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 (C

cl_in
if (C
cl_ke
if (C

statu
if (C
statu
if (C

size_
size_

statu
    &
if (C
```

```
// OpenCL Device Code:

__kernel void Naive(__global float * C,
    __global const float * A, __global
    const float * B, int N)
{
    // Get matrix dimensions.
    const int L = get_global_size(0);
    const int M = get_global_size(1);

    // Get column/row.
    int j = get_global_id(0);
    int i = get_global_id(1);

    // Compute sum(row*column) over whole matrix.
    float sum = 0.0f;
    for(int k = 0; k < N; ++k)
        sum += A[i * N + k] * B[k * L + j];

}
```

```
// 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

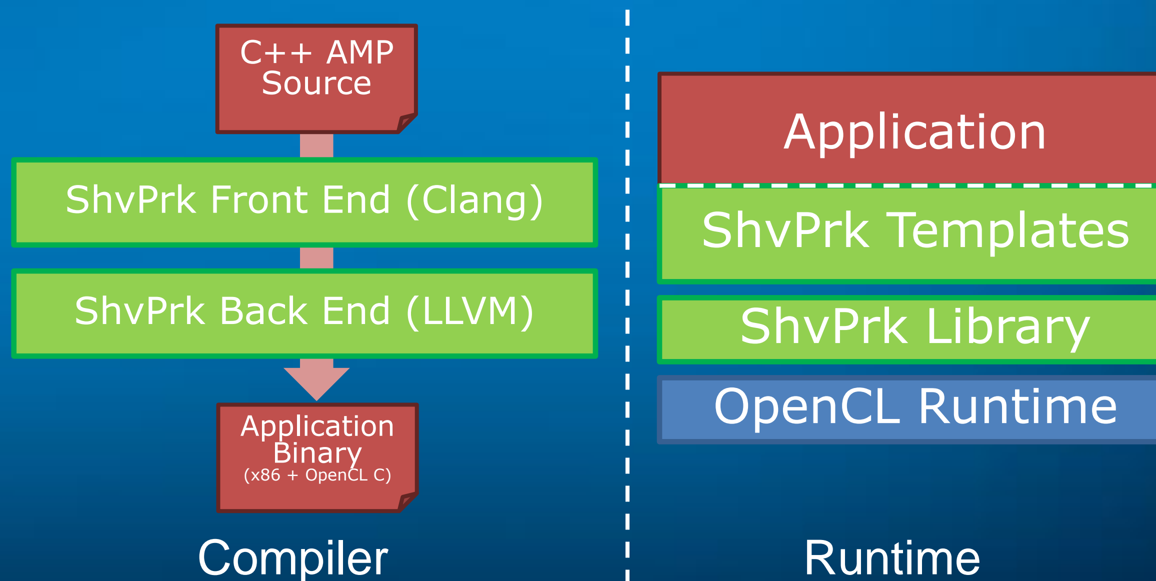
Comparing OpenCL & C++ AMP: Device Features

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

Device Exec Semantics & Compute Primitives: *Mostly* Equivalent

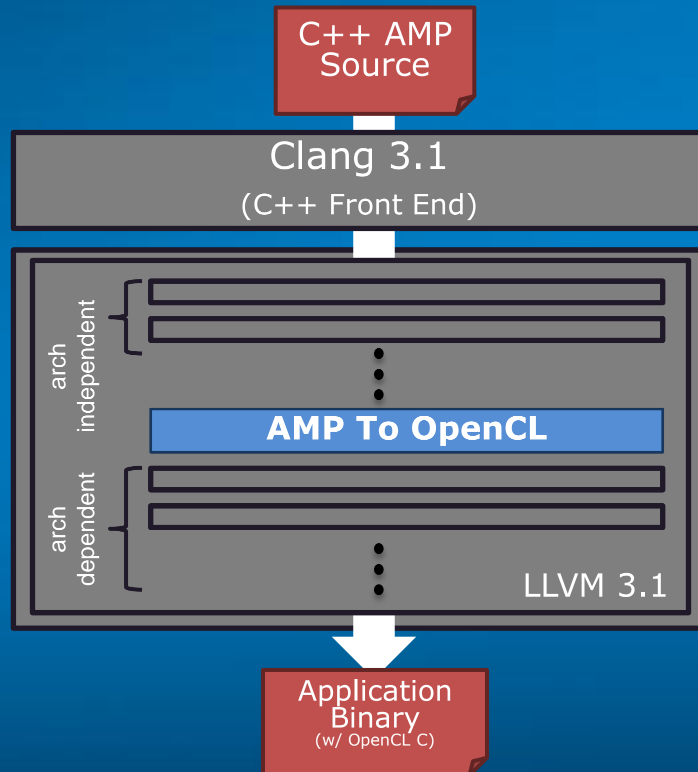
Shevlin Park Introduction and Architecture

