)
- This part of the standard is supposed to let you ship pre-compiled binaries with your code, if you didn't want to ship the kernel source code in plain text.
- This is a bit of a brain-dead thing to do:
  - You would have to ship the binaries for all target architectures and their variants.
  - Without pre-compiled kernels, your application will run on all future OpenCL compatible devices!
  - The intermediate code is still readable.
  - Good kernels are simple, which makes the intermediate code even more readable!
  - The loading of pre-compiled binaries rarely works anyway!
- Regardless, the interface is at least a standard way for obtaining the PTX code.

- NVidia allow you access to the PTX form of the kernels using the whole binary kernel section of OpenCL (see Section 5.4.1 and 5.4.5)
- This part of the standard is supposed to let you ship pre-compiled binaries with your code, if you didn't want to ship the kernel source code in plain text.
- This is a bit of a brain-dead thing to do:
  - You would have to ship the binaries for all target architectures and their variants.
  - Without pre-compiled kernels, your application will run on all future OpenCL compatible devices!
  - The intermediate code is still readable.
  - Good kernels are simple, which makes the intermediate code even more readable!
  - The loading of pre-compiled binaries rarely works anyway!
- Regardless, the interface is at least a standard way for obtaining the PTX code.

- To write the PTX to a file we need some horrible code:-

```
//Allocate some memory for all the kernel binary data
const std::vector<size_t> binSizes
    = program.getInfo<CL_PROGRAM_BINARY_SIZES>();

std::vector<unsigned char>
  binData(std::accumulate(binSizes.begin(), binSizes.end(), 0));

unsigned char* binChunk = &binData[0];

//A list of pointers to the binary data
std::vector<unsigned char*> binaries;

for(size_t i = 0; i < binSizes.size(); ++i) {
  binaries.push_back(binChunk);
  binChunk += binSizes[i];
}

program.getInfo(CL_PROGRAM_BINARIES, &binaries[0]);

std::ofstream binaryfile("kernels.bin");

if(!binaryfile.good())
  std::runtime_error("Failed to open kernels.bin for reading");

for(size_t i = 0; i < binaries.size(); ++i)
  binaryfile << binaries[i];
```

- Now we have a file 'kernels.bin', filled with delicious PTX code.

```
//
// Generated by NVIDIA NVPTX Backend for LLVM
//

.version 1.5
.target sm_13, texmode_independent

// Global Launch Offsets
.const[0] .s32 %_global_block_offset[3];
.const[0] .s32 %_global_launch_offset[3];
.const[0] .s32 %_global_num_groups[3];
.const[0] .s32 %_global_size[3];
.const[0] .u32 %_work_dim;
```

- Severin will talk about PTX code in a bit, but first we'll quickly cover how to obtain CAL code for AMD GPU's.

- Now we have a file 'kernels.bin', filled with delicious PTX code.

```
//
// Generated by NVIDIA NVPTX Backend for LLVM
//

.version 1.5
.target sm_13, texmode_independent

// Global Launch Offsets
.const[0] .s32 %_global_block_offset[3];
.const[0] .s32 %_global_launch_offset[3];
.const[0] .s32 %_global_num_groups[3];
.const[0] .s32 %_global_size[3];
.const[0] .u32 %_work_dim;
```

- Severin will talk about PTX code in a bit, but first we'll quickly cover how to obtain CAL code for AMD GPU's.

- AMD has a less standard but significantly easier way of dumping the CAL code.
- Simply set the following environment variable and the kernel source is dumped into some files in the current directory

```
myuser@mypc Examples/Ex8.Debugging $ GPU_DUMP_DEVICE_KERNEL=1 ./CLDebug -d 1
...
myuser@mypc Examples/Ex8.Debugging $ ls
CLDebug  CLDebug.cpp  CLDebug.o  Makefile  include  kernels.bin  kernels.cl
reductionNVIDIA_Cypress.il  reductionVector_Cypress.il  reduction_Cypress.il
```

- AMD has a less standard but significantly easier way of dumping the CAL code.
- Simply set the following environment variable and the kernel source is dumped into some files in the current directory

```
myuser@mypc Examples/Ex8.Debugging $ GPU_DUMP_DEVICE_KERNEL=1 ./CLDebug -d 1
...
myuser@mypc Examples/Ex8.Debugging $ ls
CLDebug  CLDebug.cpp  CLDebug.o  Makefile  include  kernels.bin  kernels.cl
reductionNVIDIA_Cypress.il  reductionVector_Cypress.il  reduction_Cypress.il
```

