

Offload C++ - Concepts and Application

From CPU to Cell to GPU

George Russell + innumerable colleagues

Codeplay background

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



Cellfactor game
from Ageia

PEPPER Project Consortium

■ Universities

- University of Vienna (coord.), Austria
- Chalmers University, Sweden
- Karlsruhe Institute of Technology, Germany
- Linköping University, Sweden
- Vienna University of Technology, Austria

■ Research center

- INRIA, France

■ Companies

- Intel, Germany
- Codeplay Software Ltd., UK
- Movidius Ltd. Ireland



CHALMERS



Performance Portability and Programmability for Heterogeneous Many-core Architectures

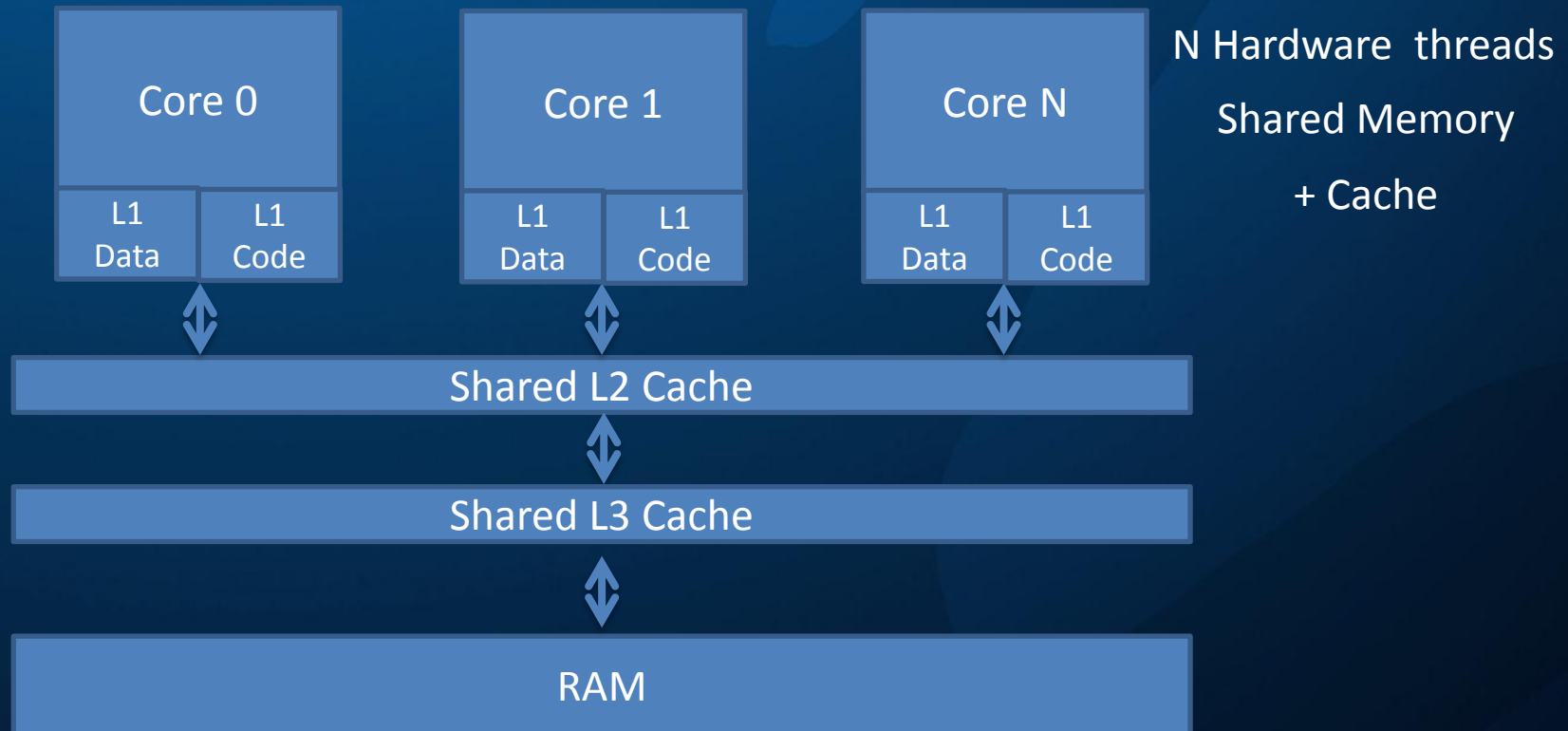
Questions?

- Feel free to ask as they occur!

Overview

- **High Level View of Accelerator Programming**
- Adapting to Accelerators
- Offload C++ Tutorial
 - Some (short) C++ examples
- C++ on GPU
- A Demo of C++ for GPU

Homogenous Multicore Memory Architecture



The Rise of Cell and GPU

- Multicore (2-32 homogenous CPU cores)
 - Shared memory + a coherent HW cache
 - General purpose ISA
 - Direct extension of single core systems
- Cell + GPU
 - Accelerator devices (> 1 in a system)
 - Accelerator cores (SPU / “stream processor”)
 - On chip memories
 - Heterogeneous systems (host + accelerators)

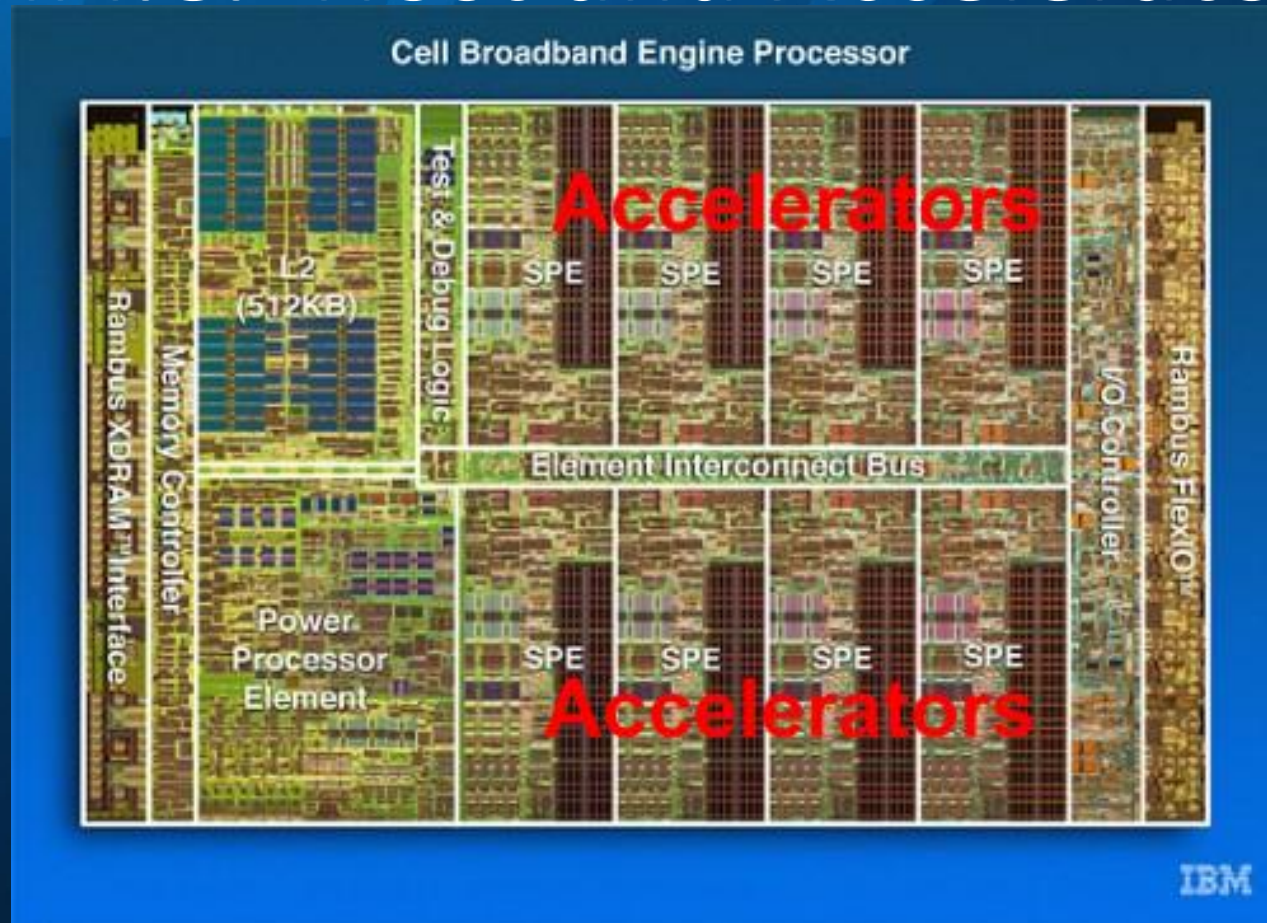
Motivations for Heterogeneity

- The memory wall
 - Excessive contention on shared resource
- Application specialisation
 - Support massive volumes of floating point
 - Data parallelism / SIMD
- Power consumption
 - Fast / slow == Power hungry / frugal
 - On chip == faster + less power usage
- Reprioritize transistor budget / die area
 - Who needs a branch predictor?

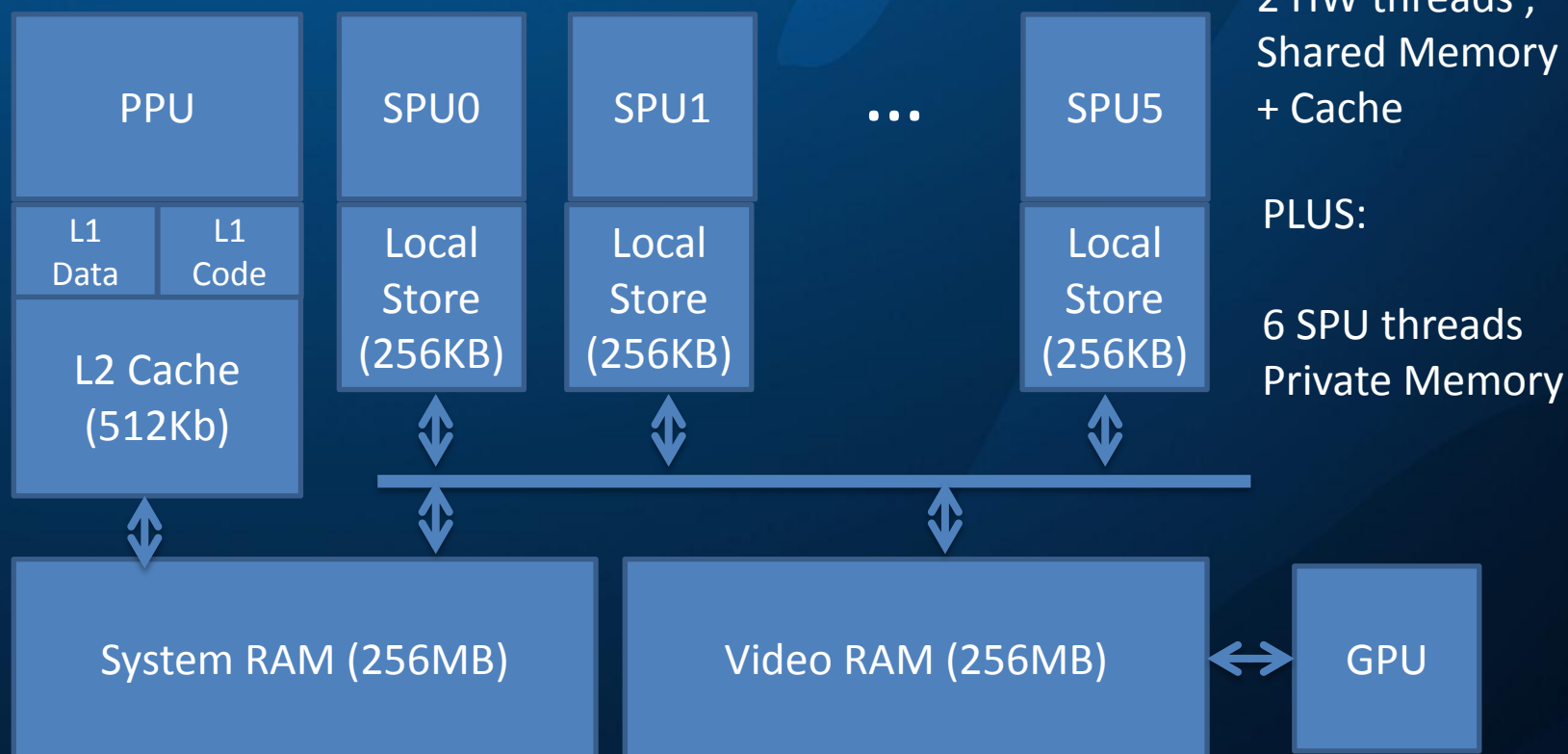
The Cell

- Automatic cross-core cache coherency seems like a waste of good transistors.
 - Jonathan Adamczewski (@twoscomplement, GPU *Compiler developer at Codeplay*)

