Shevlin Park: Implementing C++ AMP with Clang/LLVM and OpenCL

Dillon Sharlet

with Aaron Kunze, Stephen Junkins, Deepti Joshi
Agenda

• **Introduction to Microsoft C++ AMP**
  – Comparison to OpenCL

• **Introduction to Shevlin Park**
  – Architecture overview
    • Clang modifications
    • LLVM modifications
  – Compilation example

• **Performance analysis**
Microsoft C++ AMP Introduction

• **C++ AMP Specification:**
  – Elegant, minimal C++ extensions and template libraries for data parallel programming
  – Released under liberal “Community Promise” license, allowing other implementations

• **C++ AMP Implementation in Visual Studio 2012:**
  – Based on DX11 Compute Shader
  – Decent GPU support, but thus far anemic CPU support (WARP device)
  – Seamlessly integrated into Visual Studio 2012 (compiler, debugger, analyzer)

Our (subjective) experience: Writing data parallel code using C++ AMP is highly productive
Comparing OpenCL & C++ AMP

• Very similar data parallel programming models for device code

  - OpenCL integrates host and device code via driver API
    – Kernel code is separate from host code
    – Full control, host compiler independent, broad platform support

• C++ AMP integrates host and device in the same programming language
  – Simple to program, hides driver API
  – Somewhat similar to CUDA runtime API
  – Currently, only supported on Windows (by DirectX 11)
Comparing OpenCL & C++ AMP: Code Comparison

// OpenCL Host Code:

cl_command_queue q = clCreateCommandQueue(
    m_context, m_devices[0],
    m_queueProperties, &status);
if (CL_SUCCESS != status) return status;

cl_int param1 = 42;
if (CL_SUCCESS != status) return status;

cl_kernel k = clCreateKernel(p, "Simple", &status);
if (CL_SUCCESS != status) return status;

status = clSetKernelArg(k, 0, sizeof(cl_int), &param1);
if (CL_SUCCESS != status) return status;

status = clSetKernelArg(k, 1, sizeof(cl_mem), &buf);
if (CL_SUCCESS != status) return status;

size_t woffset = 0;
size_t gsize = 1024;

status = clEnqueueNDRangeKernel(q, k, 1, &woffset, &gsize, 0, 0, 0, 0);
if (CL_SUCCESS != status) return status;

// C++ AMP Heterogenous Code:

void MatrixMul(float * C, const float * A, const float * B, int M, int N, int L) {
    // Matrix dimensions.
    extent<2> exA(M, N), exB(N, L), exC(M, L);

    // Adapt host data to C++ AMP access.
    array_view<const float, 2> a(exA, A);
    array_view<const float, 2> b(exB, B);
    array_view<float, 2> c(exC, C);

    // Compute sum(row*column) over whole matrix.
    parallel_for_each(c.extent,
        [=](index<2> idx) restrict(amp) {
            float sum = 0.0f;
            for(int k = 0; k < N; ++k)
                sum += a(idx[0], k) * b(k, idx[1]);
            c[idx] = sum;
        });
}