- There is a file for each kernel, each filled with succulent CAL code.

```
mdef(96)_out(1)_in(1)
mov r0, in0
fence_threads_lds
mov out0, r0
mend
il_cs_2_0
dcl_cb cb0[9] ; Constant buffer that holds ABI data
dcl_literal l0, 4, 1, 2, 3
dcl_literal l1, 0x00FFFFFF, -1, -2, -3
dcl_literal l2, 0x0000FFFF, 0xFFFFFFFE,0x000000FF,0xFFFFFFFC
dcl_literal l3, 24, 16, 8, 0xFFFFFFFF
dcl_literal l4, 0xFFFFFF00, 0xFFFF0000, 0xFF00FFFF, 0xFFFF00FF
dcl_literal l5, 0, 4, 8, 12
dcl_literal l6, 32, 32, 32, 32
mov r769, cb0[8].x
call 1204;$
endmain
func 1204 ; __OpenCL_reduction_kernel
mov r770, l1.0
dcl_literal l7, 0x00000002, 0x00000002, 0x00000002, 0x00000002; int: 2
dcl_literal l8, 0x00000000, 0x00000000, 0x00000000, 0x00000000; int: 0
dcl_literal l9, 0x00000001, 0x00000001, 0x00000001, 0x00000001; int: 1
dcl_literal l10, 0x00000080, 0x00000080, 0x00000080, 0x00000080; int: 128
dcl_literal l11, 0x00000008, 0x00000008, 0x00000008, 0x00000008; int: 8
dcl_literal l12, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff; int: 429...
dcl_lds_id(1) 32768
dcl_max_thread_per_group 256
```

- Now Severin will talk us through some PTX code.

- There is a file for each kernel, each filled with succulent CAL code.

```
mdef(96)_out(1)_in(1)
mov r0, in0
fence_threads_lds
mov out0, r0
mend
il_cs_2_0
dcl_cb cb0[9] ; Constant buffer that holds ABI data
dcl_literal l0, 4, 1, 2, 3
dcl_literal l1, 0x00FFFFFF, -1, -2, -3
dcl_literal l2, 0x0000FFFF, 0xFFFFFFFE,0x000000FF,0xFFFFFFFC
dcl_literal l3, 24, 16, 8, 0xFFFFFFFF
dcl_literal l4, 0xFFFFFF00, 0xFFFF0000, 0xFF00FFFF, 0xFFFF00FF
dcl_literal l5, 0, 4, 8, 12
dcl_literal l6, 32, 32, 32, 32
mov r769, cb0[8].x
call 1204;$
endmain
func 1204 ; __OpenCL_reduction_kernel
mov r770, l1.0
dcl_literal l7, 0x00000002, 0x00000002, 0x00000002, 0x00000002; int: 2
dcl_literal l8, 0x00000000, 0x00000000, 0x00000000, 0x00000000; int: 0
dcl_literal l9, 0x00000001, 0x00000001, 0x00000001, 0x00000001; int: 1
dcl_literal l10, 0x00000080, 0x00000080, 0x00000080, 0x00000080; int: 128
dcl_literal l11, 0x00000008, 0x00000008, 0x00000008, 0x00000008; int: 8
dcl_literal l12, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff; int: 429...
dcl_lds_id(1) 32768
dcl_max_thread_per_group 256
```

- Now Severin will talk us through some PTX code.

Outline

1. ICD
   - What is the ICD extension?
   - Why use it?
   - How to use it?
2. Compiler Options
   - Standard Options
   - NVidia's Options
   - ATI's Options
3. Saving Kernel "Binaries"
   - About Intermediate Formats
   - NVidia
   - AMD
4. A Walk Through A PTX Kernel With Severin
   - Introduction to PTX
   - Syntax and Instruction Set of PTX
   - Code Sample

- PTX (Parallel Thread Execution) is a assembly-like intermediate language developed by NVIDIA.
- The compilers for the GPGPU solutions of NVIDIA (OpenCL, Compute Unified Device Architecture) provide backends for generating PTX code.
- It's an abstraction level on top of the actual code run on the GPUs.
- The current version of PTX is 2.0 and supports all the fancy features of the new Fermi GPU architecture.
- The full documentation describing the PTX virtual machine and the instruction set architecture is freely available and spans around 170 pages.