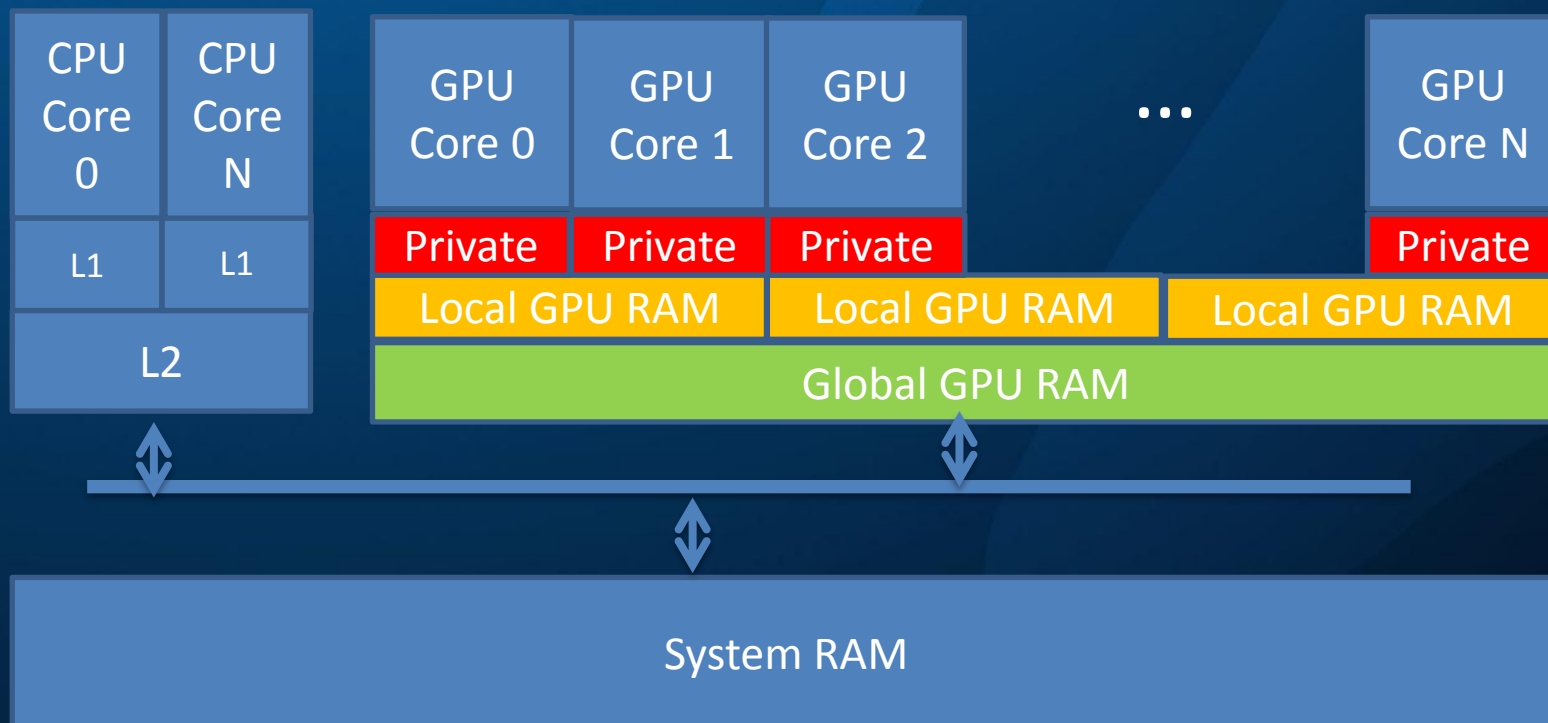
Cell i.e. 'Host and Accelerators'



PS3 Memory Architecture



GPU Memory Architecture



Programming for Multicore

- Now standard practice? (and is this *easy*?)
 - Optimise, then Parallelise
 - Ensure correct w.r.t.
 - Deadlock / livelock / data races / scheduling
 - Optimise (again)
 - Consider false sharing / cache contention
 - Does it scale?
 - To more than N cores?

Programming Complexity

- From most difficult to easiest
 - Cell/GPU > Parallel > Sequential
- Why?
 - Each level adds new complexity
 - Will focus on *offloading* code + data i.e.
 - Data movement
 - Application partitioning

Programming for Heterogeneous Systems

- Where to perform a computation?
 - On small data set? On a large data set?
- Where to place data?
 - In large, relatively slow RAM?
 - In smaller, faster on chip RAM?
- How to coordinate work on different cores?
 - How to program the different cores?
- Is most code your computation?
 - Or code to move data and coordinate?

Impact on Program Correctness

- How is architecture exposed to applications?
 - Does that introduce new potential for errors?
 - All too often, yes
- Coordination + control is hard
 - Correct synchronisation hard to achieve
 - Misuse may lead to subtle memory corruption bugs
 - Errors can be timing dependent

Impact on Code Reuse + Portability

- Worst Case: No Reuse
 - Write / debug / tune / maintain separate code
 - For every class of architecture
- Challenging to accommodate portably
 - Yet, desirable to avoid re-write / duplication
- A promising approach is to combine
 - Compiler techniques
 - Application level portable refactoring
 - Data access tuning

A Software View of Hardware

- Can we hide architecture?
 - Provide illusion of flat memory + homogeneity?
 - Let the compiler
 - move data + code to / from local stores?
 - optimize appropriately for each core type?
- Developers want control
 - Performance may require direct access

A Software View of Hardware (2)

- An old view of hardware pervades languages
 - Scalar types and operations
 - Concurrency via libraries
 - (pthreads, Win32)
 - No support for multiple memory spaces
- Now being addressed for Cell / GPU
 - e.g. Offload C++, OpenCL, C++ AMP

Data Locality Matters...

- Latency + Bandwidth varies
 - It takes longer to access 'distant' memory
 - Higher bandwidths to contention free memory
- Visibility and capacity
 - Can all cores 'see' all the memory?
 - Faster is usually smaller (since slower is cheaper)
- Lots of code becomes memory bound
 - Need to feed cores with code + data
 - Need to move chunks of both around...

Data Access Matters...

- Just because RAM stands for Random Access Memory doesn't mean you should strive to access memory randomly.
 - Colin Riley (@domipheus, *Games Technology Director at Codeplay*)

GPU Programming

- Massively data parallel
 - Many cores (scalar | SIMD) needing many lightweight threads for efficiency
 - Want sets of threads executing in step
 - Work scheduled to mask access latency
 - Care needed to ensure efficient accesses
- Need to get data into device RAM
 - Need to move data within device for efficiency

Cell Programming

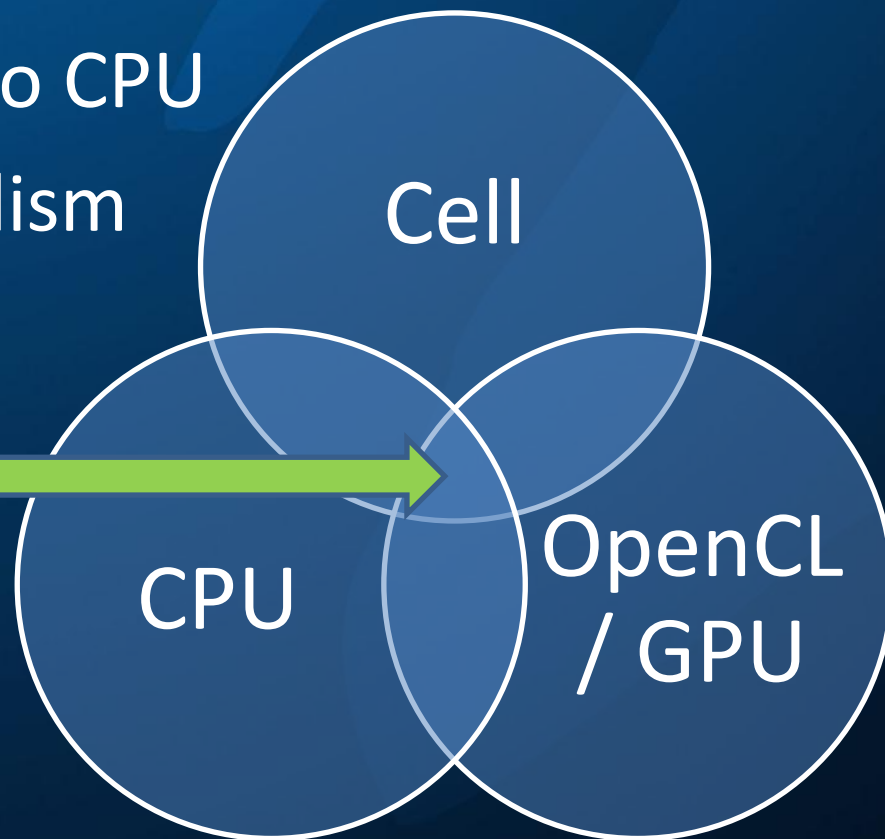
- Flexible (Task / Thread / Data parallel)
 - 6 to 16 SPE independent cores (4 x float SIMD)
 - For efficiency, computing on streaming data
 - Mask latency by interleaving computation and transfer
 - Prefer operating on data in SPU memory
 - SPU SIMD is ubiquitous (no scalar ops!)
 - Branching is costly (no prediction HW)

Accelerator Programming

- Restricted relative to CPU
- Favour data parallelism
- Data on device

Accelerators

- Usable subset?
- Portable subset?
- Large subset?