- **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:

```
// 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;
}
```


Software and Services Group

Optimization Notice

intel Software

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

```
// Original source
int f() restrict(amp) { return 1; }
int f() restrict(cpu) { return 0; }

int g() restrict(amp, cpu) { return f(); }
```

```
int f() __attribute__((restrict(0))) (CompoundStmt 0xbb22c8
  (ReturnStmt 0xbb22b8 <col:25, col:32>
    (IntegerLiteral 0xbb2298 <col:32> 'int' 1)))


int f() __attribute__((restrict(1))) (CompoundStmt 0xbb23f8
  (ReturnStmt 0xbb23e8 <col:25, col:32>
    (IntegerLiteral 0xbb23c8 <col:32> 'int' 0)))


int g() __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))))))
```


Software and Services Group

Optimization Notice

intel Software

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.)

Optimization Notice
intel Software

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

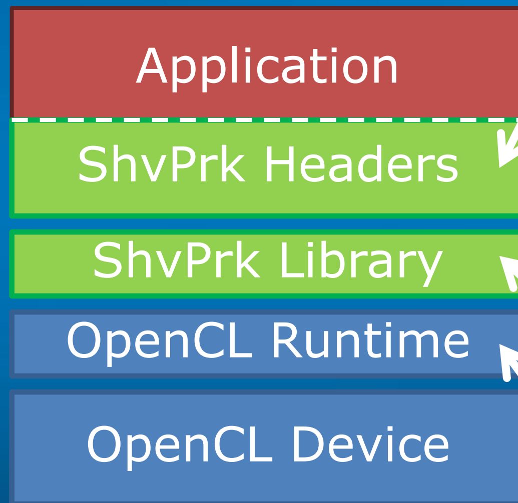
- Modifications contained entirely in new LLVM pass
  - Generates OpenCL C for functions identified as amp-restricted by LLVM metadata
  - Generates runtime functions for accessing metadata about OpenCL C code
- Kernels in C++ AMP are lambdas (functors)
  - Functor member variables are generated as kernel 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