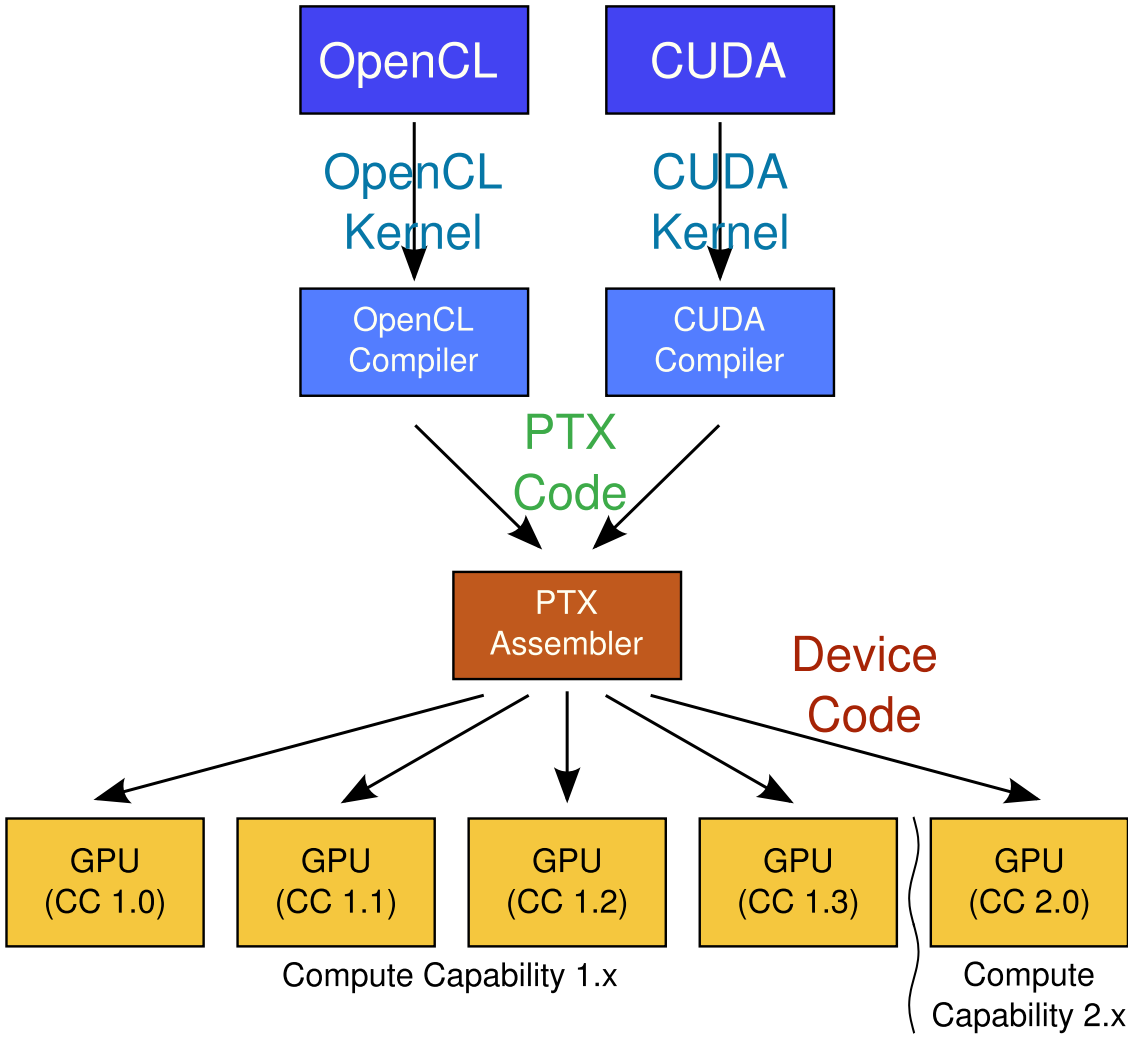


Figure: PTX in the context of the NVIDIA architecture.

- PTX uses a plain text (ASCII) format and borrows some features from C (preprocessor directives, comments).
- The actual code consists of two types of statements: directives and instructions.
- Directive statements start with a dot ('.') and are used to distinguish address spaces and provide scopes and some meta data.
- Instruction statements are the main part of the code and are formed by an opcode and a list of operands (zero to four, depending on the operation).

- The main classes of opcodes are:
  - memory and texture access
  - arithmetic instructions (integer, floating point)
  - comparison and control flow
  - logic, shift and conversion
  - control flow
- Most opcodes use one or several type specifiers to identify the type of the operands (e.g. .s32, .u64, .f32, .f64).
- The arithmetic instructions additionally include modifiers regarding rounding mode and handling of denormalized numbers (floating point only).

Let's have a look at some real code!

Part II

Debugging OpenCL

Outline

5 Printing To The Screen
- Availability
- Usage
- Combining With Defines

6 GDB and OpenCL
- Prerequisites
- Running GDB

- Functions/Objects like `printf` and `std::cout` are the simplest and most common form of debugging.
- If something goes wrong you can write out enough data to help you figure out what happened.
- But in parallel environments these commands are usually scarce as they are not inherently thread-safe.
- This is even worse for GPU's, the only output you usually get is in the form of a buffer!

- Functions/Objects like `printf` and `std::cout` are the simplest and most common form of debugging.
- If something goes wrong you can write out enough data to help you figure out what happened.
- But in parallel environments these commands are usually scarce as they are not inherently thread-safe.
- This is even worse for GPU's, the only output you usually get is in the form of a buffer!

- Functions/Objects like `printf` and `std::cout` are the simplest and most common form of debugging.
- If something goes wrong you can write out enough data to help you figure out what happened.
- But in parallel environments these commands are usually scarce as they are not inherently thread-safe.
- This is even worse for GPU's, the only output you usually get is in the form of a buffer!