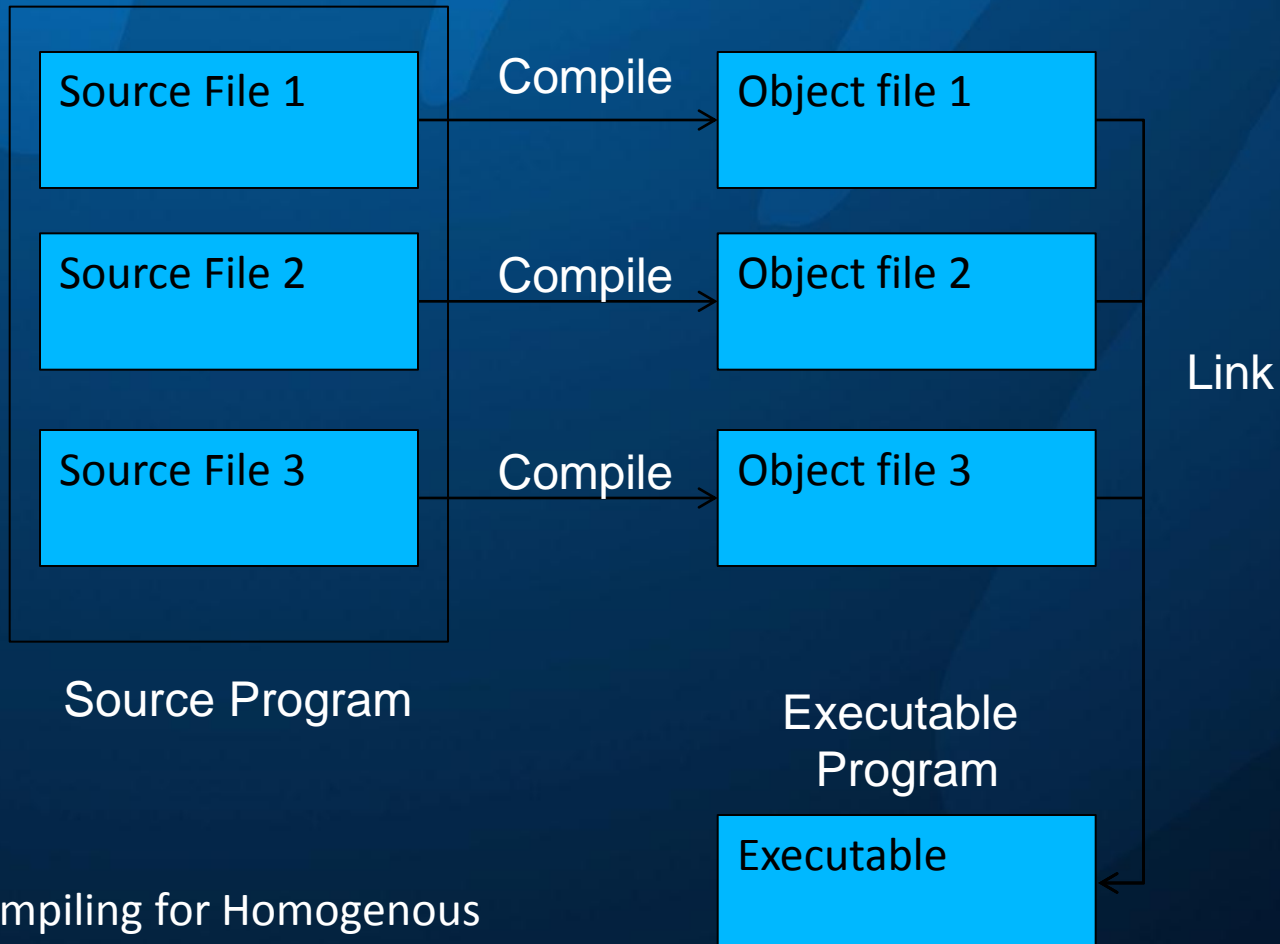
Overview

- High Level View of Accelerator Programming
- **Adapting to Accelerators**
- Offload C++ Tutorial
 - Some (short) C++ examples
- C++ on GPU
- A Demo of C++ for GPU

Adapting to Accelerators

- How to adapt software to accelerators?
 - Without great effort?
 - Avoiding needing to (re)write lots of additional code
 - Avoiding duplication and maintenance
 - With good performance?
 - Without loss of portability?
 - Across a range of distinct classes of system
- More than mere recompile for a new platform

Normal Compilation

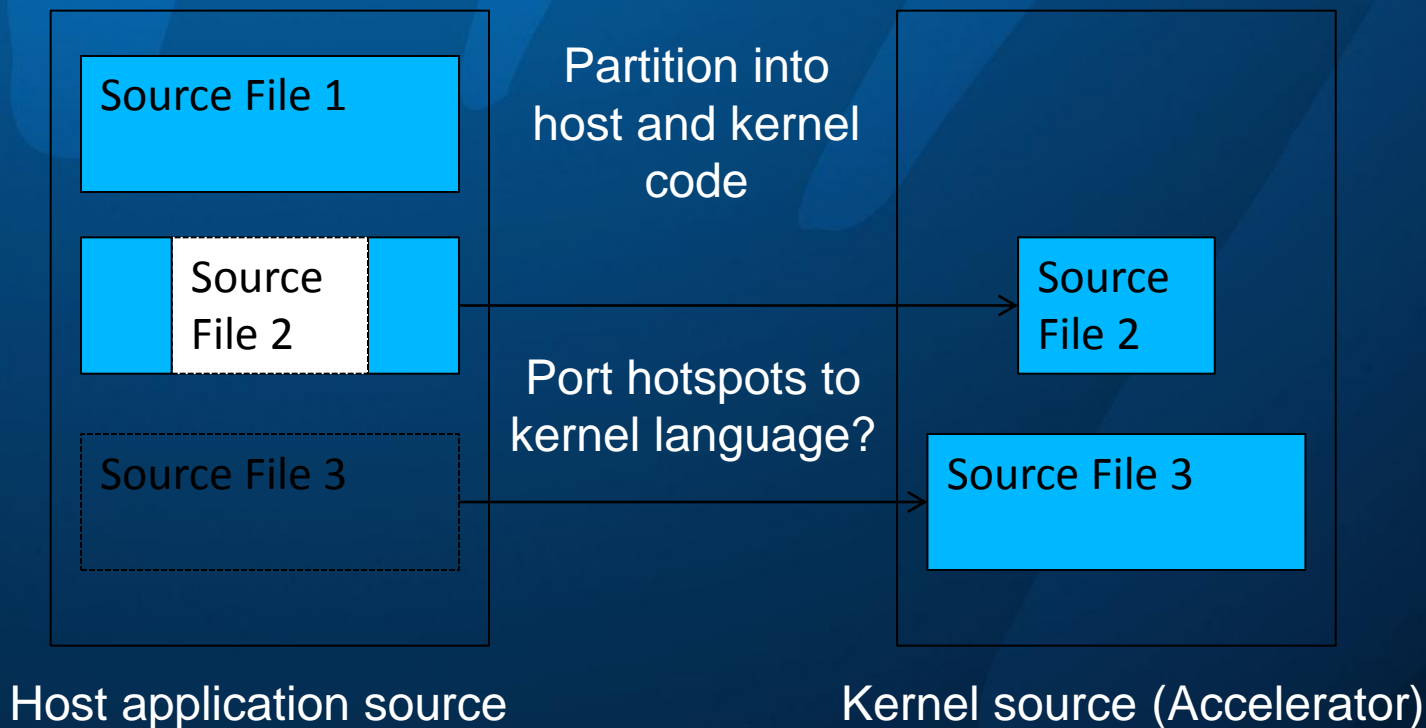


Compiling for Homogenous
Multicore, Shared Memory

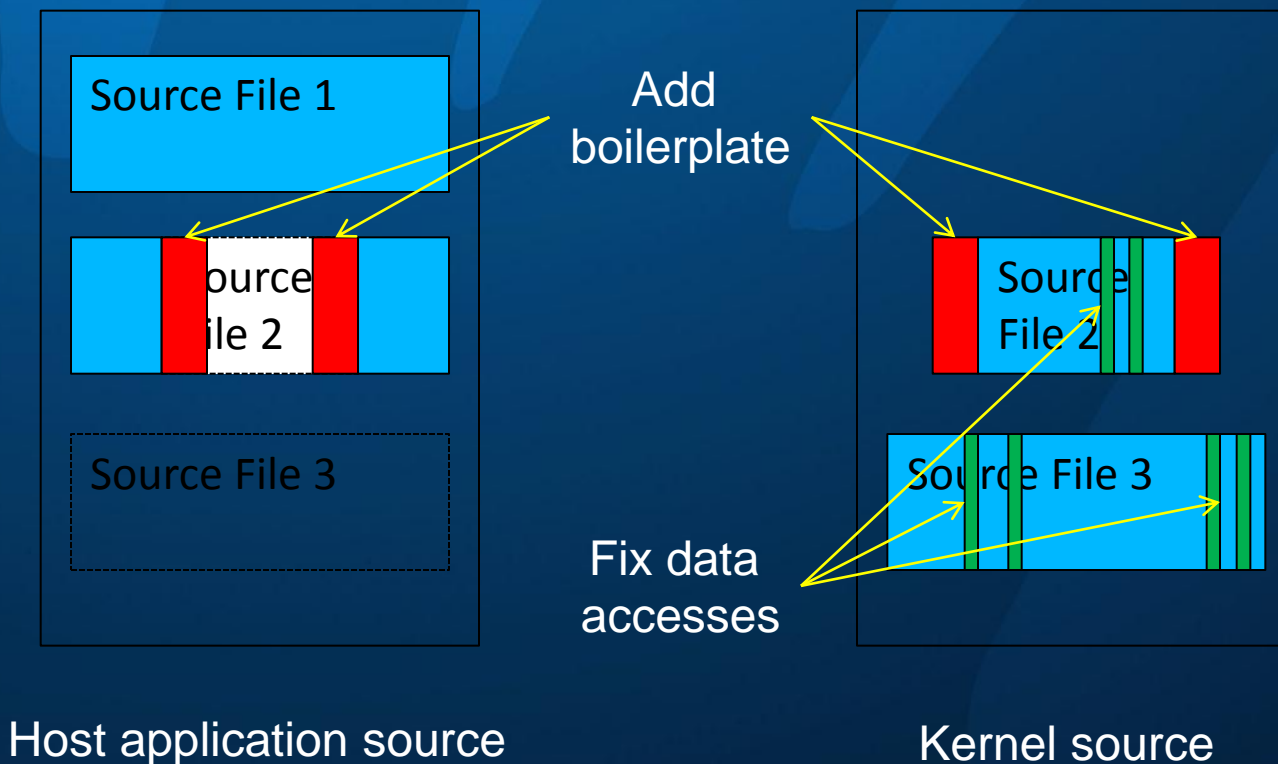
Adapting to Accelerators

- Cannot (sadly) run unmodified on accelerators
 - No silver bullet
- If adapting manually
 - Profile, and partition: run hotspots on accelerator
 - Extract compute intensive code into ‘kernels’
 - Glue this all together
 - Compile kernels, compile host, link together

