Programmable Address Spaces

Systems Seminar - University of Glasgow

Dr. Paul Keir, Andrew Gozillon

School of Engineering and Computing
University of the West of Scotland, Paisley Campus

March 15th, 2017

Overview

- Address Spaces in Hardware & Software
- Compiler Support
- Involving C++ Templates
- Testing with OpenCL
- Conclusion

GPGPU Thread Hierarchy

- Single Instruction Multiple Threads (SIMT)
- Memory latency is mitigated by
  - launching many threads in lock-step; and
  - switching warps/wavefronts whenever an operand isn't ready.

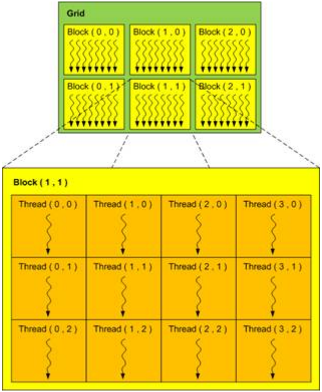


Image `http://cuda.ce.rit.edu/cuda_overview/cuda_overview.htm`

GPGPU Memory Hierarchy

- Registers and local memory are unique to a thread
- Shared memory is unique to a block
- Global, constant, and texture memories exist across all blocks
- The scope of these disjoint memory banks is shown below
- 2 threads execute in each of 2 blocks (4 threads):

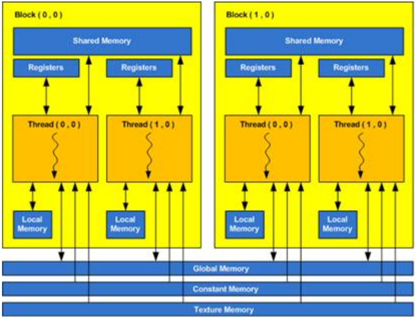


Image `http://cuda.ce.rit.edu/cuda_overview/cuda_overview.htm`

Address Space Qualifiers

- Simple processors employed in large numbers
- Hardware and also software caching is routinely absent
- Memory banks are abstracted by address space qualifiers
- OpenCL C recognises 4 disjoint address spaces:
  - Global, constant, local and private
  - An array declared in fast, shared, on-chip memory:
  - `__local float x[10];`

Cell Broadband Engine Architecture (2000)

- In addition to the main memory of the PPU host
- 256 KB of fast local memory was available to each SPU

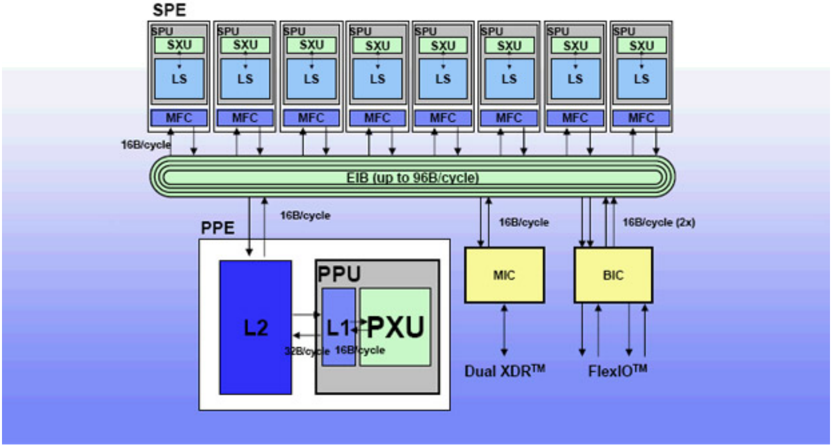


Image `http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/cellengine/`

Accessing PPU variables from SPU programs

- IBM XL C/C++ provided PPE address space support on SPE
- Effective address space support, with a software cache
- The `__ea` type qualifier was provided as an extension

```
extern int __ea i;
```

- Indicates that the variable being declared in SPU code, already exists in the PPE address space

Accessing PPU variables from SPU programs

- IBM XL C/C++ provided PPE address space support on SPE
- Effective address space support, with a software cache
- The `__ea` type qualifier was provided as an extension

```
extern int __ea i;
```

- Indicates that the variable being declared in SPU code, already exists in the PPE address space

A pointer in PPU address space pointing to PPU address space:

```
extern __ea int* __ea p;
```

Accessing PPU variables from SPU programs

- IBM XL C/C++ provided PPE address space support on SPE
- Effective address space support, with a software cache
- The `__ea` type qualifier was provided as an extension

```
extern int __ea i;
```

- Indicates that the variable being declared in SPU code, already exists in the PPE address space

A pointer in PPU address space pointing to PPU address space:

```
extern __ea int* __ea p;
```

Dynamic memory allocation was also available from the SPU:

```
{
  __ea int *p = malloc_ea(sizeof(int));
}
```

Duplicated Effort

Many recent SDKs support multiple address space programming

For example, the compilers which implement:

- NVIDIA CUDA
- OpenCL C/C++ (and Apple's Metal)
- Microsoft C++ AMP
- HSA IL

Could this be within the language, rather than ad-hoc extensions?