- Functions/Objects like `printf` and `std::cout` are the simplest and most common form of debugging.
- If something goes wrong you can write out enough data to help you figure out what happened.
- But in parallel environments these commands are usually scarce as they are not inherently thread-safe.
- This is even worse for GPU's, the only output you usually get is in the form of a buffer!

- But these functions are so useful they often are implemented!
- NVidia has implemented `printf` for it's CUDA architecture, but it's not available in OpenCL, yet!
- But AMD has implemented `printf` for it's **CPU** implementation (extension `cl_amd_printf`).

- But these functions are so useful they often are implemented!
- NVidia has implemented `printf` for it's CUDA architecture, but it's not available in OpenCL, yet!
- But AMD has implemented `printf` for it's **CPU** implementation (extension `cl_amd_printf`).

- But these functions are so useful they often are implemented!
- NVidia has implemented `printf` for it's CUDA architecture, but it's not available in OpenCL, yet!
- But AMD has implemented `printf` for it's **CPU** implementation (extension `cl_amd_printf`).

- As it's an extension that provides some kernel functions you must first enable it inside your kernel source.

```
#pragma OPENCL EXTENSION  cl_amd_printf : enable

__kernel void mykernel()
{
  ...
  printf("Greetings from work item %d", get_global_id(0));
}
```

- But we don't really want to write two versions of the code, one with printf functions and one without...
- We can use defines as a neat way of controlling `printf` output!

- As it's an extension that provides some kernel functions you must first enable it inside your kernel source.

```
#pragma OPENCL EXTENSION  cl_amd_printf : enable

__kernel void mykernel()
{
  ...
  printf("Greetings from work item %d", get_global_id(0));
}
```

- But we don't really want to write two versions of the code, one with printf functions and one without...
- We can use defines as a neat way of controlling `printf` output!

- As it's an extension that provides some kernel functions you must first enable it inside your kernel source.

```
#pragma OPENCL EXTENSION  cl_amd_printf : enable

__kernel void mykernel()
{
  ...
  printf("Greetings from work item %d", get_global_id(0));
}
```

- But we don't really want to write two versions of the code, one with printf functions and one without...
- We can use defines as a neat way of controlling `printf` output!

- First, wrap our `printf` statements in some `#ifdef`'s

```
#ifdef ATI_DEBUG_MODE
#pragma OPENCL EXTENSION  cl_amd_printf : enable
#endif
__kernel void mykernel()
{
#ifdef ATI_DEBUG_MODE
  printf("Greetings from work item %d", get_global_id(0));
#endif
  ...
}
```

- Now, we can detect if we can use the printf statments, and enable it by defining an extra build option

```
std::string extensions = devices[0].getInfo<CL_DEVICE_EXTENSIONS>();
...
if (ATIdebug && (extensions.find("cl_amd_printf") != std::string::npos))
  buildOptions += " -g  -DATI_DEBUG_MODE";
...
program.build(devices, buildOptions.c_str());
```

- First, wrap our `printf` statements in some `#ifdef`'s

```
#ifdef ATI_DEBUG_MODE
#pragma OPENCL EXTENSION  cl_amd_printf : enable
#endif
__kernel void mykernel()
{
#ifdef ATI_DEBUG_MODE
  printf("Greetings from work item %d", get_global_id(0));
#endif
  ...
}
```

- Now, we can detect if we can use the printf statments, and enable it by defining an extra build option

```
std::string extensions = devices[0].getInfo<CL_DEVICE_EXTENSIONS>();
...
if (ATIdebug && (extensions.find("cl_amd_printf") != std::string::npos))
  buildOptions += " -g  -DATI_DEBUG_MODE";
...
program.build(devices, buildOptions.c_str());
```

- So this is the most rudimentary form of debugging, being able to `printf` debugging information from inside of a kernel.
- We can use some of the useful functions `isnan`, `isinf` or `isfinite` to make sure that we have sane numbers inside of our kernels.
- NaN's, infinite values and other numerical errors are the usual failures in kernels, as not much complicated usually occurs in them.
- But is there a more powerful method to debug OpenCL code?

- So this is the most rudimentary form of debugging, being able to `printf` debugging information from inside of a kernel.
- We can use some of the useful functions `isnan`, `isinf` or `isfinite` to make sure that we have sane numbers inside of our kernels.
- NaN's, infinite values and other numerical errors are the usual failures in kernels, as not much complicated usually occurs in them.
- But is there a more powerful method to debug OpenCL code?

- So this is the most rudimentary form of debugging, being able to `printf` debugging information from inside of a kernel.
- We can use some of the useful functions `isnan`, `isinf` or `isfinite` to make sure that we have sane numbers inside of our kernels.
- NaN's, infinite values and other numerical errors are the usual failures in kernels, as not much complicated usually occurs in them.
- But is there a more powerful method to debug OpenCL code?