Adapting to Accelerator Cores

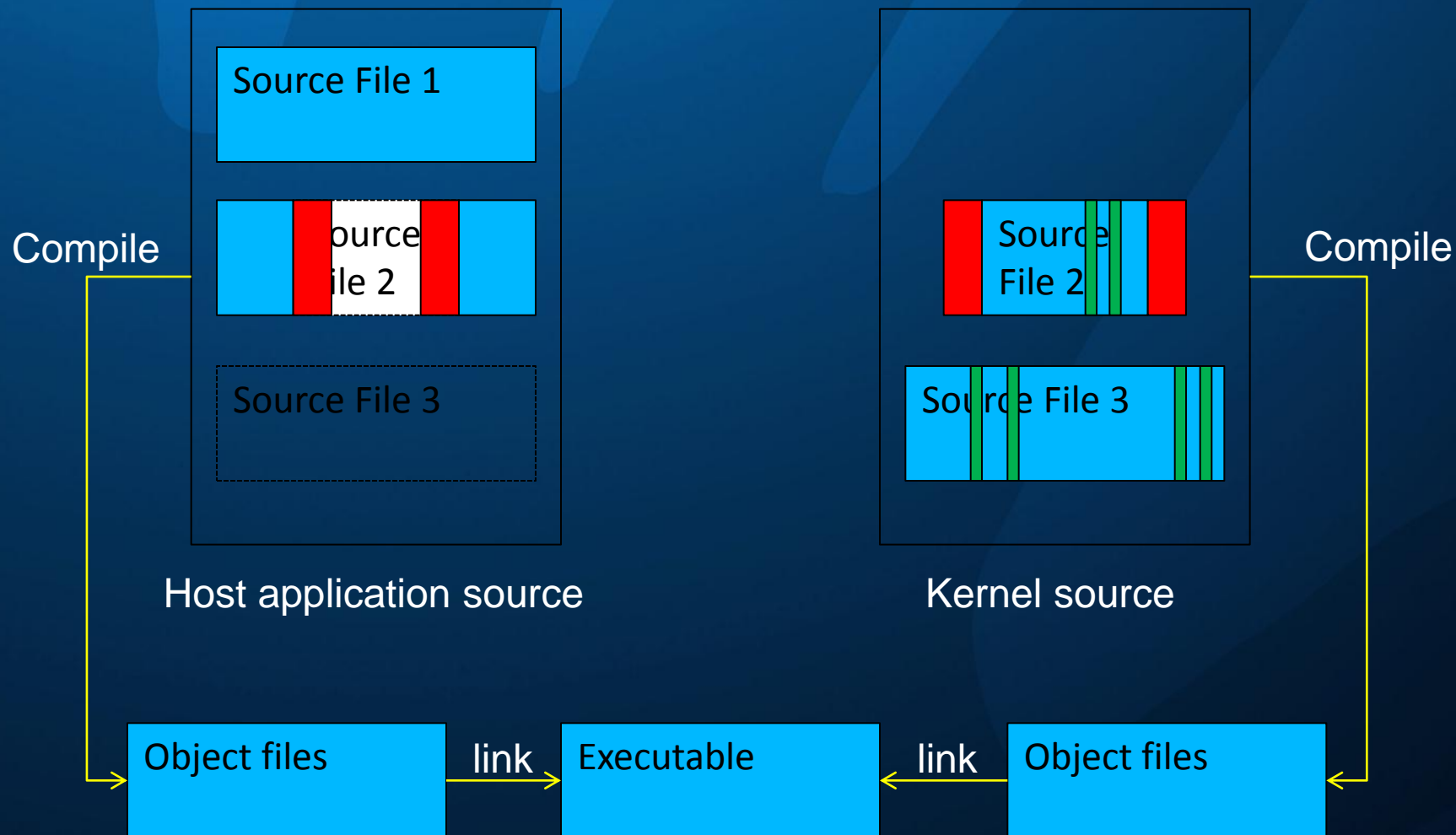


Adapting to Accelerator Cores



Need to add code to control kernel start-up and exit
Need to consider how the kernel accesses data

Adapting to heterogeneous cores



Accelerator Sub-Programs

- Embedded sub-programs
 - On Cell, SPU ELF .o objects
 - On GPU, compile to OpenCL source
 - Dynamically compiled
- Invoked by a runtime system
 - On Cell, various: SPURS, MARS, LibSPE
 - OpenCL: StarPU and the OpenCL API
- Don't want to write glue code to control these

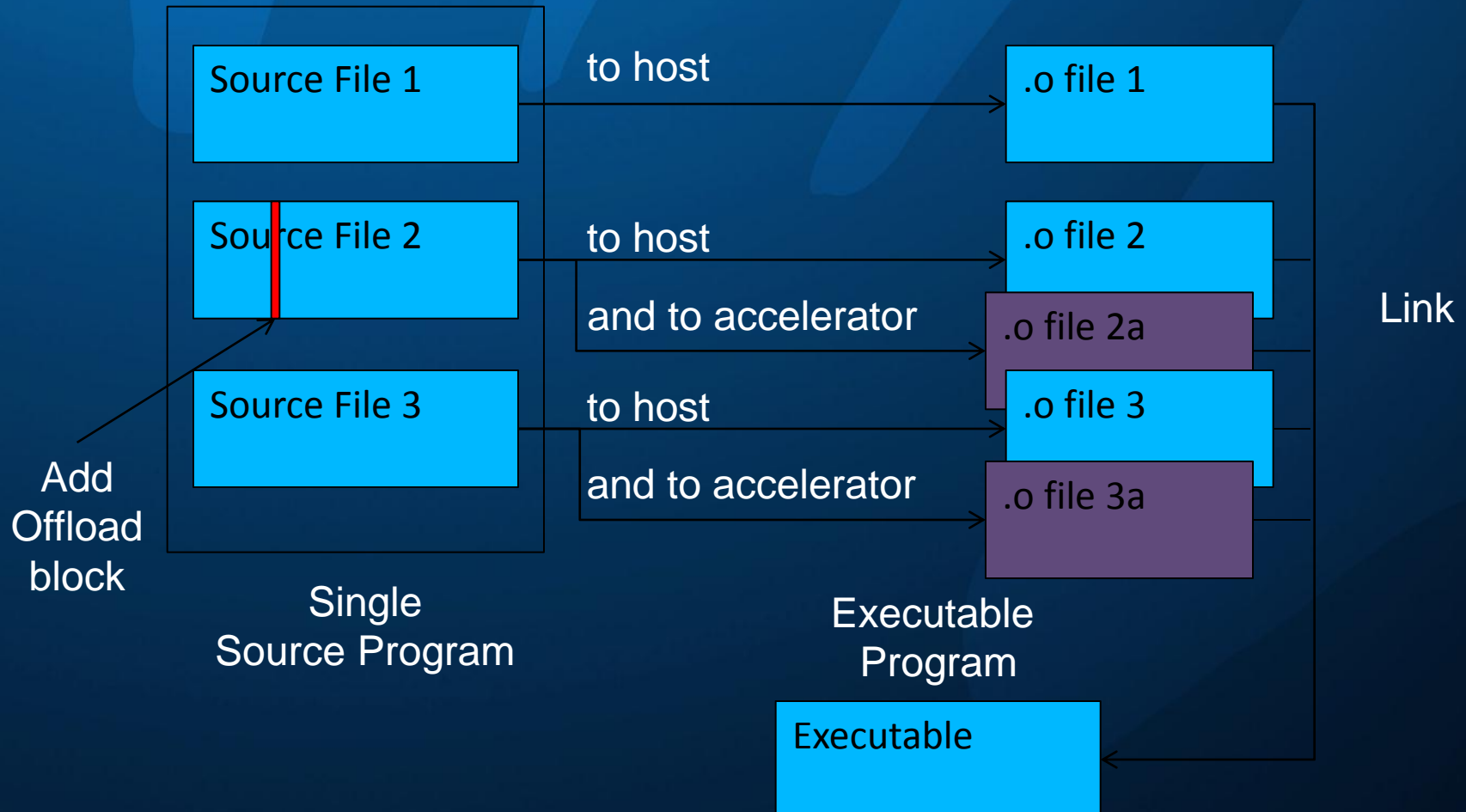
Creating Sub-Programs for Accelerators

- Automate where possible
 - Annotate
 - As little as possible
 - At some suitable level
 - Loop level? Function call? Call-graph? Task / Thread?
 - Let tools do the work
 - This effectively means a compiler
 - Let tools generate boilerplate
 - Free programmer to experiment

Overview

- High Level View of Accelerator Programming
- Adapting to Accelerators
- **Offload C++ Tutorial**
 - Some (short) C++ examples
- C++ on GPU
- A Demo of C++ for GPU

Offload: Adapt Automatically



How to Offload to an accelerator?

- By adding an Offload block to the source
 - And compiling with an Offload C++ compiler
 - (and often, for simple code, that is sufficient)

```
void some_function(T* data, size_t len) {  
    process_data( data, len );  
}
```

```
void some_function(T* data, size_t len) {  
    __offload { process_data( data, len ); };  
}
```

Added offload block

Offloading to an Accelerator

```
3 void offloaded(unsigned char* screenbuf) {.
4     float x_incr = (MAX_X - MIN_X)/(float)gWidth;.
5     float y_incr = (MAX_Y - MIN_Y)/(float)gHeight;.
6     __offload (( x_incr, y_incr, screenbuf )) {.
7         for(int j = 0; j < gHeight; ++j ) .
8             for(int k = 0; k < gWidth; ++k ) .
9                 screenbuf[j*gWidth+k] = mand(k, j, x_incr, y_incr);
10     } .
11 } .
12 .
```

Parameters

Access host memory

Call graph duplication

A Synchronous Offload Block

```
#include <algorithm>
#include <string.h>
int main() {
    int p[] = {4, 3, 9, 8, 34};
    __blockingoffload {
        int local=9; int* ptr = &local;
        std::sort(p,p+5, [=](int l, int r){return l < (*ptr>1?r:0);});
    }
    for (int i=0; i<5; i++)
        std::printf("p[%i] = %i;\n",i,p[i]);
}
```

Invoke library code on
accelerator

C++0x Lambda as
predicate

How to Offload to accelerators?

- We want to use >1 accelerator!
 - We don't want to block the host
- We want to offload threads
 - Existing threads, or create new threads
- We use asynchronous offload blocks
 - Similar to conventional thread APIs
 - Can spawn an offload block
 - Obtain a handle to the running offload block
 - And wait for it to terminate via a `join()` call.

An Asynchronous Offload Block

```
void some_function(T* data, size_t len) {  
    // Spawn thread to run on accelerator  
    thread_t handle = __offload( data, len )  
    { // Process 3/4 on the accelerator (e.g. SPU)  
        do_work( data, len-(len/4) );  
    };  
    // Process 1/4 on host (e.g. on PPU)  
    do_work( data + len-(len/4), (len/4) );  
    join(handle) ; // await completion of offload thread  
}
```

Creates accelerator thread
and returns a thread handle

Can capture copies of local
stack variables

Code in block executes on
accelerator in parallel with
host

The host can also perform
computation

Join call awaiting accelerator task/thread exit

Offload + Deferred Function Calls

- Call occurs on another thread
 - On Accelerator, with Offload
 - `thread <RETURN_TYPE, ARGUMENT_TYPE, FUNCTION_NAME>`

```
using namespace  
    liboffload::constructs;  
thread<int, int, PlusPlus>  tPlus;  
int start = 2;  
tPlus.spawn(start);  
int r0 = tPlus.join();
```

```
int PlusPlus(int v) {  
    return ++v;  
}
```

Specify function + types

Invoke with argument

Get return value

Offload C++

- Conservative C++ extension
 - Compiler, run-time, and libraries
 - Applicable to *existing* code bases
- Targets heterogeneous cores
 - Host core + accelerator cores (e.g. Cell)
 - Support for distinct memory spaces
 - Compiler generates / checks cross core code
- Programming model
 - Migrate threads /tasks onto an accelerator

Task Parallelism

- Games often use Task Parallelism
 - Work is decomposed into tasks
 - Relatively small, self contained
 - Tasks are executed on a schedule
 - Scheduled to allow data to be produced + consumed
 - Scheduled to avoid contention on data
 - Tasks can be run on CPU or accelerators
- A task \sim An asynchronous function call

Offload Tasks

```
__offloadtask thread_t an_accelerator_task(params...) {  
    do_work(params...); // Compiled for accelerator  
}
```

// Capture the function call

thread_t handle = an_accelerator_task(1,2);

// Start function call execution (on accelerator)

offloadThreadStart(handle);

join(handle); // Wait for completion

Task is created, but not yet invoked

Invoke the task, with arguments given previously

Wait for task completion

Its not quite so simple

- Where is the data?
- How is data to be accessed?
- In our examples, we offloaded *computation*
 - The array “data” resides in host memory
 - What are the implications of that?

```
void some_function(T* data, size_t len) {  
    __offload { process_data( data, len ); };  
}
```