C++AMP: Unified Host & Device C++ Source Compilation
Comparing OpenCL & C++ AMP: Device Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>OpenCL</th>
<th>C++AMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration / Instance semantics</td>
<td>__kernel (Local size $L_2 \times L_1 \times L_0$)</td>
<td>parallel_for_each&lt;$L_0, L_1, L_2$&gt;</td>
</tr>
<tr>
<td></td>
<td>Global size $G_2 \times G_1 \times G_0$</td>
<td>Global extent $G_0 \times G_1 \times G_2$</td>
</tr>
<tr>
<td></td>
<td>$\frac{G_2}{L_2} \times \frac{G_1}{L_1} \times \frac{G_0}{L_0}$ workgroups</td>
<td>$\frac{G_0}{L_0} \times \frac{G_1}{L_1} \times \frac{G_2}{L_2}$ tiles</td>
</tr>
<tr>
<td></td>
<td>Workgroup</td>
<td>Tile</td>
</tr>
<tr>
<td></td>
<td>$L_2 \times L_1 \times L_0$ workitems</td>
<td>$L_0 \times L_1 \times L_2$ threads</td>
</tr>
<tr>
<td></td>
<td>__local memory, barriers, memory fences</td>
<td>tile_static memory, barriers, memory fences</td>
</tr>
<tr>
<td>Buffers</td>
<td>1D buffers</td>
<td>1D, 2D, 3D array&lt;&gt; and array_view&lt;&gt;</td>
</tr>
<tr>
<td></td>
<td>Allows device access to pre-allocated system memory</td>
<td>Allows device access to pre-allocated system memory</td>
</tr>
<tr>
<td></td>
<td><strong>Buffers migrate where needed</strong></td>
<td>(implementation doesn’t use this, DX limitation)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instance of array&lt;&gt; specified to live on device or CPU</td>
</tr>
<tr>
<td>Texture, surface, sampler support</td>
<td>Formats: {char, short, int, half, float, double} $\times$ {1, 2, 3, 4}</td>
<td>Formats: {char, short, int, half, float, double} $\times$ {1, 2, 3, 4}</td>
</tr>
<tr>
<td></td>
<td>Defines mapping from channels to colors</td>
<td>Channels are ordered as they are in memory.</td>
</tr>
<tr>
<td></td>
<td><strong>Defines image sampler operations</strong></td>
<td>Lacks sampler parameterization (interpolation, addressing modes).</td>
</tr>
<tr>
<td>Short vec data types</td>
<td><strong>char, uchar, short, int, uint, long, ulong, float, double</strong> $\times$ {2, 3, 4, 8, 16}</td>
<td>{int, uint, float, double, norm, unorm} $\times$ {2, 3, 4}</td>
</tr>
<tr>
<td>Atomics</td>
<td>atomic_*</td>
<td>concurrency::atomic_*</td>
</tr>
<tr>
<td>Math primitives</td>
<td>Explicit control via native_* functions</td>
<td>fast_math, precise_math namespaces</td>
</tr>
<tr>
<td></td>
<td><strong>Appears to be a superset of C++ AMP</strong></td>
<td>Lacks half_* , minmag/maxmag, vector built-ins</td>
</tr>
</tbody>
</table>

Device Exec Semantics & Compute Primitives: *Mostly Equivalent*
Shevlin Park Introduction and Architecture

- **Shevlin Park**: Implementation of Microsoft C++ AMP specification
  - Built on Clang/LLVM, targets OpenCL for GPU & CPU
- Includes a modified Clang/LLVM compiler, AMP template library, and a small runtime library

AMP implementation not tied to Visual Studio or DirectX!
Shevlin Park C++ AMP Compiler

- **Clang modifications** for C++ AMP language extensions
- **LLVM modifications** for OpenCL codegen and runtime library interface support
- **Details ahead...**

Source to source translation (via LLVM IR) of C++AMP device code to OpenCL C
C++ AMP restrict Keyword Overview

- Language restrictions imposed by restrict qualifiers are a contract between the compiler and developer
- restrict(amp) designed to enable compiler to generate code for alternative devices, such as Direct3D or OpenCL devices
- Examples:

```c++
// Function call restrictions
int f() /*restrict(cpu*/) { return 0; }

// Error: no ‘f’ with restrict(amp)
int g() restrict(amp) { return f(); }

// Overloading and overload resolution
int f() restrict(amp) { return 1; }
int f() restrict(cpu) { return 0; }
int g() restrict(amp, cpu) { return f(); }

// Restriction examples
void f() restrict(amp)
{
    // short not allowed
    short x;
    jmp: // Static variables not allowed
    static int y;
    // Goto not allowed
    goto jmp;
}
```
Motivation Behind restrict Keyword Implementation

- Expressions that result in overload resolution must be evaluated once per restriction of the calling context
  - As far as I know, this cannot be represented with existing AST features
- First approach: clone FunctionDecls that are marked with more than one restrict qualifier
  - Perform semantic analysis for each of the cloned functions
  - Difficult to clone AST nodes correctly...
Shevlin Park Frontend: Clang Modifications

• Added parser support for restrict syntax
  – Attaches RestrictAttr (Attr derived class) to FunctionDecl instances
  – This implementation is temporary, to be replaced with restrict qualifiers implemented as part of function types