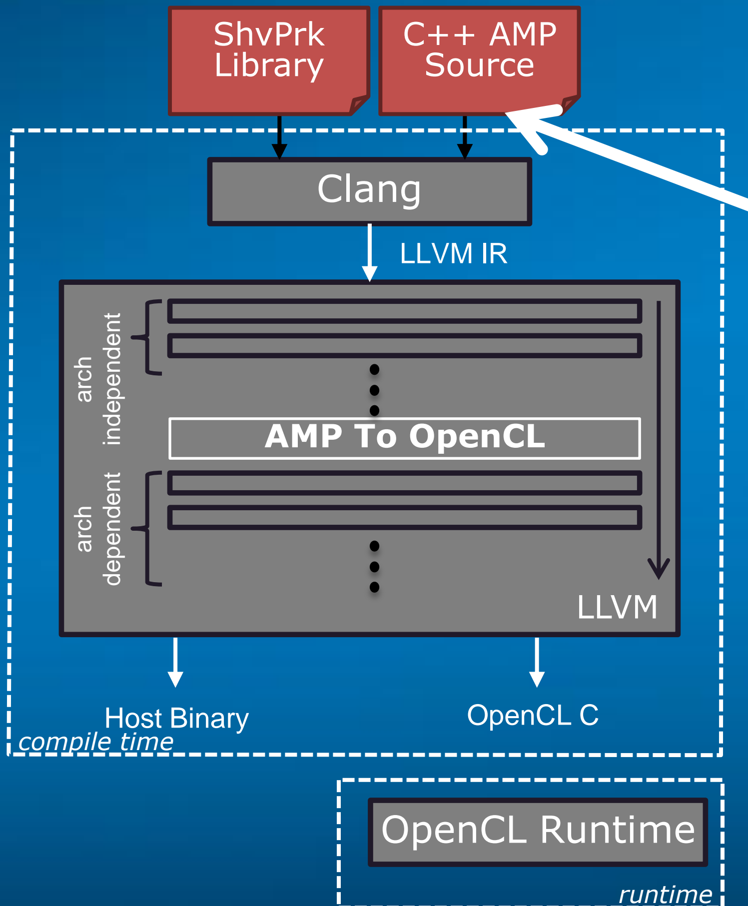
- Shevlin Park runtime is a combination of C++ templates and a small library



- AMP header library implements the AMP user API
  - array<T,N>, array_view<T,N>, 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

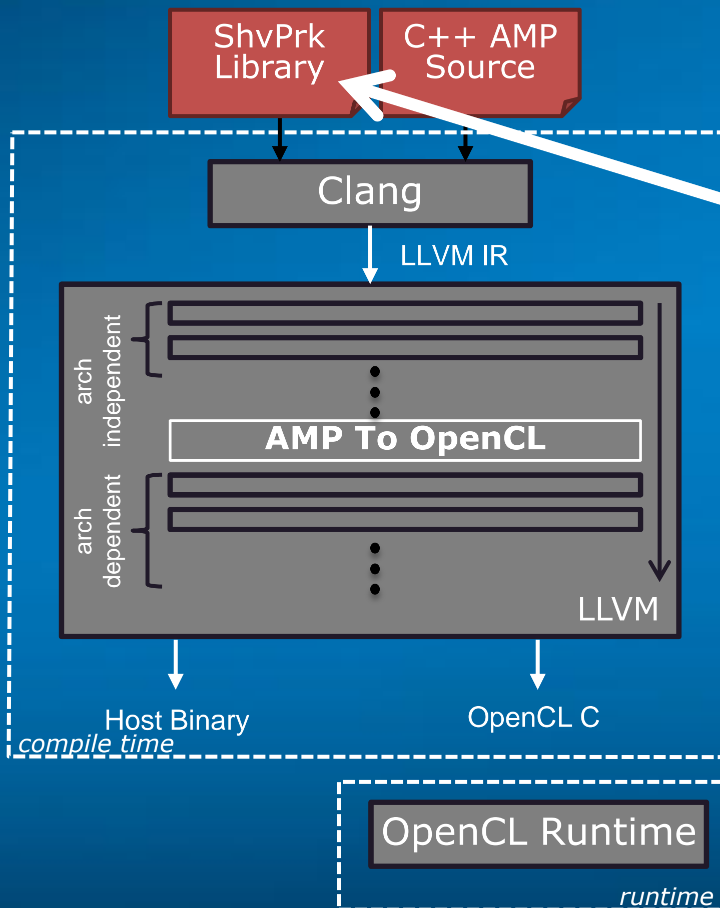
Shevlin Park compiles C++ AMP programs to programs with OpenCL kernels usable with a standard OpenCL implementation