Embedded C

- Published as a technical report: ISO/IEC TR 18037
- “C - Extensions to support embedded processors”
- For microcontroller based applications with limited resources
- Implemented in the Keil compiler
- Supports *named address spaces*
- C type qualifiers can now include an address space name
- Implementations may provide a set of *intrinsic* address spaces
  - Such names should be reserved; i.e start with `_[A-Z]|__`
- One address space may be a subset of another

Embedded C

- Published as a technical report: ISO/IEC TR 18037
- “C - Extensions to support embedded processors”
- For microcontroller based applications with limited resources
- Implemented in the Keil compiler
- Supports *named address spaces*
- C type qualifiers can now include an address space name
- Implementations may provide a set of *intrinsic* address spaces
  - Such names should be reserved; i.e start with `_[A-Z]|__`
- One address space may be a subset of another

“The most significant constraint is that an address space name cannot be used to qualify an object that has automatic storage duration.”

`http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1275.pdf`

GCC support for Address Spaces

- GNU C supports named address spaces as an extension
- As defined in ISO/IEC DTR 18037 (i.e. Embedded C)
- Support is configured for only particular compile targets
- Adoptive targets include AVR, SPU, M32C, RL78, and x86
- On x86 GCC will compile this as C code with no switches:

```
__seg_fs int g;
__seg_gs int *p;
```

GCC support for Address Spaces

- GNU C supports named address spaces as an extension
- As defined in ISO/IEC DTR 18037 (i.e. Embedded C)
- Support is configured for only particular compile targets
- Adoptive targets include AVR, SPU, M32C, RL78, and x86
- On x86 GCC will compile this as C code with no switches:

```
__seg_fs int g;
__seg_gs int *p;
```

- Internal to GCC, a target may call `c_register_addr_space`
- The SPU port uses the following to declare `__ea` with AS #1

```
#define ADDR_SPACE_EA 1
c_register_addr_space ("__ea", ADDR_SPACE_EA);
```

LLVM support for Address Spaces

- LLVM supports *numbered address spaces*
- The default address space is zero
- Clang syntax builds on the GCC `__attribute__` keyword
- Unlike GCC, Clang supports both C and C++ input languages
- Functionality provides Clang compiler support for OpenCL C
- Similar restrictions apply as with GCC's named address spaces
  - ...though with less documentation

```
#define __seg_fs __attribute__((address_space(1)))
#define __seg_gs __attribute__((address_space(2)))
__seg_fs int g;
__seg_gs int *p;
```

C++ Templates

How about:

```
template <int N>
void foo() {
  __attribute__((address_space(N))) int *p;
}
```

C++ Templates

How about:

```
template <int N>
void foo() {
  __attribute__((address_space(N))) int *p;
}
```

or:

```
template <int N>
struct bar {
  __attribute__((address_space(N))) int *p;
};
```

C++ Templates

How about:

```
template <int N>
void foo() {
  __attribute__((address_space(N))) int *p;
}
```

or:

```
template <int N>
struct bar {
  __attribute__((address_space(N))) int *p;
};
```

- Non-type (integral) template parameters
- To align with SFINAE metaprogramming; or C++ Concepts
- Significantly more expressive...but non-standard

C++ Templates

How about:

```
template <int N>
void foo() {
  __attribute__((address_space(N))) int *p;
}
```

or:

```
template <int N>
struct bar {
  __attribute__((address_space(N))) int *p;
};
```

- Non-type (integral) template parameters
- To align with SFINAE metaprogramming; or C++ Concepts
- Significantly more expressive...but non-standard
- "Embedded C++" is non (ISO) standard, with no templates
- Similar interface consideration within Codeplay's Offload C++

A C++ Address Space Container

A few design options present themselves:

1. A new smart pointer, with expected operator overloads

```
template <int N>
void zod(as_ptr<int,N> as_i) { *as_i = 12345; }
```

2. An extra template parameter to an existing C++ smart pointer
3. Rather than scalars, augment containers; such as `std::vector`
4. C++ containers use `std::allocator`; so, extend here

A C++ Address Space Container

A few design options present themselves:

1. A new smart pointer, with expected operator overloads

```
template <int N>
void zod(as_ptr<int,N> as_i) { *as_i = 12345; }
```

2. An extra template parameter to an existing C++ smart pointer
3. Rather than scalars, augment containers; such as `std::vector`
4. C++ containers use `std::allocator`; so, extend here

Choices, choices, choices...

Ultimately, this is type level information. Use type traits...

