The Offload C++ Programming System

George Russell,
Compiler Engineer,
Codeplay
Codeplay background

• Compiler company based in Edinburgh, Scotland
• 8 years experience in C/C++ and shader compilers
• Target special-purpose parallel hardware architectures
  - PlayStation®2
  - Ageia PhysX
  - Cell BE, PlayStation®3
  - Multi-core processors
  - x86: SSE, MMX, 3DNow!
• Have developed technology to simplify application deployment on complex & evolving parallel systems
Outline

• Challenge of programming for ‘host and accelerators’
• Offload C++
• Automatic Call Graph Duplication
• Methodology for Offloading C++
• Conclusion & Questions
‘Host and Accelerators’
‘Host and Accelerators’
‘Host and Accelerators’
Porting Software to Multi-core

• Challenges
  – Maintain portability, limit scope of change
  – Hardware limitations
  – Scope for error: less static checking
  – Explicit management of data transfers
  – Time consuming

• Hard to adapt existing concurrent software
Offload C++

- Conservative C++ extension
  - Applicable to *existing* code bases
  - Can `#define` extensions out of code

- Targets heterogeneous cores
  - Host core + accelerator cores
  - Distinct memory spaces

- Programming model
  - Migrate a host thread onto an accelerator
void offloaded(unsigned char* screenbuf) {
    float x_incr = (MAX_X - MIN_X)/(float)gWidth;
    float y_incr = (MAX_Y - MIN_Y)/(float)gHeight;
    __offload (( x_incr, y_incr, screenbuf )); {
        for(int j = 0; j < gHeight; ++j )
            for(int k = 0; k < gWidth; ++k )
                screenbuf[j*gWidth+k] = mand(k, j, x_incr, y_incr);
    }
}.
void offloaded(unsigned char* screenbuf) {
    float x_incr = (MAX_X - MIN_X)/(float)gWidth;
    float y_incr = (MAX_Y - MIN_Y)/(float)gHeight;
    __offload (( x_incr, y_incr, screenbuf )) {
        for(int j = 0; j < gHeight; ++j ) {
            for(int k = 0; k < gWidth; ++k ) {
                screenbuf[j*gWidth+k] = mand(k, j, x_incr, y_incr);
            }
        }
    }
}.
void offloaded(unsigned char* screenbuf) {
    float x_incr = (MAX_X - MIN_X)/(float)gWidth;
    float y_incr = (MAX_Y - MIN_Y)/(float)gHeight;
    
    __offload (( x_incr, y_incr, screenbuf )) {
        for(int j = 0; j < gheight; ++j ) {
            for(int k = 0; k < gWidth; ++k ) {
                screenbuf[j*gWidth+k] = mand(k, j, x_incr, y_incr);
            }
        }
    }
}
void offloaded(unsigned char* screenbuf) {
    float x_incr = (MAX_X - MIN_X)/(float)gWidth;
    float y_incr = (MAX_Y - MIN_Y)/(float)gHeight;
    __offload (( x_incr, y_incr, screenbuf )) {
        for(int j = 0; j < gHeight; ++j )
            for(int k = 0; k < gWidth; ++k )
                screenbuf[j*gWidth+k] = mand(k, j, x_incr, y_incr);
    }
}
Offload Blocks

```c++
void offloaded(unsigned char* screenbuf) {
    float x_incr = (MAX_X - MIN_X)/(float)gWidth;
    float y_incr = (MAX_Y - MIN_Y)/(float)gHeight;
    __offload (( x_incr, y_incr, screenbuf )) {
        for(int j = 0; j < gHeight; ++j )
            for(int k = 0; k < gWidth; ++k )
                screenbuf[j*gWidth+k] = mand(k, j, x_incr, y_incr);
    }
};
```

Call graph duplication
Automatic Call Graph Duplication

• Compiles for host and accelerator
• Adapts code to handle distinct memory spaces
  – Produces ‘offload’ duplicates to run on accelerator
• Automation
  – Time saving / Enable experimentation
  – Increase the amount of code offloaded
  – Reduce scope of modifications to program code
  – Keep a single version of program source code
Multiple Memory Spaces in C++

- C++ assumes a single memory space
- Not true for ‘hosts and accelerators’
  - May have a multi-level memory hierarchy
- Introduces scope for programmer error
  - Confusion of pointers to different memory spaces
- Interacts badly with C++
  - function pointers / vtables
Offload C++ and Pointers

• Distinct kinds of pointers & references
  - __outer and local pointers (host, accelerator)
  - Enable output of data transfer operations

• Incompatible at the type level
  - int *p; int __outer *q;
  - p = q; // Type error
  - q = p; // Type error
  - *q = *p; // OK; data transfer!
Offload C++ and Pointers (2)

• Passing pointers/references as parameters
  - int f(int &x, int &y) { return a*b; }
  - f(a, b);

• Compiler generates duplicates on demand
  - offload int f(int &x, int &y) { return a*b; }
  - offload int f(int _outer&x, int &y) { return a*b; }
  - offload int f(int &x, int _outer&y) { return a*b; }
  - offload int f(int _outer&x, int _outer&y) { return a*b; }
Type Inference in Functions