- So this is the most rudimentary form of debugging, being able to `printf` debugging information from inside of a kernel.
- We can use some of the useful functions `isnan`, `isinf` or `isfinite` to make sure that we have sane numbers inside of our kernels.
- NaN's, infinite values and other numerical errors are the usual failures in kernels, as not much complicated usually occurs in them.
- But is there a more powerful method to debug OpenCL code?

Outline

5 Printing To The Screen
- Availability
- Usage
- Combining With Defines

6 GDB and OpenCL
- Prerequisites
- Running GDB

- When running on AMD's CPU implementation, and when the host program and kernel code are compiled with debugging flags ("-g"), we can actually use a debugger to inspect a running kernel!
- You can either set the debug flag in the build options you pass to the OpenCL compiler

```
std::string extensions = devices[0].getInfo<CL_DEVICE_EXTENSIONS>();
...
if (ATIdebug && (extensions.find("cl_amd_printf") != std::string::npos))
  buildOptions += " -g -DATI_DEBUG_MODE";
...
program.build(devices, buildOptions.c_str());
```

- Or you can use an environment variable to set additional OpenCL CPU compiler flags

```
myuser@mypc ~/.../Ex8.Debugging $ export CPU_COMPILER_OPTIONS="-g"
```

- When running on AMD's CPU implementation, and when the host program and kernel code are compiled with debugging flags ("-g"), we can actually use a debugger to inspect a running kernel!
- You can either set the debug flag in the build options you pass to the OpenCL compiler

```
std::string extensions = devices[0].getInfo<CL_DEVICE_EXTENSIONS>();
...
if (ATIdebug && (extensions.find("cl_amd_printf") != std::string::npos))
  buildOptions += " -g  -DATI_DEBUG_MODE";
...
program.build(devices, buildOptions.c_str());
```

- Or you can use an environment variable to set additional OpenCL CPU compiler flags

```
myuser@mypc ~/.../Ex8.Debugging $ export CPU_COMPILER_OPTIONS="-g"
```

- When running on AMD's CPU implementation, and when the host program and kernel code are compiled with debugging flags ("-g"), we can actually use a debugger to inspect a running kernel!
- You can either set the debug flag in the build options you pass to the OpenCL compiler

```
std::string extensions = devices[0].getInfo<CL_DEVICE_EXTENSIONS>();
...
if (ATIdebug && (extensions.find("cl_amd_printf") != std::string::npos))
  buildOptions += " -g  -DATI_DEBUG_MODE";
...
program.build(devices, buildOptions.c_str());
```

- Or you can use an environment variable to set additional OpenCL CPU compiler flags

```
myuser@mypc ~/.../Ex8.Debugging $ export CPU_COMPILER_OPTIONS="-g"
```

- There is an additional requirement, you must link your program with the `pthread` library (add -pthread to your g++ invocation), otherwise gdb will give this helpful error...

```
[Thread debugging using libthread_db enabled]
Cannot find new threads: generic error
(gdb)
```

- Also, you might wish to reduce the number of parallel threads to one, (essentially serialize your code).
- This makes it easier to debug although you might miss synchronization errors.
- It is easy to reduce the number of active threads on the CPU using an environment variable...

```
myuser@mypc ~/.../Ex8.Debugging $ export CPU_MAX_COMPUTE_UNITS=1
```

- There is an additional requirement, you must link your program with the `pthread` library (add -pthread to your g++ invocation), otherwise gdb will give this helpful error...

```
[Thread debugging using libthread_db enabled]
Cannot find new threads: generic error
(gdb)
```

- Also, you might wish to reduce the number of parallel threads to one, (essentially serialize your code).
- This makes it easier to debug although you might miss synchronization errors.
- It is easy to reduce the number of active threads on the CPU using an environment variable...

```
myuser@mypc ~/.../Ex8.Debugging $ export CPU_MAX_COMPUTE_UNITS=1
```

- There is an additional requirement, you must link your program with the `pthread` library (add -pthread to your g++ invocation), otherwise gdb will give this helpful error...

```
[Thread debugging using libthread_db enabled]
Cannot find new threads: generic error
(gdb)
```

- Also, you might wish to reduce the number of parallel threads to one, (essentially serialize your code).
- This makes it easier to debug although you might miss synchronization errors.
- It is easy to reduce the number of active threads on the CPU using an environment variable...

```
myuser@mypc ~/.../Ex8.Debugging $ export CPU_MAX_COMPUTE_UNITS=1
```

- There is an additional requirement, you must link your program with the `pthread` library (add -pthread to your g++ invocation), otherwise gdb will give this helpful error...

```
[Thread debugging using libthread_db enabled]
Cannot find new threads: generic error
(gdb)
```

- Also, you might wish to reduce the number of parallel threads to one, (essentially serialize your code).
- This makes it easier to debug although you might miss synchronization errors.
- It is easy to reduce the number of active threads on the CPU using an environment variable...

```
myuser@mypc ~/.../Ex8.Debugging $ export CPU_MAX_COMPUTE_UNITS=1
```

- The final requirement is that you have a gdb-like debugger.
- Most debuggers on linux wrap or base themselves around gdb (eclipse CDT, kdbg, DDD, emacs, etc.).
- Here, we copy the ATI guide on debugging and use the gdb console to type our commands in.