Shevlin Park Compilation Example




```
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



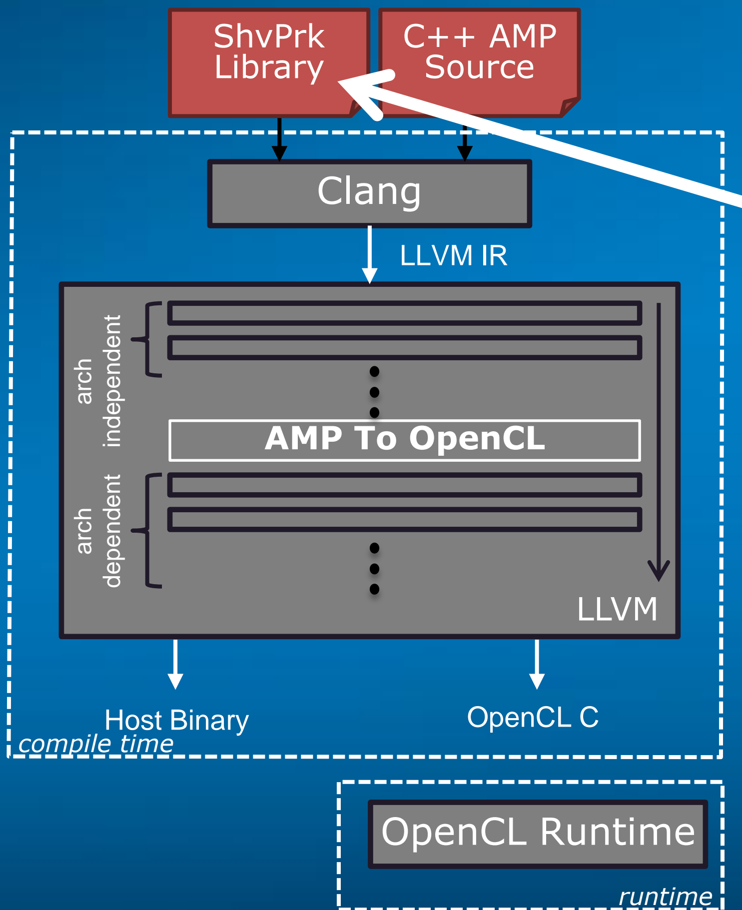

```
// From amp.h:

// 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);
}
```


Software and Services Group

Optimization Notice

intel Software

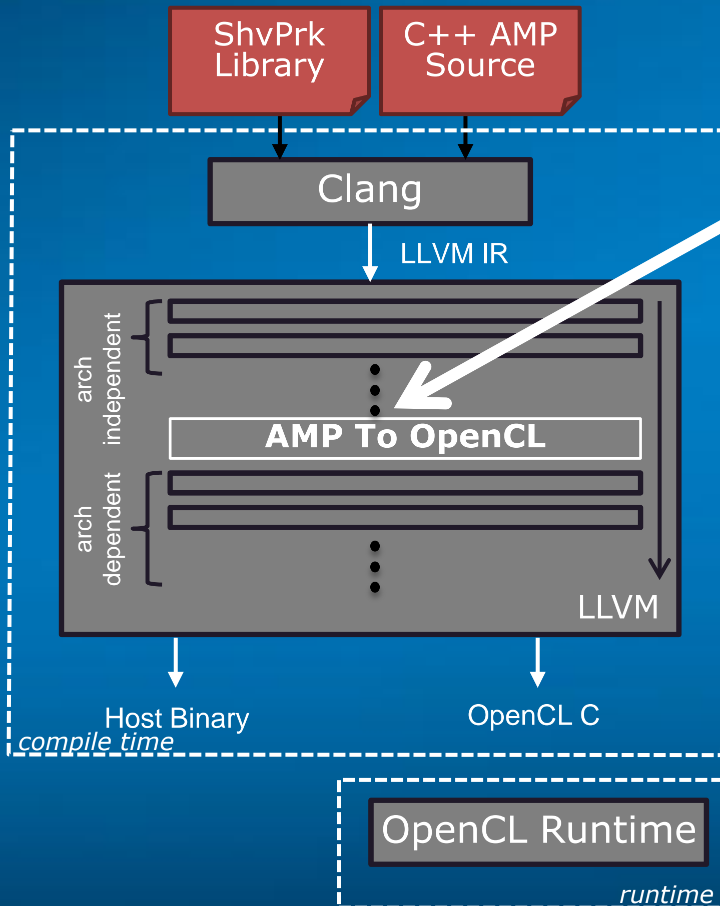
Shevlin Park Compilation Example