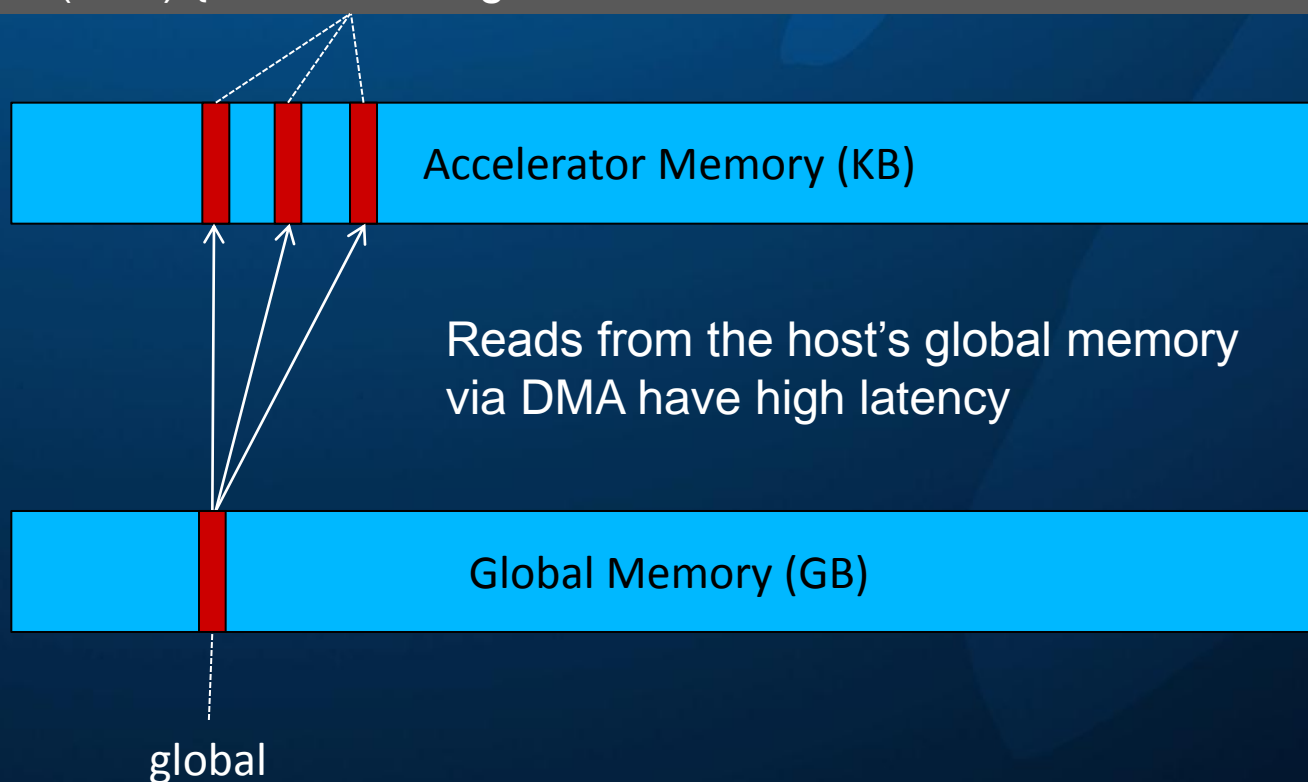
Where is “data” allocated?

Data Access on Host

- What happens to read a global variable?
 - On the host (e.g. PPU), issue a read instruction
 - Check the L1 / L2 cache for data
 - ~60 cycles on L1 miss
 - ~500 cycles on L2 miss
 - Not there? Get it from RAM
 - Slow, but at least it'll be in the cache next access
 - (Probably)
 - How about from an Offload block on an SPU?
 - Need a DMA transfer to access host RAM

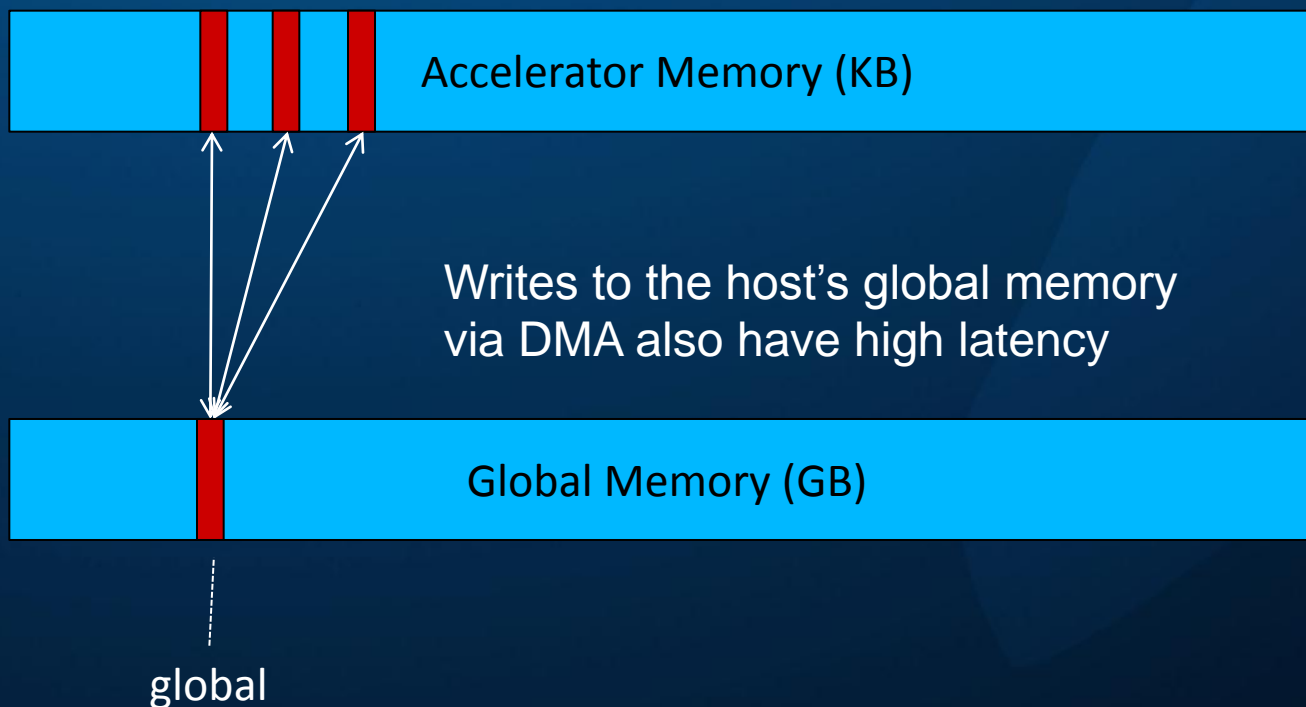
Data Access on Accelerator

```
float fun(int n) { float local = global; return n == 0 ? local : local + fun(n-1); }
```



Data Access on Accelerator

```
void fun(int n) {float local = global + n; global = local; fun(n-1); }
```

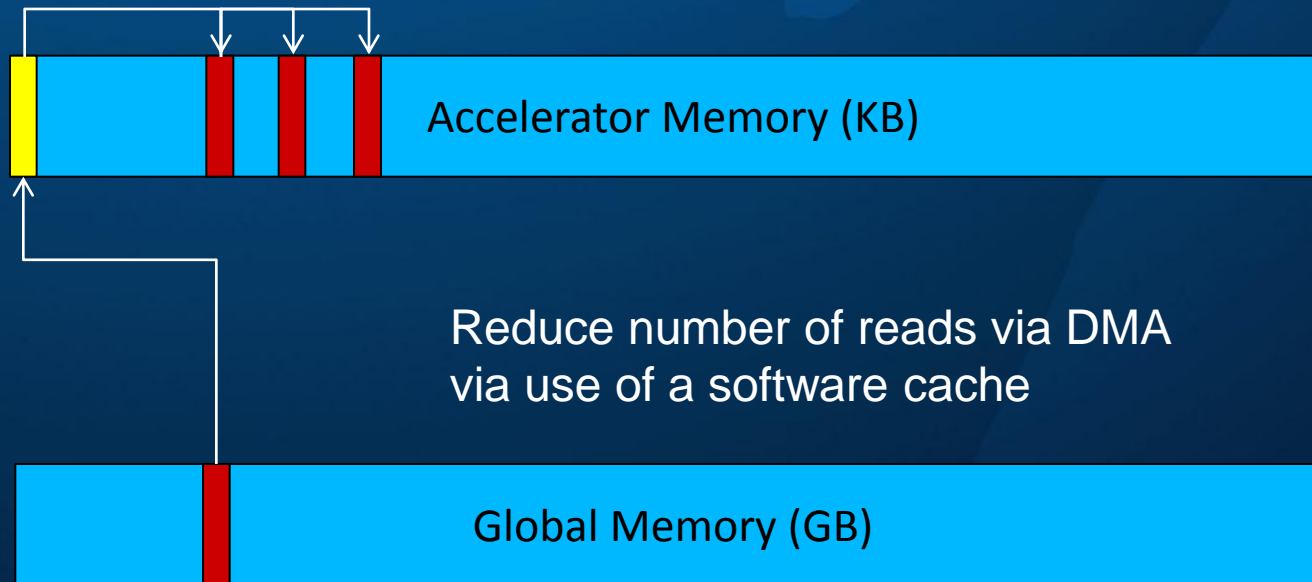


Avoiding Repeated DMA

- Offload Compiler generates data access code
 - Could naively generate direct DMA transfer
 - Generates read via software cache instead
 - Reserves a little accelerator memory for cache
- Could use a software cache explicitly
 - Convenient / safer to have compiler insert the calls

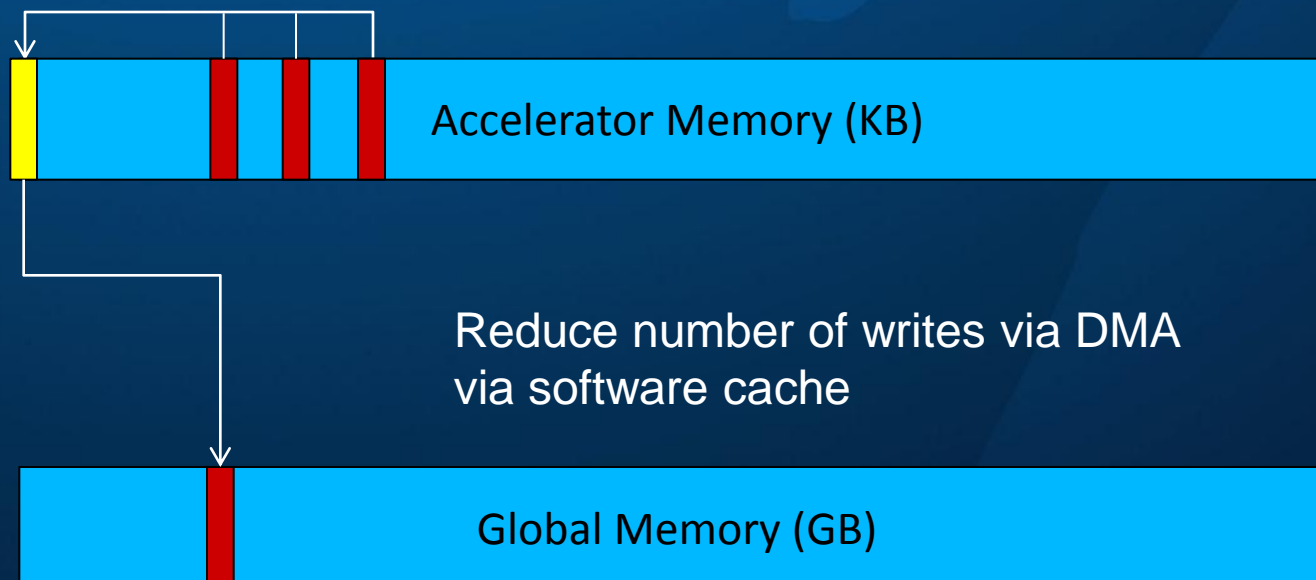
Software Cache - Reads

```
__offload float fun(int n) { float local = global; return n == 0 ? local : local + fun(n-1); }
```