• Modified AST to support restrict keyword
  – Added RestrictOverloadExpr to support constructing expressions per restrict context
    • Stores an expression per restrict context
  – Added restrict information to DeclRefExpr, MemberExpr
  – Added support for RestrictOverloadExpr to constant expression evaluators
  – (Example on next slide)
RestrictOverloadExpr Example

```c
int f() __attribute__((restrict(0))) __attribute__((restrict(1))) (CompoundStmt 0xbb2650
  (ReturnStmt 0xbb2640 <col:30, col:39> 
    (RestrictOverloadExpr 0xbb2630 <col:37, col:39> 'int'
      (CallExpr 0xbb25b8 <col:37, col:39> 'int'
        (ImplicitCastExpr 0xbb25a8 <col:37> 'int (*)(void)' <FunctionToPointerDecay>
          (DeclRefExpr 0xb574 'int (void)' lvalue Function 0xbb21f0 'f' 'int (void)' restrict(amp))))
    (CallExpr 0xbb2600 <col:37, col:39> 'int'
      (ImplicitCastExpr 0xbb25f0 <col:37> 'int (*)(void)' <FunctionToPointerDecay>
        (DeclRefExpr 0xb5d4 'int (void)' lvalue Function 0xbb2320 'f' 'int (void)' restrict(cpu)))))))

// Original source
int f() restrict(amp) { return 1; }
int f() restrict(cpu) { return 0; }
int g() restrict(amp, cpu) { return f(); }
```
Clang Modifications, cont’d

- **Modified semantic analysis**
  - Added diagnostics for C++ AMP restrictions
  - Overload resolution performed for each calling restriction context, results stored in `RestrictOverloadExpr`

- **Modified code generation**
  - Modified `GlobalDecl` to refer to a declaration and a restrict context
  - Functions are emitted once per restrict context (once per `RestrictAttr`)
  - For functions being emitted for `restrict(amp)`:
    - Modify name mangling to append ‘__AMP’
    - Add LLVM metadata to identify `restrict(amp)` functions
    - Set linkage to internal
  - Added visitors for `RestrictOverloadExpr` to select expression based on restrict context of current `GlobalDecl`

- **Added miscellaneous support** (`StmtDumper`, `StmtPrinter`, `Serialization`, etc.)
Diagnostics examples

// Original source
void f() restrict(amp);
void g() restrict(cpu) { f(); }

AMP.cpp:3:26: error: no matching function for call to 'f'
void g() restrict(cpu) { f(); }
    ^
AMP.cpp:1:6: note: candidate function not viable: incompatible restriction specifiers
void f() restrict(amp);
    ^
1 error generated.

// Original source
struct A { char x; };
void f() restrict(amp)
{
    A a;
}

AMP.cpp:8:2: error: type 'A' is not amp compatible
    A a;
    ^
AMP.cpp:3:7: note: see declaration of member 'x'
    char x;
    ~~~~~^~
AMP.cpp:1:8: note: see declaration of type 'A'
struct A
~~~~~~~
1 error generated.
Shevlin Park Backend: LLVM Modifications

• Modifications contained entirely in new LLVM pass
  – Generates OpenCL C for functions identified as amp-restricted by LLVM metadata
  – Generates runtime functions for accessing metadata about OpenCL C code

• Kernels in C++ AMP are lambdas (functors)
  – Functor member variables are generated as kernel arguments in OpenCL C
  – Lambda parameter (the index) is generated in a bit of kernel prolog code

• OpenCL C kernel metadata is identified by member function pointers (to the function operator)
  – Kernel name
  – Number of arguments
  – Argument pointers/sizes
LLVM Modifications, cont’d

- Structured control flow generated from LLVM’s unstructured control flow
  - Not ideal! It would be easier to generate OpenCL C from the AST...
  - ... but we are hoping to intercept SPIR, which would be an appropriate target

- Address space analysis performed on LLVM IR
  - __global pointers in array/array_views
  - __local pointers for tile_static declarations
  - Address spaces are then propagated through instructions so correct OpenCL C can be generated
    - This is not always possible to do statically...
Shevlin Park Runtime

- **Shevlin Park runtime is a combination of C++ templates and a small library**
  - AMP header library implements the AMP user API
    - array\langle T,N\rangle, array_view\langle T,N\rangle, etc. from the specification
    - Math, atomics, and other restrict(amp) built-ins
    - parallel_for_each, creates and compiles kernels using internal interface exposed by...
  - ...thin library to interface with OpenCL runtime
  - OpenCL executes compiled kernels

Shevlin Park compiles C++ AMP programs to programs with OpenCL kernels usable with a standard OpenCL implementation
Shevlin Park Compilation Example