```
// From amp.h:

template < int N >
static void parallel_for_each(
    const extent < N > & compute_domain,
    const Kernel < index < N > > & f)
{
    // Compile kernel code.
    cl_kernel k = CompileKernel(_amp_program(),
                                _amp_kernel_name(f),
                                device);

    // Set kernel arguments.
    int args = _amp_kernel_arg_count(fn);
    for(int i = 0; i < args; ++i)
    {
        const void * ptr = _amp_kernel_arg_ptr(…);
        int size = _amp_kernel_arg_size(…);
        SetKernelArg(k, i, ptr, size);
    }

    // Enqueue kernel object.
    EnqueueKernel(queue, k, compute_domain);
}
```

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

Shevlin Park Compilation Example

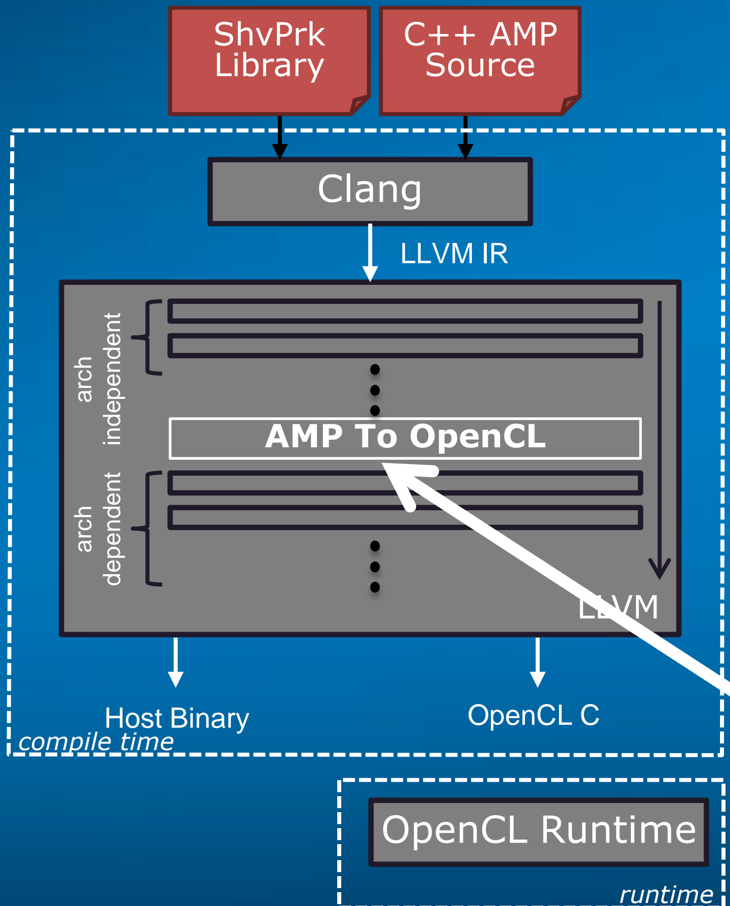


Before AMP-to-OpenCL pass:

```
define void @lambda_body(
    %struct.lambda_body_struct* nocapture %this,
    %"class.concurrency::index"* nocapture %i)
{
  …
  %arrayidx.i = getelementptr addrspace(1)* %1, i32 %0
  %2 = load i32 addrspace(1)* %arrayidx.i
  %add = add nsw i32 %2, 7
  store i32 %add, i32 addrspace(1)* %arrayidx.i, align
4, !tbaa !4
  ret void
}