Software Cache - Writes

```
__offload void fun(int n) { float local = global + n; global = local; fun(n-1); }
```

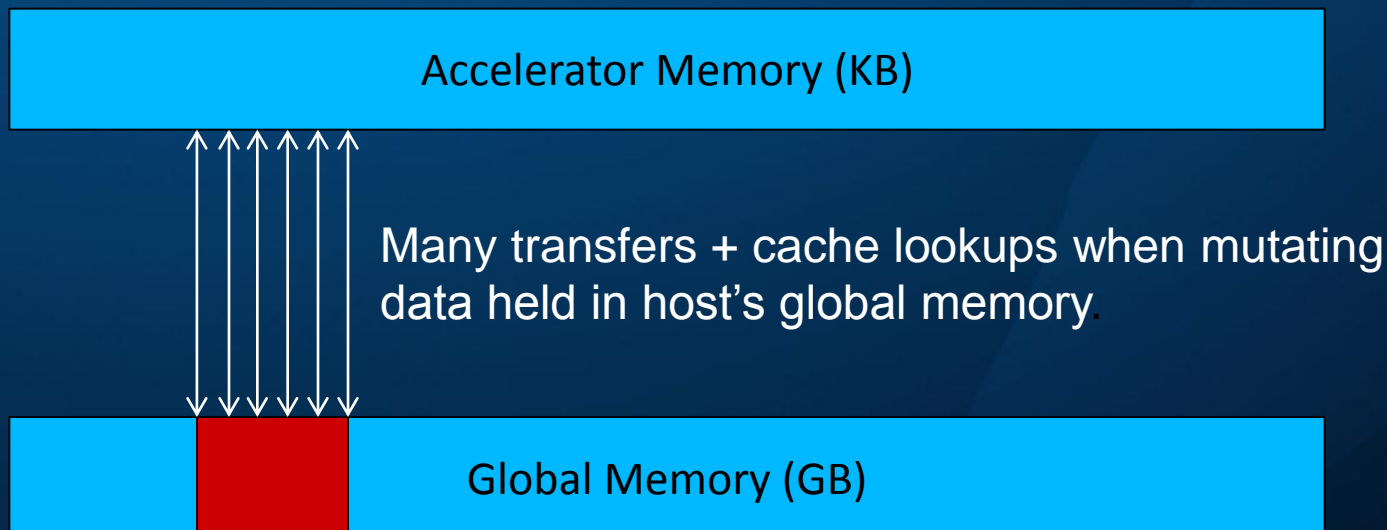


Data Movement Strategies

- None: fetch on demand
- Software caching
- Local shadowing
 - Offloaded local variables allocated in local store
 - Prefer local variable access to global
 - Copy data in; use it; copy out if needed
 - i.e. `local_v = global_v; ... ; global_v = local_v;`

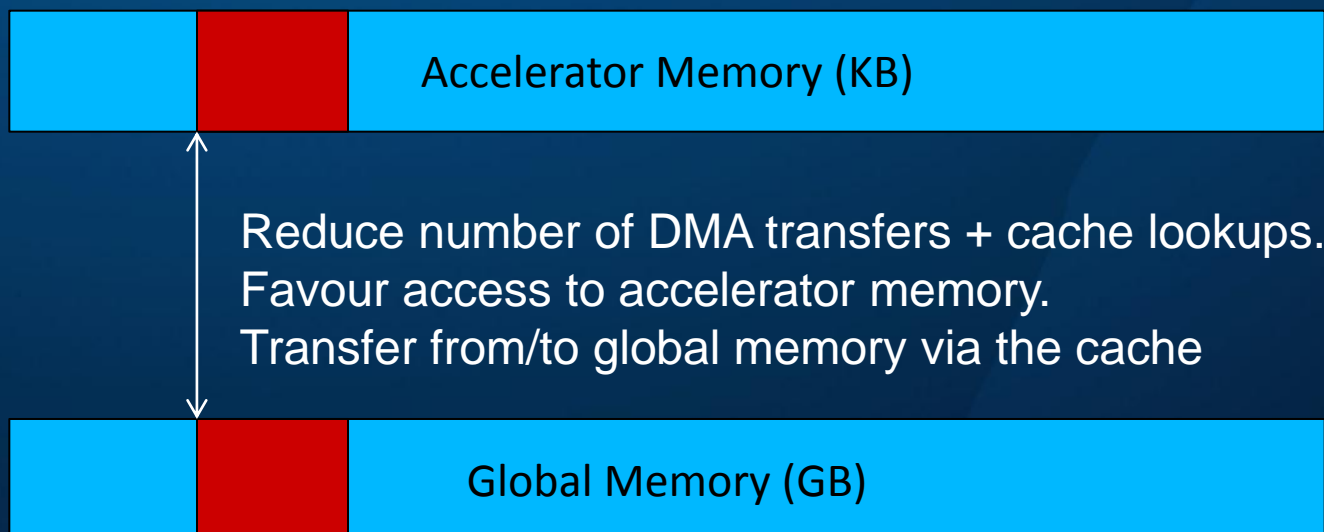
(Without) Local Shadowing

```
void fun () { __offload { mutate(&global_data); }; }
```



Local Shadowing

```
void fun() { __offload { T local = global; mutate(&local); global = local; } }
```



CPU, Cell, and GPU differences

- The GPU is most restrictive
 - Must identify & buffer all data a kernel *may* access
- The Cell allows on demand fetches
 - Can be flexible, at cost of performance
- On CPU, can freely follow pointers
 - No visible ‘transfers’, just ‘accesses’
- Discuss later an approach for Cell/CPU/GPU

Accelerator Programming

- Accelerators provide fast + specialised operations
 - On SPU, vector + data transfer operations
 - Exposed to programmer as intrinsic functions
 - Use of which is very non-portable...
- We want to use these, when available
 - Have a portable fall back

Overloading for Offloads

- Add overloads optimized for accelerator
 - Compiler selects function from location of call

```
#ifdef __offloadcpp
__offload void process_data( T* data, size_t len ) { ... }
#endif
void process_data( T* data, size_t len ) { ... }
```

```
void some_function(T* data, size_t len) {
    __offload { process_data( data, len ); };
}
```

Offload Contexts

- Permit intrinsic functions in `__offload` code
- In an offload context:
 - inside an offload block
 - in an `__offload` function
- We use overloads & C++ templates
 - Build portable, efficient data transfer abstractions
 - Operator overloads mimic regular operations

Data View / Transfer Abstractions

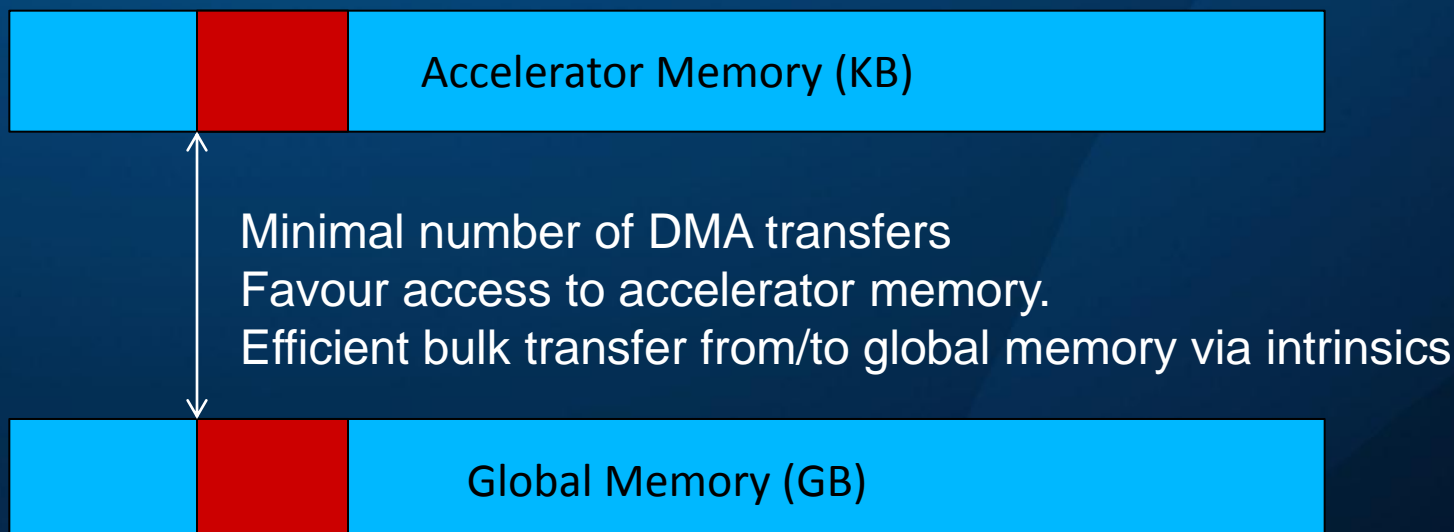
- Data structure / access pattern specific
 - Read / Write / Read Write
 - Traversal aware? Random access?
- Motivations
 - Fast access to (subsets of) of data
 - Hide specific transfer mechanisms
 - Provide usage equivalent to main-memory use case

Downsides of a Software Cache

- No application level knowledge
- Non-optimal performance
 - Migration aid, developer convenience
 - Better than naïve DMA
 - Not suited to all access patterns / applications
 - Customisable is desirable
 - Operates synchronously
- Permit bypassing the cache

Data Movement Strategy

```
void fun() { ReadWriteArray<T,N> local(&global_data[0]); mutate(&local[0]); }
```



Data Movement Strategies

- C++ bulk transfer strategy templates
 - Bypass software cache
 - Can be instantiated in portable code
 - Hide non-portable code behind template interface
e.g. DMA intrinsics
 - Provide a portable fall-back implementation
 - Select strategy implementation at compile time
 - Instantiate on basis of context and target core
- Permit asynchronous transfers

Offloading a Method Call

- A common use case for Offload
 - Run a method on an accelerator

```
__offload() {  
    ptr->func(ptrarg1,ptrarg2);  
}
```

Wrap call site in
offload block

Call takes pointers
as arguments

Here, pointers to
host memory

Pass by reference
is common.

Implicit parameter to method of
'this' – also a host pointer

The call is to a method on an
object allocated in host memory

Offloading a Method Call

- Performance penalties?
 - (Frequent) access object fields via “this” pointer
 - (Frequent) access to data via pointer arguments
- Worthwhile caching data pointed to
 - Emulate passing data by value
 - Subject to some caveats:
 - Space in SPU local store e.g. SPU stack
 - Consideration of side-effects
 - Legality of copy / instantiation in C++

Offloading a Method Call (2)

- Cache instance data
 - Run a method on an accelerator with object data in local store

```
__offload() {  
    auto localObject = *ptr;  
    localObject.func(ptrarg1, ptrarg2);  
    *ptr = localObject;  
}
```

DMA transfer or SW cache read into local

Call arguments
still pointers to
host memory

Implicit parameter to method of
'this' – now a local pointer

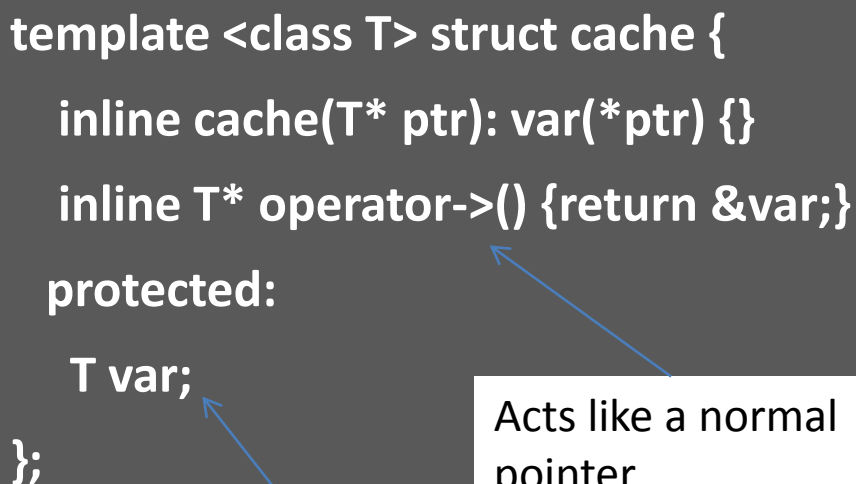
Transfer object back, if desired or
needed. Careful not to clobber
writes performed in func!