```cpp
const int N = 32;

int data[N];
array_view < int, 1 > data_av(N, data);
parallel_for_each(data_av.extent,
    [] (index<1> i) restrict(amp)
    {
        data_av[i] = data_av[i] + 7;
    });
data_av.synchronize();
```
Shevlin Park Compilation Example

// From amp.h:

// Despite not being marked static, these functions
// have internal linkage.
extern "C"
{
    // OpenCL code for this translation unit.
    const char *__amp_program();

    // Kernel name for a pointer to a function in this
    // translation unit.
    const char *__amp_kernel_name(const void *f);

    // Kernel argument info.
    int __amp_kernel_arg_count(const void *f);
    const void *__amp_kernel_arg_ptr(const void *f,
                                      const void *this,
                                      int arg);
    int __amp_kernel_arg_size(const void *f, int arg);
    void *__amp_kernel_arg_sync(const void *f, const
                                  void *this, int arg);
}

Shevlin Park's implementation of parallel_for_each only relies on (_amp_*) intrinsics to access OpenCL kernel code.
Shevlin Park Compilation Example

Clang generates same code as for any C++ program. Keyword “restrict(amp)” just adds LLVM metadata.
AMP To OpenCL pass translates restrict(amp) code to OpenCL kernels and defines the intrinsics for use by the runtime library.
Shevlin Park Compilation Example

OpenCL C:

```c
__kernel void k_lambda_body( i32 __global*arg1,
                           i32 arg2)
{
    // Kernel prolog.
    _class_concurrency__array_view arg0;
    arg0._0._0._0._0 = arg1;
    arg0._0._0._1._0._0[0] = arg2;
    _class_concurrency__index __index;
    __index._0._0[0] = get_global_id(0);
    _class_concurrency__index *i = &__index;

    // _entry:
    i32 *__arrayidx_i_i_i = &i[0]._0._0[0];
    i32 _1 = *__arrayidx_i_i_i;
    i32 __global**__ptr_i = &arg0._0._0._0._0;
    i32 __global*_2 = *__ptr_i;
    i32 __global*_arrayidx_i = &_2[_1];
    i32 _3 = *__arrayidx_i;
    i32 _add = _3 + 7;
    *__arrayidx_i = _add;
    return;
}
```

Generated code is standard OpenCL C, to be compiled at run time.
### Shevlin Park Runtime Example

**C++ AMP Source**