- Inference propagates `__outer`
  - across initialisation
  - and casts

- Inference failure leads to compile error

```cpp
void f(int * param) {
    int* local1 = param;
    char* local2 = (char*) param;
    int* local3 = 0;
    ....
    local3 = param;
}
```
Accelerator Specific features

- How to use non-portable features directly?
- In an offload context:
  - inside an offload block, or in a function called directly or indirectly inside an offload block
- Dual C++ dialects
  - host and accelerator
  - Allow accelerator dialect in an offload context
Overloading Call Graph Duplication

• Overload portable functions
  – void f() {...};
  – offload void f() {...};

• Overload selected function duplicates
  – void f(int *p, int *q) {...};
  – offload void f(int *p, int *q) {...};

• offload functions callable in an offload context
Offloading Virtual Methods

• Call graph duplication of late bound calls
  – function pointers / virtual methods
• Limited code space
• Offload block ‘domains’
  – select functions to duplicate for indirect calls
  – Lookup accelerator implementation via host address
Domains Example

```cpp
struct B {
    virtual void g(B*);
    virtual void f();
    virtual void f(int);
};

struct C: B {
    virtual void f();
    virtual void f(int);
    virtual void g(B*);
};

B* ptr = new C;

// both overloads of C::f in the domain.
__offload [C::f] {
    ptr->f();
}.

// offloading C::f(int) only.
__offload [(void(C::*)(int)) &C::f].
{
    ptr->f(0);
}.
```
Domains Example

```c
struct B {
    virtual void g(B*);
    virtual void f();
    virtual void f(int);
};

struct C : B {
    virtual void f();
    virtual void f(int);
    virtual void g(B*);
};

__offload.
    // #1 outer this pointer,
    // outer pointer parameter.
    (void (C::*)(__outer B*)) & C::g,
    // #2 inner this pointer,
    // local pointer parameter.
    (void (C::*)(B*)) & C::g this.
} {
    // virtual call #1, on outer pointer.
    ptr->g(ptr);
    B* inner = new C;
    // virtual call #2, on local pointer.
    inner->g(inner);
};
```
Offloading in Large Codebases

• Duplication of functions across compilation units
  – Extended function attributes

• Calling host only routines
  – Duplication requires source code

• Overlays
  – Support for limited accelerator code memory
Offloading Methodology

1. Get code on the accelerator
2. Tune for performance on a single accelerator
3. Algorithm restructuring and inlining
4. Accelerator specific optimisations
5. Parallelise
Offloading to Accelerators

```c
float x_incr = (MAX_X - MIN_X)/(float)gWidth;
float y_incr = (MAX_Y - MIN_Y)/(float)gHeight;

unsigned int handles[NUM_SPE];

int chunkSize = gHeight/NUM_SPE;
for (int h = 0; h < NUM_SPE; h++) {
    int start = h*chunkSize;
    int end = start + chunkSize;

    handles[h] = __offload(( start, end, x_incr, y_incr, screenbuf ));
    for (int j = start; j < end; ++j)
        for (int k = 0; k < gwidth; ++k)
            screenbuf[j*gWidth+k] = mand(k, j, x_incr, y_incr);

}

for (int h = 0; h < NUM_SPE; h++)
    offloadThreadJoin(handles[h]);
```

- Divide work
- Spawn offload threads
- Host awaits for thread exits
Offload C++ for Cell BE

- PS3® GameOS and Cell Linux
- Optimising Single Source C++ Compiler
  - Interoperable with GCC
  - Altivec®, Cell intrinsics
  - Generates C with Cell intrinsics / data types
  - Translates PPE vmx to SPE simd
Ease of Offloading

- Offloading should be quick, easy
- Applied to a AAA PS3® Game Renderer
  - In two hours
  - ~800 functions
  - ~170KB SPE object code
  - ~45% of host performance on a single SPE
- Plenty of scope for Cell specific optimisations to follow that
Conclusion

• Future work
  – GPU, Other, Compile to OpenCL.

• Offloading can be simple
  – even late bound calls across compilation units
  – complex dynamic data structure processing
  – type checking data transfer code
  – no extensive modifications to code
  – can use accelerator specific code too
Questions?

Offload C++ for Cell

• http://offload.codeplay.com/
Offloading Methodology (1)

- Enclose the code in an offload scope
- Assist compiler if needed to compile
- Add domain entries for late bound calls
- Use syntax extensions in macros
- Compare offload versus host code
- Keeps code portable to other compilers
- Check behaviour / performance vs host
Offloading Methodology (2)

- Reduce offload accesses to host memory
  - default access is via a software cache
  - offload block arguments
  - Make effective use of fast local storage
  - typesafe templates for data access use cases
- Compiler can report outer accesses
Offloading Methodology (3)

• Compiler optimisation
  • e.g. inlining

• Algorithm restructuring
  • portable code may not be good on accelerator
  • accelerator cores specialised
Offloading Methodology (4)

- Accelerator specific optimisations
- Introduce non-portable code
- Needs some expertise
  - SIMDize code
  - Restructure data for efficient access
  - Consider data transfer strategies
    - e.g. double buffering
- Like directly programming the accelerator
Offloading Methodology (5)

• Parallelise for multiple accelerators
• Similar to multi-threading
• If already parallelised, offload the threads
• Target threads for available accelerators
• May be worth parallelising before optimising individual offloads