The call is now to a method on an object
allocated in accelerator memory

Writing an Offload Cache Class

- Want to minimize code change
 - Would like to encapsulate this caching pattern

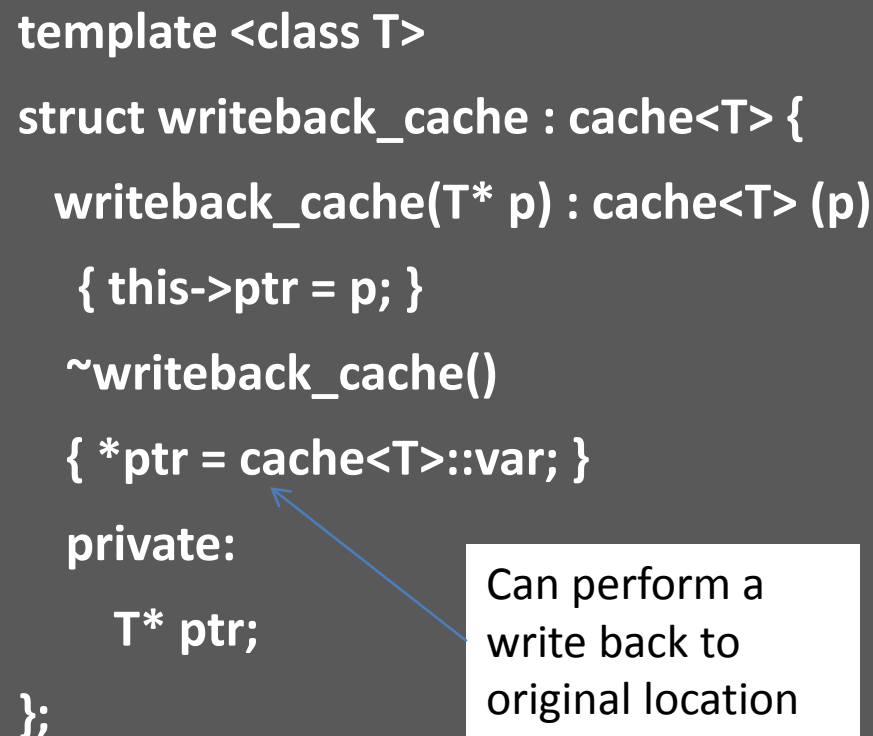
```
template <class T> struct cache {  
    inline cache(T* ptr): var(*ptr) {}  
    inline T* operator->() {return &var;}  
protected:  
    T var;  
};
```



Captures a copy
in local store
when created

Acts like a normal
pointer

```
template <class T>  
struct writeback_cache : cache<T> {  
    writeback_cache(T* p) : cache<T> (p)  
    { this->ptr = p; }  
    ~writeback_cache()  
    { *ptr = cache<T>::var; }  
private:  
    T* ptr;  
};
```



Can perform a
write back to
original location

Offloading a Method Call (3)

- Again, now using our cache class
 - We could further cache the arguments

```
__offload() {  
    Type* tmp = ptr;  
    writeback_cache ptr(tmp);  
    ptr->func(ptrarg1, ptrarg2);  
    ...  
}
```

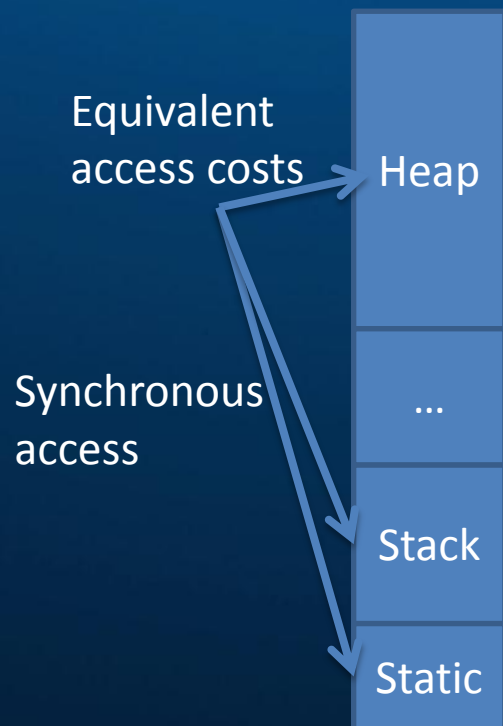
Name shuffling to shadow the global 'ptr' with the local cache variable 'ptr'

Implicit parameter to method of 'this' is a local pointer via our cache

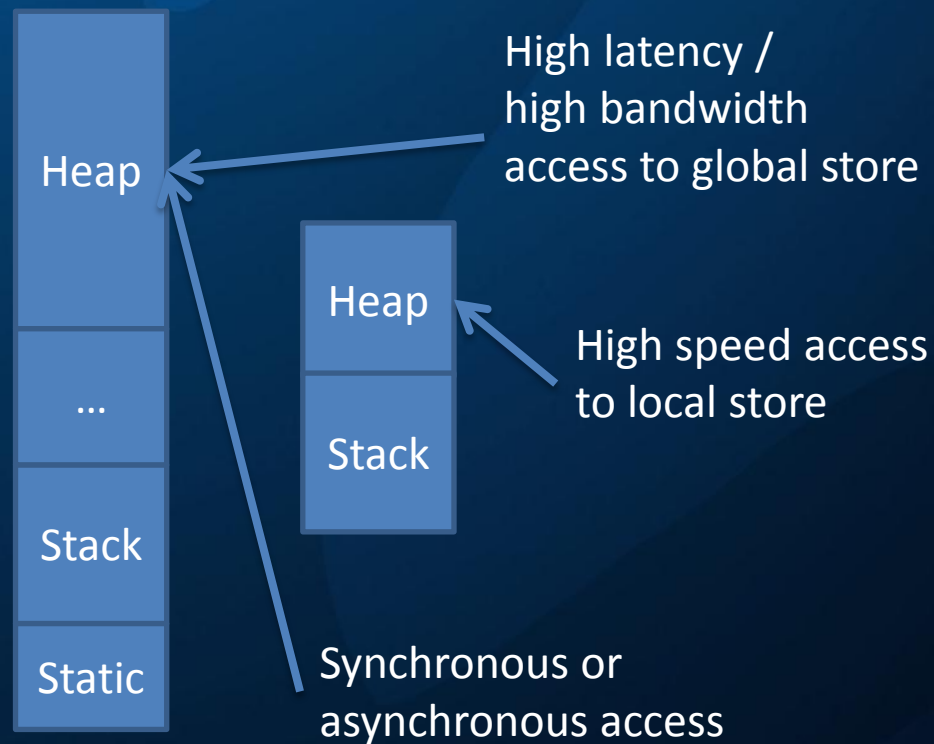
Call and any subsequent calls via ptr need no modification

Data Access Costs

Task on CPU



Task on Accelerator



Fixing Data Accesses

- The compiler can inform of inefficient access
 - Fragmented transfers / Misaligned transfers
 - Efficient transfers of contiguous, aligned data

```
int add(int*a, int*b)
{ return *a + *b; }

int main() { int oVar=1;
__offload() {
    int r = add(&oVar,&oVar);
}}
```

Compiled with options

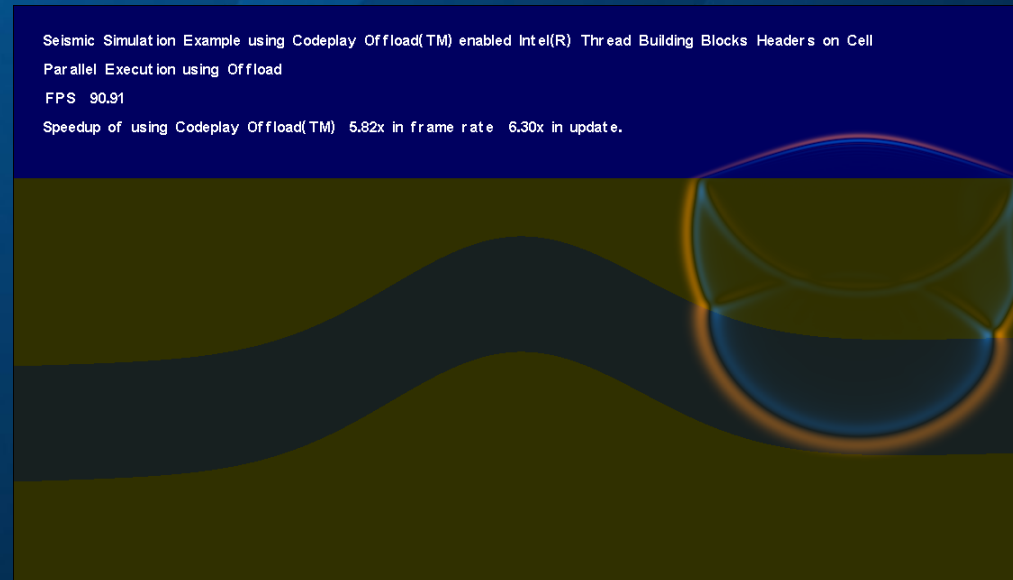
-warnonouterreadswrites -fno-inline

```
*** WARNING: Generating outer
read, Alignment 4 , Size: 4, in function:
_Z3addPU7__outeriPiEU3_SL1.
--- In file: offloadadd.cpp, at line: 7,
column: 0
```

Compiler issues a
warning per access

The Seismic Demo

- Animated simulation
- Wave propagation in rocks, water
- Two compute loops
 - calculating stresses and wave velocities
 - Traverse large 2D data arrays e.g. vertical and horizontal stress, velocity, damping



The Seismic Demo

- Demo in Intel Threading Building Blocks (TBB)
 - TBB is task based parallel programming library
 - Provides high level parallel constructs
 - `parallel_for`, `parallel_reduce`, `parallel_invoke`, ...
- How can we offload this onto Cell and GPU?

Why Offload TBB Code?

- Concurrency via C++ template library
 - C++ templates permit compile time techniques
- Explicit indication of (potential) parallelism
 - No explicit threading / work division
- Implemented with multi-threaded runtime
 - We can offload threads...