!amp.restrict = !{!0}

!0 = metadata !{void (%struct.lambda_body_struct*,
%"class.concurrency::index"*)* @lambda_body}

declare i8* @_shvprk_program()…
```

Clang generates same code as for any C++ program. Keyword "restrict(amp)" just adds LLVM metadata.

Shevlin Park Compilation Example

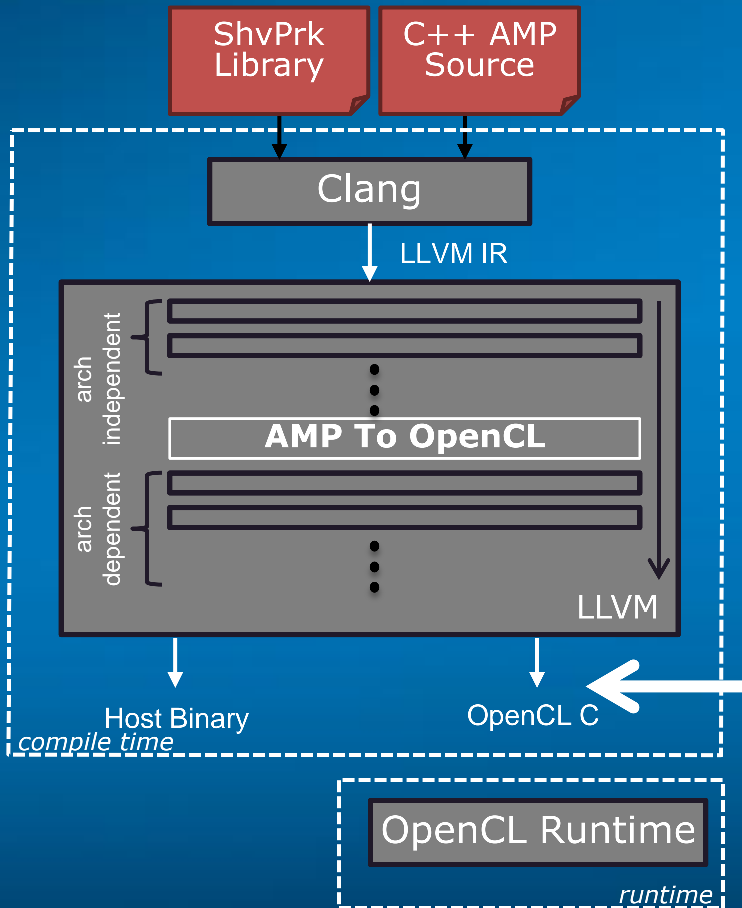


After AMP-to-OpenCL pass:

```
declare void @lambda_body(
    %struct.lambda_body_struct* nocapture %this,
    %"class.concurrency::index"* nocapture %i)

@__shvprk_program = internal constant [3666 x i8]
    c"// OpenCL C Code for the lambda (next slide)... "

define internal i8* @_shvprk_program()
{
    ret i8* getelementptr inbounds (
        [3666 x i8]* @__shvprk_program, i32 0, i32 0)
}

define internal i8* @_shvprk_kernel_name(i8*) {...}
define internal i32 @_shvprk_kernel_arg_count(i8*)
{...}
define internal i8* @_shvprk_kernel_arg_ptr(i8*, i8*,
i32) {...}
define internal i32 @_shvprk_kernel_arg_size(i8*, i32)
{...}
```

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:

```
__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._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

Optimization Notice

intel Software

Shevlin Park Runtime Example

C++ AMP Source	Generated OpenCL Runtime Calls
`int data1[N], data2[N];`	
`array_view < int, 1 > av1(N, data1);`	`clGetPlatformIDs` *(per process)* `clGetDeviceIDs` *(per process)* `clCreateContext` *(per process)* `clCreateBuffer`
`array_view < int, 1 > av2(N, data2);`	`clCreateBuffer`