  `http://developer.amd.com/ZONES/OPENCLZONE/pages/debuggingopenclapps.aspx`
- But this can all be done in a GUI too (I'll actually show you this in emacs).

- The final requirement is that you have a gdb-like debugger.
- Most debuggers on linux wrap or base themselves around gdb (eclipse CDT, kdbg, DDD, emacs, etc.).
- Here, we copy the ATI guide on debugging and use the gdb console to type our commands in.

  `http://developer.amd.com/ZONES/OPENCLZONE/pages/debuggingopenclapps.aspx`
- But this can all be done in a GUI too (I'll actually show you this in emacs).

- The final requirement is that you have a gdb-like debugger.
- Most debuggers on linux wrap or base themselves around gdb (eclipse CDT, kdbg, DDD, emacs, etc.).
- Here, we copy the ATI guide on debugging and use the gdb console to type our commands in.

  `http://developer.amd.com/ZONES/OPENCLZONE/pages/debuggingopenclapps.aspx`
- But this can all be done in a GUI too (I'll actually show you this in emacs).

- The final requirement is that you have a gdb-like debugger.
- Most debuggers on linux wrap or base themselves around gdb (eclipse CDT, kdbg, DDD, emacs, etc.).
- Here, we copy the ATI guide on debugging and use the gdb console to type our commands in.

  `http://developer.amd.com/ZONES/OPENCLZONE/pages/debuggingopenclapps.aspx`
- But this can all be done in a GUI too (I'll actually show you this in emacs).

- There is an additional requirement, you must link your program with the `pthread` library (add -pthread to your g++ invocation), otherwise gdb will give this helpful error...

```
[Thread debugging using libthread_db enabled]
Cannot find new threads: generic error
(gdb)
```

- Also, you might wish to reduce the number of parallel threads to one, (essentially serialize your code).
- This makes it easier to debug although you might miss synchronization errors.
- It is easy to reduce the number of active threads on the CPU using an environment variable...

```
myuser@mypc ~/.../Ex8.Debugging $ export CPU_MAX_COMPUTE_UNITS=1
```

- We're now ready to run our example code!

- There is an additional requirement, you must link your program with the `pthread` library (add -pthread to your g++ invocation), otherwise gdb will give this helpful error...

```
[Thread debugging using libthread_db enabled]
Cannot find new threads: generic error
(gdb)
```

- Also, you might wish to reduce the number of parallel threads to one, (essentially serialize your code).
- This makes it easier to debug although you might miss synchronization errors.
- It is easy to reduce the number of active threads on the CPU using an environment variable...

```
myuser@mypc ~/.../Ex8.Debugging $ export CPU_MAX_COMPUTE_UNITS=1
```

- We're now ready to run our example code!

- There is an additional requirement, you must link your program with the `pthread` library (add -pthread to your g++ invocation), otherwise gdb will give this helpful error...

```
[Thread debugging using libthread_db enabled]
Cannot find new threads: generic error
(gdb)
```

- Also, you might wish to reduce the number of parallel threads to one, (essentially serialize your code).
- This makes it easier to debug although you might miss synchronization errors.
- It is easy to reduce the number of active threads on the CPU using an environment variable...

```
myuser@mypc ~/.../Ex8.Debugging $ export CPU_MAX_COMPUTE_UNITS=1
```

- We're now ready to run our example code!

- There is an additional requirement, you must link your program with the `pthread` library (add -pthread to your g++ invocation), otherwise gdb will give this helpful error...

```
[Thread debugging using libthread_db enabled]
Cannot find new threads: generic error
(gdb)
```

- Also, you might wish to reduce the number of parallel threads to one, (essentially serialize your code).
- This makes it easier to debug although you might miss synchronization errors.
- It is easy to reduce the number of active threads on the CPU using an environment variable...

```
myuser@mypc ~/.../Ex8.Debugging $ export CPU_MAX_COMPUTE_UNITS=1
```

- We're now ready to run our example code!

- There is an additional requirement, you must link your program with the `pthread` library (add -pthread to your g++ invocation), otherwise gdb will give this helpful error...

```
[Thread debugging using libthread_db enabled]
Cannot find new threads: generic error
(gdb)
```

- Also, you might wish to reduce the number of parallel threads to one, (essentially serialize your code).
- This makes it easier to debug although you might miss synchronization errors.
- It is easy to reduce the number of active threads on the CPU using an environment variable...

```
myuser@mypc ~/.../Ex8.Debugging $ export CPU_MAX_COMPUTE_UNITS=1
```

- We're now ready to run our example code!