```c++
int data1[N], data2[N];
array_view<int, 1> av1(N, data1);
array_view<int, 1> av2(N, data2);
parallel_for_each(data_av.extent, [] (index<1> i) restrict(amp)
{ av1[i] = av2[i] + 7; });
parallel_for_each(data_av.extent, [] (index<1> i) restrict(amp)
{ av1[i] = av2[i] + 7; });
av1.synchronize();
// av1 destructor
```

**Generated OpenCL Runtime Calls**

- `clGetPlatformIDs (per process)`
- `clGetDeviceIDs (per process)`
- `clCreateContext (per process)`
- `clCreateBuffer`
- `clCreateBuffer`
- `clCreateCommandQueue (per accelerator_view)`
- `clCreateProgramWithSource (per trans. unit)`
- `clBuildProgram (per translation unit)`
- `clCreateKernel`
- `clSetKernelArg (per captured variable)`
- `clEnqueueNDRangeKernel`
- `clCreateKernel`
- `clSetKernelArg (per captured variable)`
- `clEnqueueNDRangeKernel`
- `clEnqueueMapBuffer`
- `clEnqueueUnmapMemObject`

Shevlin Park’s C++ AMP implementation maps well to OpenCL runtime API calls.
Performance Analysis Overview*

Workloads & Workload Porting:

• 4 Workloads: SGEMM, Convolution, Histogram, FFT
• Ports are generally “naive ports” parallelized with few hours porting effort
• No ninja workload tuning to the programming model
• No ninja OpenCL optimization, no ninja C++AMP employed

Benchmarking Machine:

• IVB CPU @2.6 GHz
• IVB HD Graphics 4000 @650-1250 MHz (turbo)

Software Environment & Tools:

• Windows 7 64 bit (32-bit exes)
• Windows 8 64 bit (WARP device)
• IVB HD Graphics 4000 Driver 15.28.7
• Visual Studio 2012
• Intel® SDK for OpenCL applications

* Software and workloads used in performance tests described in this presentation may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.
Performance Data (GPU)

- SGEMM performance improved due to constant folding (and then loop unrolling) not available in OpenCL C
- Small (~10%) penalty for OpenCL source → source translation in most cases
- VC++ 2012 C++ AMP exhibits DX11 buffer model memory transfer cost
  - OpenCL and Shevlin Park utilize OpenCL shared memory buffers (CL_MEM_USE_HOST_PTR)
  - Computation ‘dense’ workloads (SGEMM, Convolution) affected less
Performance Data (CPU)

- SGEMM performance improved due to constant folding (and then loop unrolling) not available in OpenCL C
- Penalty for OpenCL source → source translation is large in some cases
- For these workloads, the DX11 WARP device (CPU DX11 device implementation) used by VS2012 C++ AMP appears less optimized than OpenCL
Performance Observations and Analysis

- **C++ AMP** enables ‘whole platform optimizations’ not necessarily available to OpenCL
  - SGEMM performance improved due to constant folding across host - device boundary (and then loop unrolling)
  - Other optimizations could be possible with more compiler education of parallel_for_each
- **Small (<~10%) penalty** for OpenCL source → source translation in most cases
  - But, significant in CPU FFT (~100% penalty)
  - SPIR would likely help by avoiding extra round trip through OpenCL C
Conclusions

• C++ AMP can be successfully implemented in Clang/LLVM, with OpenCL as a runtime
  – So, C++ AMP could be cross platform!

• We are really looking forward to SPIR for OpenCL
  – SPIR will greatly expand the useful domain of OpenCL to machine tools
    • Compilers for new languages
    • Even heterogeneous ones like C++ AMP!
    – Specifically for us, SPIR will greatly improve the quality of our C++ AMP implementation

• Clang/LLVM are fantastic tools for making prototype compilers accessible projects
  – Thank you!!
Questions?
**C++AMP restrict Restrictions**

- The function can call only functions that have the restrict(direct3d) clause.
- The function must be inlinable.
- The function can declare only int, unsigned int, float, and double variables, and classes and structures that contain only these types.
- Lambda functions cannot capture by reference and cannot capture pointers.
- References and single-indirection pointers are supported only as local variables and function arguments.

- Recursion.
- Variables declared with the `volatile` keyword.
- Virtual functions.
- Pointers to functions.
- Pointers to member functions.
- Pointers in structures.
- Pointers to pointers.
- goto statements.
- Labeled statements.
- try, catch, or throw statements.
- Global variables.
- Static variables. Use `tile_static` Keyword instead.
- `dynamic_cast` casts.
- The typeid operator.
- `asm` declarations.
- Varargs.

- These restrictions are similar to other GPGPU programming languages.
- Microsoft expects the restrictions to ease over time.
C++11 Lambda Example

```cpp
const int N = 10;
int a[N] = {1, }, b[N] = {2, }, c[N];

// Lambda computing c[i] = a[i] + b[i]. Could later be invoked via sum object.
auto sum = [&] (int i) restrict(cpu)
{
    if(i < N)
        c[i] = a[i] + b[i];
};
```

- **auto sum**: This is a declaration of a lambda object.
- **[&, N]**: This is the capture expression. This expression says:
  - By default, capture by reference. Any variables used in the body not explicitly captured are captured by this semantic.
  - Capture N by value.
- **(int i)**: This is the argument list, accepting one integer i.
- **restrict(cpu)**: This is the modifier declaration.
- **{...}**: The body of the lambda, using captured variables and arguments.
```cpp
#include <amp.h>
#include <iostream>
using namespace concurrency;

const int size = 5;

void CppAmpMethod() {
    int aCPP[] = {1, 2, 3, 4, 5};
    int bCPP[] = {6, 7, 8, 9, 10};
    int sumCPP[size];

    // Create C++ AMP objects.
    array_view<const int, 1> a(size, aCPP);
    array_view<const int, 1> b(size, bCPP);
    array_view<int, 1> sum(size, sumCPP);
    sum.discard_data();

    parallel_for_each(
        // Define the compute domain, which is the set
        // of threads that are created.
        sum.extent,
        // Define the code to run on each thread on the
        // accelerator.
        [=](index<1> idx) restrict(amp) {
            sum[idx] = a[idx] + b[idx];
        }
    );

    // Print the results. The expected output is "7, 9,
    // 11, 13, 15".
    for (int i = 0; i < size; i++) {
        std::cout << sum[i] << "\n";
    }
}
```
Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804