`parallel_for_each(data_av.extent,`
`[] (index<1> i) restrict(amp)`
`{`
`av1[i] = av2[i] + 7;`
`});`
	`clCreateCommandQueue` *(per accelerator_view)*
	`clCreateProgramWithSource` *(per trans. unit)*
	`clBuildProgram` *(per translation unit)*
	`clCreateKernel`
	`clSetKernelArg` *(per captured variable)*
	`clEnqueueNDRangeKernel`

`parallel_for_each(data_av.extent,`
`[] (index<1> i) restrict(amp)`
`{ av1[i] = av2[i] + 7; });`
	`clCreateKernel`
	`clSetKernelArg` *(per captured variable)*
	`clEnqueueNDRangeKernel`

`av1.synchronize();`	`clEnqueueMapBuffer`
`// av1 destructor`	`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:

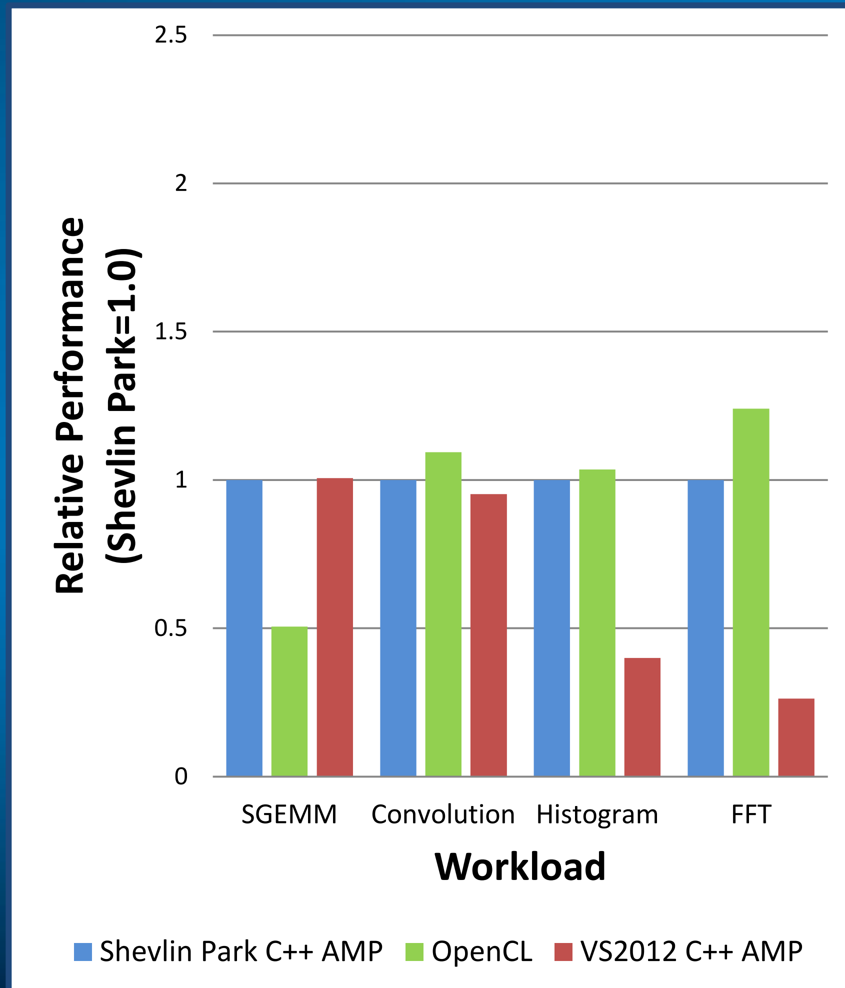
- IVB CPU @2.6 GHz
- IVB HD Graphics 4000 @650-1250 MHz (turbo)