A Type Trait API for Address Spaces

- Define an address space trait class template; say `as_trait`
- We need not concern ourselves with the definition
- Akin to C++17 structured bindings' use of `tuple_element`
- `as_trait<int *>::address_space` equals zero
- No address space language extension exposed to the user

```
template <typename T, typename U>
void zot(T p1, U p2) {
  const auto value1 = as_trait<T>::address_space;
  const auto value2 = as_trait<U>::address_space;
  using type1 = typename as_trait<T>::type;
  using type2 = typename as_trait<T>::type;
  static_assert(value1==value2);
  static_assert(std::is_same_v<type1,type2>);
  *p1 = *p2;
}
```

Validation

- The `as_trait` type trait has potential
- A formal proposal based on it could be prepared
- But we would like to validate the system with a real target

Validation

- The `as_trait` type trait has potential
- A formal proposal based on it could be prepared
- But we would like to validate the system with a real target
- A problem comes from Clang:

```
template <int N>
void foo() {
  __attribute__((address_space(N))) int *p;
}
```

```
test.cpp:3:39:  error:  address_space attribute requires an
integer constant
```

Validation

- The `as_trait` type trait has potential
- A formal proposal based on it could be prepared
- But we would like to validate the system with a real target
- A problem comes from Clang:

```
template <int N>
void foo() {
  __attribute__((address_space(N))) int *p;
}
```

```
test.cpp:3:39:  error:  address_space attribute requires an
integer constant
```

- The integer value is dependent on a template parameter
- When the prototype is generated, there is no integer
- Nothing in place to allow later reassessment upon instantiation

LLVM Proposal: `ext_address_space`

- A new type attribute: `ext_address_space`
- A drop in replacement for LLVM's `address_space`
- Accommodates integral non-type template parameters
- As LLVM code for the type attribute: `ext_vector_type`
- ...`ext_address_space` similarly extends `address_space`
    - ...to allow template-dependent **`int`** values to be used

Testing with OpenCL

- OpenCL C uses address spaces and is supported by Clang
- However, OpenCL 2.1 is not yet supported: no templates
- Tobias Zirr (Alpha New) presents a Khronos patched solution
  - The compiler sets the C++ flag when compiling OpenCL C
  - Then passes the output to a Khronos LLVM ↔ SPIR converter
  - With further merges and patches we can now execute the following as SPIRV

OpenCL with C++

```
template <typename T>
T add(T a, T b)
{
  return a+b;
}

__kernel void vec_op(__global const float *,
                     __global const float *,
                     __global const float *)
                       asm("vec_op");

__kernel void vec_op(__global const float *a,
                     __global const float *b,
                     __global const float *c)
{
  int i = get_global_id(0);
  c[i] = add<float>(a[i],b[i]);
}
```

OpenCL with C++

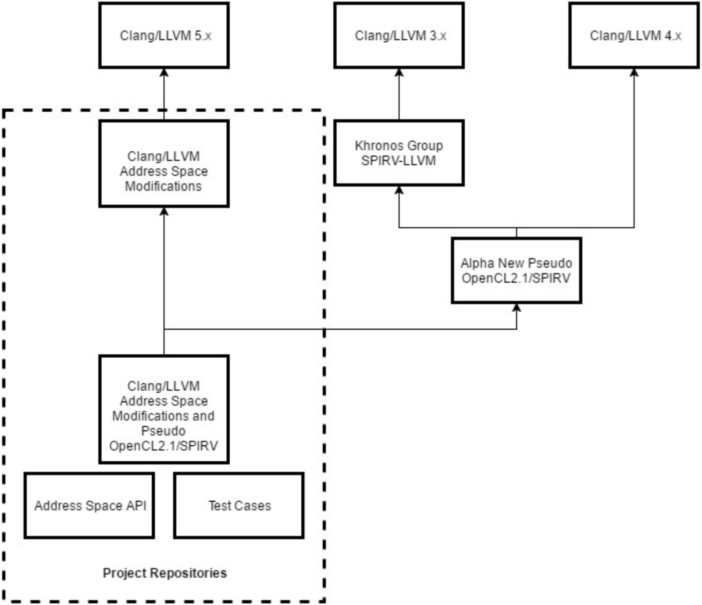
```
template <typename T>
T add(T a, T b)
{
  return a+b;
}

__kernel void vec_op(__global const float *,
                     __global const float *,
                     __global const float *)
                       asm("vec_op");

__kernel void vec_op(__global const float *a,
                     __global const float *b,
                     __global const float *c)
{
  int i = get_global_id(0);
  c[i] = add<float>(a[i],b[i]);
}
```

The **`asm("vec_op")`** prevents the name being mangled, and sets it to “vec_op”

Our Repositories



Clang/LLVM 5.x

Clang/LLVM 3.x

Clang/LLVM 4.x

Clang/LLVM Address Space Modifications

Khronos Group SPIRV-LLVM

Alpha New Pseudo OpenCL2.1/SPIRV

Clang/LLVM Address Space Modifications and Pseudo OpenCL2.1/SPIRV

Address Space API

Test Cases

Project Repositories

Conclusion

- Minimal LLVM compiler modifications to implement and explore dependent address space API design
- Complete the C++ type traits API and test within OpenCL
- Look into integration with the C++ Concepts proposal
- Explore further (SFINAE) template abstractions
- Propose to the BSI ISO-C++ Panel
- Could other (e.g. function) attributes fit within templates?
  - e.g. GCC's `target(arch=`*`ARCH`*`)`
- HPC Clusters? PGAS languages?