- The commands to start your debugger are well documented in the ATI programming guide, but I'll repeat them here too.
- Start gdb and point it to your program (built with debug flags).

```
$ gdb ./CLDebug
warning: Can not parse XML syscalls information; XML support was disabled...
GNU gdb (Gentoo 7.0.1 p1) 7.0.1
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.h...
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.gentoo.org/>...
Reading symbols from /..../Ex8.Debugging/CLDebug...done.
(gdb)
```

- Now, set a breakpoint just before the kernel you want to debug. We'll just set it before any kernel is run...

```
(gdb) b clEnqueueNDRangeKernel
Breakpoint 1 at 0x402c38
(gdb)
```

- The commands to start your debugger are well documented in the ATI programming guide, but I'll repeat them here too.
- Start gdb and point it to your program (built with debug flags).

```
$ gdb ./CLDebug
warning: Can not parse XML syscalls information; XML support was disabled...
GNU gdb (Gentoo 7.0.1 p1) 7.0.1
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.h...
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.gentoo.org/>...
Reading symbols from /..../Ex8.Debugging/CLDebug...done.
(gdb)
```

- Now, set a breakpoint just before the kernel you want to debug. We'll just set it before any kernel is run...

```
(gdb) b clEnqueueNDRangeKernel
Breakpoint 1 at 0x402c38
(gdb)
```

- The commands to start your debugger are well documented in the ATI programming guide, but I'll repeat them here too.
- Start gdb and point it to your program (built with debug flags).

```
$ gdb ./CLDebug
warning: Can not parse XML syscalls information; XML support was disabled...
GNU gdb (Gentoo 7.0.1 p1) 7.0.1
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.h...
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.gentoo.org/>...
Reading symbols from /..../Ex8.Debugging/CLDebug...done.
(gdb)
```

- Now, set a breakpoint just before the kernel you want to debug. We'll just set it before any kernel is run...

```
(gdb) b clEnqueueNDRangeKernel
Breakpoint 1 at 0x402c38
(gdb)
```

- Now run the example code with the –debug flag which builds the kernels with debugging symbols

```
(gdb) r ––debug
Starting program: /...Ex8.Debugging/CLDebug ––debug
[Thread debugging using libthread_db enabled]
...(Output of the program ommitted)
...
New Thread 0x7ffff7ebd710 (LWP 17567)]
[New Thread 0x7ffff7e7c710 (LWP 17568)]

Breakpoint 1, 0x00007ffff7ec6bb0 in clEnqueueNDRangeKernel () from
  /usr/lib/libOpenCL.so.1
(gdb)
```

- Our debugger is now at the point in the code where some kernel is about to be run.

- Now run the example code with the –debug flag which builds the kernels with debugging symbols

```
(gdb) r --debug
Starting program: /...Ex8.Debugging/CLDebug --debug
[Thread debugging using libthread_db enabled]
...(Output of the program ommitted)
...
New Thread 0x7ffff7ebd710 (LWP 17567)]
[New Thread 0x7ffff7e7c710 (LWP 17568)]

Breakpoint 1, 0x00007ffff7ec6bb0 in clEnqueueNDRangeKernel () from
  /usr/lib/libOpenCL.so.1
(gdb)
```

- Our debugger is now at the point in the code where some kernel is about to be run.

- Now that our kernel is compiled and about to be run, we can actually look for it in the debugger (we couldn't find and set breakpoint in the kernel before as it wasn't even compiled yet!).
- We ask gdb to list all functions beginning with `_OpenCL_`...

```
(gdb) i functions _OpenCL
All functions matching regular expression "_OpenCL":

File OCLXqado6.cl:
void __OpenCL_reductionNVIDIA_kernel(const float *, void *, ...);
void __OpenCL_reductionVector_kernel(const float *, void *, ...);
void __OpenCL_reduction_kernel(const float *, void *, const ...);

Non-debugging symbols:
.... (ignore these)
```

- The kernel our example runs for AMD CPU's is the `__OpenCl_reductionVector_kernel`, so we must set a breakpoint there...

```
(gdb) b __OpenCL_reductionVector_kernel
Breakpoint 2 at 0x7ffff270d7b8: file OCLXqado6.cl, line 67.
(gdb)
```

- Now that our kernel is compiled and about to be run, we can actually look for it in the debugger (we couldn't find and set breakpoint in the kernel before as it wasn't even compiled yet!).
- We ask gdb to list all functions beginning with `_OpenCL_`...

```
(gdb) i functions _OpenCL
All functions matching regular expression "_OpenCL":

File OCLXqado6.cl:
void __OpenCL_reductionNVIDIA_kernel(const float *, void *, ...);
void __OpenCL_reductionVector_kernel(const float *, void *, ...);
void __OpenCL_reduction_kernel(const float *, void *, const ...);

Non-debugging symbols:
.... (ignore these)
```