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

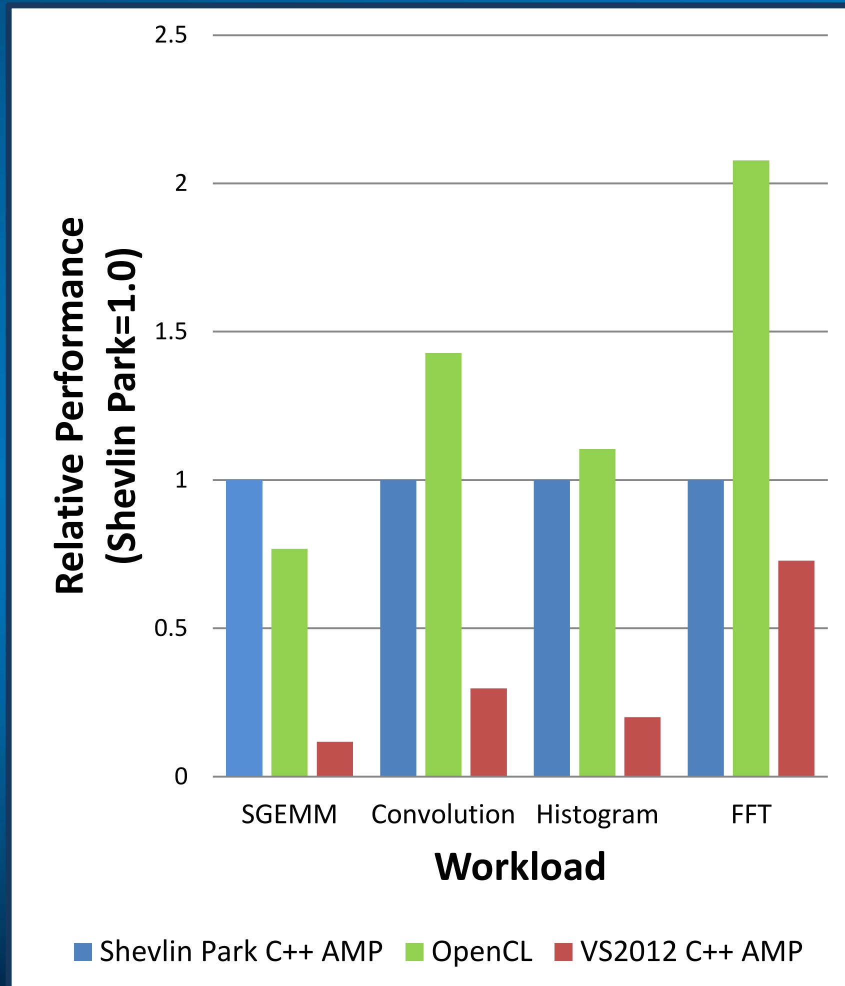
* Software and workloads used in performance tests described in this presentation may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any 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

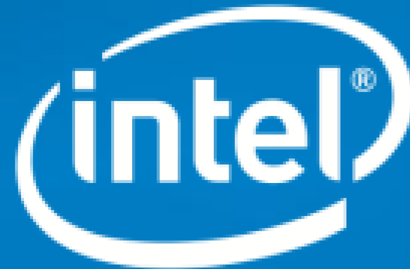
Performance Observations and Analysis

- 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

```
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 = [&, N] (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.

Code Sample

(from: http://msdn.microsoft.com/en-us/library/hh265136.aspx)

```
#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

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

Legal Disclaimer

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Intel may make changes to specifications and product descriptions at any time, without notice.

All products, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice.

Intel, processors, chipsets, and desktop boards may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Code names featured are used internally within Intel to identify products that are in development and not yet publicly announced for release. Customers, licensees and other third parties are not authorized by Intel to use code names in advertising, promotion or marketing of any product or services and any such use of Intel's internal code names is at the sole risk of the user

Software and workloads used in performance tests 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.

Intel, the Intel logo, Intel386, Intel486, IntelDX2, IntelDX4, IntelSX2, Intel Atom, Intel Atom Inside, Intel Core, Intel Inside, Intel Inside logo, are trademarks of Intel Corporation in the U.S. and other countries.

OpenCL and OpenCL logo are trademarks of Apple Inc. used by permission by Khronos

*Other names and brands may be claimed as the property of others.

Copyright © 2010-2012. Intel Corporation.