- The kernel our example runs for AMD CPU's is the `__OpenCl_reductionVector_kernel`, so we must set a breakpoint there...

```
(gdb) b __OpenCL_reductionVector_kernel
Breakpoint 2 at 0x7ffff270d7b8: file OCLXqado6.cl, line 67.
(gdb)
```

- Now that our kernel is compiled and about to be run, we can actually look for it in the debugger (we couldn't find and set breakpoint in the kernel before as it wasn't even compiled yet!).
- We ask gdb to list all functions beginning with `_OpenCL_`...

```
(gdb) i functions _OpenCL
All functions matching regular expression "_OpenCL":

File OCLXqado6.cl:
void __OpenCL_reductionNVIDIA_kernel(const float *, void *, ...);
void __OpenCL_reductionVector_kernel(const float *, void *, ...);
void __OpenCL_reduction_kernel(const float *, void *, const ...);

Non-debugging symbols:
.... (ignore these)
```

- The kernel our example runs for AMD CPU's is the `__OpenCl_reductionVector_kernel`, so we must set a breakpoint there...

```
(gdb) b __OpenCL_reductionVector_kernel
Breakpoint 2 at 0x7ffff270d7b8: file OCLXqado6.cl, line 67.
(gdb)
```

- Now we let the debugger continue to the next breakpoint (inside the kernel)...

```
(gdb) c
Continuing.
[Switching to Thread 0x7ffff7e7c710 (LWP 17568)]

Breakpoint 2, __OpenCL_reductionVector_kernel (input=0x7fffe270a000,
                    output=0x60f000, size=33554432, buffer=0xb58c00)
                                                        at OCLXqado6.cl:67
67    printf("Work Item %d", get_global_id(0));
(gdb)
```

- You can see we're at line 67, if we type `list` we can even see line 67 and the surrounding code.

```
(gdb) list
62
63 __kernel void reductionVector(__global const float* input,
64 __global float* output, const unsigned int size, ...) {
65
66 #ifdef ATI_DEBUG_MODE
67   printf("Work Item %d", get_global_id(0));
68 #endif
69
70 size_t idx = get_local_id(0);
71 buffer[idx] = 0;
(gdb)
```

- Now we let the debugger continue to the next breakpoint (inside the kernel)...

```
(gdb) c
Continuing.
[Switching to Thread 0x7ffff7e7c710 (LWP 17568)]

Breakpoint 2, __OpenCL_reductionVector_kernel (input=0x7fffe270a000,
                    output=0x60f000, size=33554432, buffer=0xb58c00)
                                                        at OCLXqado6.cl:67
67    printf("Work Item %d", get_global_id(0));
(gdb)
```

- You can see we're at line 67, if we type `list` we can even see line 67 and the surrounding code.

```
(gdb) list
62
63 __kernel void reductionVector(__global const float* input,
64 __global float* output, const unsigned int size, ...) {
65
66 #ifdef ATI_DEBUG_MODE
67    printf("Work Item %d", get_global_id(0));
68 #endif
69
70 size_t idx = get_local_id(0);
71 buffer[idx] = 0;
(gdb)
```

- We can print the output of functions or variables

```
(gdb) p get_global_id(0)
$1 = 0
(gdb)
```

- Or run to the next line of code

```
(gdb) n
70 size_t idx = get_local_id(0);
(gdb) n
71 buffer[idx] = 0;
(gdb) n
67   printf("Work Item %d", get_global_id(0));
(gdb) n
74 for(size_t pos = get_global_id(0); pos < size / 4;
```

- The rest I'll leave up to the many gdb tutorials on the net!

- We can print the output of functions or variables

```
(gdb) p get_global_id(0)
$1 = 0
(gdb)
```

- Or run to the next line of code

```
(gdb) n
70 size_t idx = get_local_id(0);
(gdb) n
71 buffer[idx] = 0;
(gdb) n
67   printf("Work Item %d", get_global_id(0));
(gdb) n
74 for(size_t pos = get_global_id(0); pos < size / 4;
```

- The rest I'll leave up to the many gdb tutorials on the net!

- We can print the output of functions or variables

```
(gdb) p get_global_id(0)
$1 = 0
(gdb)
```

- Or run to the next line of code

```
(gdb) n
70 size_t idx = get_local_id(0);
(gdb) n
71 buffer[idx] = 0;
(gdb) n
67   printf("Work Item %d", get_global_id(0));
(gdb) n
74 for(size_t pos = get_global_id(0); pos < size / 4;
```

- The rest I'll leave up to the many gdb tutorials on the net!