Introducing parallel_for

- `for (int i = 0; i < N; i++) { ... }` becomes
- `parallel_for(range<int>(1,N),Body());`
- Body is a function object (or lambda in C++0x)

```
struct Body {  
    void operator() (range<int> & r ) {  
        for(int i=r.begin(); i!=r.end(); ++i ) { ... }  
    } };
```

Inside parallel_for

- Loop iteration space represented by a range
- ranges are *splittable*
 - work (loop iterations) divided between tasks executed on threads
 - tasks apply the function object to sub-ranges
- parallel_for is a facade over the task scheduler
 - scheduler manages task creation, work stealing
- Implement our own parallel_for
 - Offload some or all work

Offloading parallel_for

- Implemented parallel_for with Offload C++
 - Execute unmodified code on SPUs and PPU
- How?
 - Using `__offload {}` blocks to put code on SPU
 - Using automatic call-graph duplication
 - Compile call graph reached from function object
 - Divide work range between SPUs and PPU
- Use our header and compiler

Offloading parallel_for

- Several implementations
 - Different work divisions
 - Static / Dynamic / Work Stealing
 - Different breakdown of workspace
 - Rows / tiles / strides
 - Use differing worker cores
 - 1-N SPU, PPU too?
- TBB evolves with C++
 - Function objects, now also C++0x lambda blocks

Offloading parallel_for

```
// Spawn a thread per SPU
```

```
subranges[nthreads+1] = ...
```

```
for (int i = 0; i < nthreads; i++ ) {
```

```
    range<int>subrange(subranges[i]);
```

```
    handles[i] = __offload( body, subrange ) {
```

```
        body(subrange); // Execute a sub-range asynchronously on SPU
```

```
    };
```

```
}
```

```
body(subranges[nthreads]); // Execute a sub-range on PPU
```

```
for (int i = 0; i < nthreads; i++ )
```

```
    join(handles[i]); // Await SPU threads
```

Divide N-dimensional work ranges into sub parts

Spawn configurable number of threads

Sequential work on sub range on accelerator

Sequential work on sub range on CPU

Wait for async threads on accelerator, CPU work was done synchronously

Seismic Demo in 1 Slide

```

void Universe::ParallelUpdateVelocity(tbb::affinity_partitioner &affinity) {
    tbb::parallel_for( tbb::blocked_range<int>( 0, UniverseHeight-1 ),
        UpdateVelocityBody(*this),
        affinity );
}

struct UpdateVelocityBody {
    Universe & u_;
    UpdateVelocityBody(Universe & u):u_(u){}
    void operator()( const tbb::blocked_range<int>& y_range ) const {
        u_.UpdateVelocity(Universe::Rectangle(0,y_range.begin(),
            u_.UniverseWidth-1,y_range.size()));
    }
};

void Universe::UpdateVelocity(Rectangle const& r) {
    for( int i=r.StartY(); i<r.EndY(); ++i )
        for( int j=r.StartX(); j<r.EndX(); ++j )
            V[i][j] = D[i][j]*(V[i][j] +
                L[i][j]*(S[i][j] - S[i][j-1] + T[i][j] - T[i-1][j]));
}

```

Index space for loop

Body of loop

Affinity hint

Functor struct

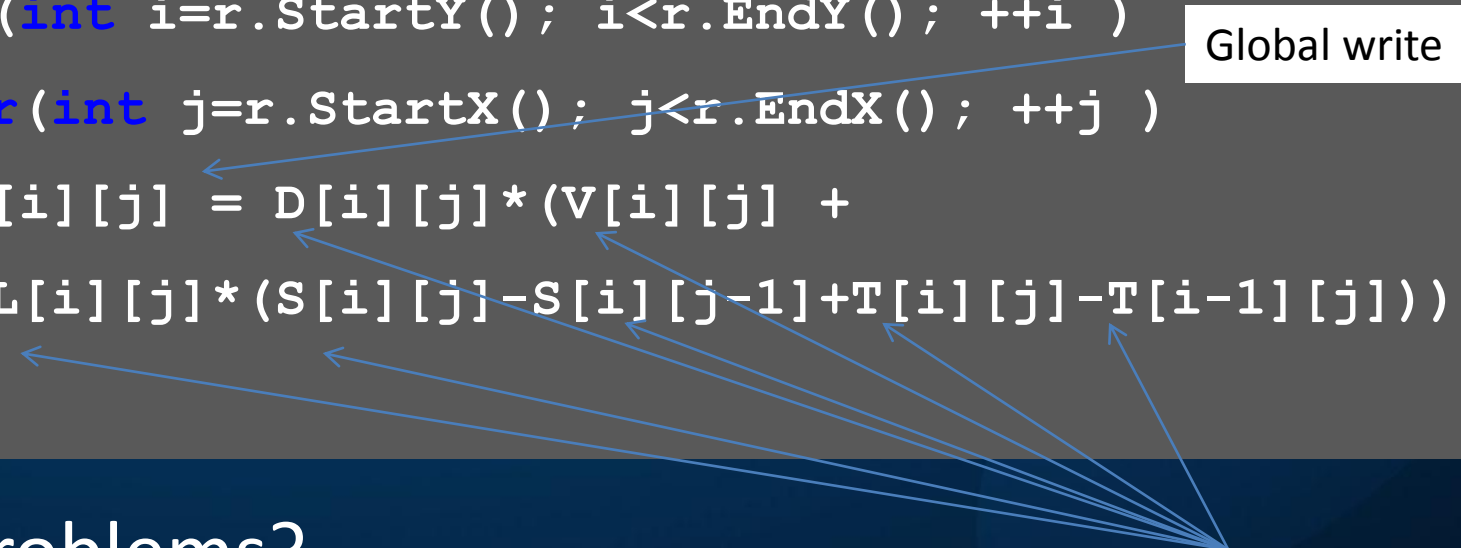
Parameter to functor

Body of loop as functor

Body of loop as function

Data Access Template Example

```
void Universe::UpdateVelocity(Rectangle const& r) {  
    for(int i=r.StartY(); i<r.EndY(); ++i )  
        for(int j=r.StartX(); j<r.EndX(); ++j )  
            V[i][j] = D[i][j]*(V[i][j] +  
                L[i][j]*(S[i][j]-S[i][j-1]+T[i][j]-T[i-1][j]));  
}
```



- Problems?
 - Many global reads and writes.

Data Access Template on Cell

```
void Universe::UpdateVelocity(Rectangle const& r) {  
    for(int i=r.StartY(); i<r.EndY(); ++i) {  
        ReadArray<float,Width>  lD(&D[i][0]),  
                                lL(&L[i][0]), lS(&S[i][0]),  
                                lT(&T[i][0]), lpT(&T[i-1][0]);  
        ReadWriteArray<float, Width> lV(&V[i][0]);  
        for(int j=r.StartX(); j<r.EndX(); ++j )  
            lV[j] = lD[j] * ( lV[j] + lL[j] * ( lS[j] - lS[j-1] + lT[j] - lpT[j] ) );  
    }  
}
```

Pull input to fast local store

Mutated data gets written back

Compute on data in fast local store

Custom Portable Abstractions

- User definable, overloadable, extendable
 - Implement custom versions of abstractions
 - provide custom data access templates
 - Abstract over platform APIs e.g. (mutex, lock)
 - Hide use of language extensions
- Implement other parallel APIs?

C++ and Memory Spaces

- Standard C++ specifies no semantics for NUMA with multiple memory spaces
- Offload C++ extends type checks
 - Pointers to different memory spaces are incompatible
 - Detection of inconsistent usage is safer
 - Type level distinction aids in informing programmer

Types in Offload C++

- Generate different code for accesses via different kinds of pointer
 - Prevent device pointers escaping onto host

```
void f() { int on_host;  
  __offload {  
    int on_device;  
    int __outer* a = &host;  
    int * b = &on_device;  
    a = b; // Illegal assignment!  
  }  
}
```

An __outer pointer

Annotations? We don't want those!

Reject usage of device pointer in place of host pointer

Call Graph Duplication + Type Inference

- Analogous to C++ template instantiation
 - A normal C/C++ function

```
void add(int* res, int *a, int *b){*res=*a+*b;}
```

- A normal C++ function template

```
template <typename T>
```

```
void add(T* res, T *a, T *b){*res=*a+*b;}
```

- We can instantiate with a specific type

```
add<int>(res, a, b);
```

Call Graph Duplication

- What can we infer?
 - The types of arguments
 - Therefore, which function is called
- What else?
 - We know the context of the call site
 - We know to which memory space pointers refer
- We “instantiate” a suitable function
 - At compile time
 - For appropriate processor + memory type

```
int res=0, a=1, b=2;  
add(&res, &a, &b);
```

Why Call-Graph Duplication?

- Combinatorial explosion of signatures
 - Don't want to explicitly duplicate in source
 - Maintenance nightmare

```
void add(int * res, int* a, int * b);  
void add(int * res, int* a, int __outer * b);  
void add(int * res, int __outer * a, int * b);  
void add(int * res, int __outer * a, int __outer * b);  
void add(int __outer* res, int* a, int * b);  
void add(int __outer * res, int* a, int __outer * b);  
void add(int __outer * res, int __outer * a, int * b);  
void add(int __outer * res, int __outer * a, int __outer * b);
```

Call Graph Duplication + Inference

- Propagate `__outer` during compilation
 - Through initialisations, casts, usage
 - Recursively to compile whole call graph
 - Create only needed duplicates

```
int* a, *b, *r;  
void fun() {  
    f(r,a,b);  
    __offload {  
        f(r,a,b); int * x, *y, *z  
        f(r,x,y); f(x,y,z);  
    }  
}
```

```
void f (int * res, int* a, int * b) {  
    int * z = a;  
    *res = g(z , b);  
}
```

Call Graph Specialisation

- C++ Templates support specialization
 - Special casing specific types
- Analogously, we can override duplication
 - `__offload` function overloads suppress duplication
 - Can optimize specific cases

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
- Automation allows rapid experiments + larger scale

Ease of Offloading

- Applied to PS3® complex Game AI code
 - Running on SPU in < 1 hour (~30% speed)
 - Just offload block + SW cache
 - 7hrs iterative optimization of *running* SPU code
 - Profiling memory access
 - Add 20 loc (portable C++ data access templates)
 - Final speedup ~4x *faster* than PPU

NASCAR The Game 2011™

Paris Game AI Conference 2011
@sheredom, Technology Lead,
Codeplay



NASCAR The Game 2011 - Developed by Eutechnyx

Offloads in NASCAR

- Late in development cycle AI Offload
 - CPU over utilized / little time to implement
- Serial AI? less possible AI bot characters
- Offloaded 3 major AI components
 - Onto 1 and 4 SPUs
 - Small code changes (~100 / ~20 loc)
- 50% AI speed increase on PS3
 - < 1 month dev time
 - Incremental profile, refactor, offload

What Code to Offload?

- C++ is not a small language
 - Chance of encountering feature X increases with size of call graph attempted...
- Some challenges (Implemented in Offload C++ for PS3)
 - Calls to 3rd party library code on CPU (middleware)
 - On Cell, can perform callback - not on GPU?
 - Indirect calls
 - Virtual methods + calls via function pointer?
 - Inline assembly?
 - Intrinsic operations e.g. Altivec?

Do you have a flat profile?

- It is likely code has already been optimized
 - If it were too slow, it probably was fixed
- Does your application have a hot-spot?
 - Scientific computing on large data sets – yes
 - A simulation is going to spend a lot of time simulating...
 - Many different real time processes
 - E.g. a game – maybe
 - Simulation + AI + path finding + graphics + sound + physics + decompression + scripting + shading ...

Do you have complex data?

- That contains pointers?
 - (trees, graphs, lists, objects, ...)
 - Including hidden in vtables
 - More structured than arrays of ‘plain old data’ ?
- Patching data
 - Copied data needs fixed
 - Find, and adjust, *each* pointer
 - (if wrong, your program will crash – at some point)
- Easy to evolve a convoluted data layout

Do you have complex code?

- Does it perform many different actions?
 - Does it operate on many different types?
 - Does it dynamically select operations?
 - Is it parameterised by code?
 - Does it invoke call-backs? virtual methods?
 - Does it just take arguments + return a result?
 - Does it read or mutate global variables directly?
 - Or indirectly, via pointers?
- Is it split over many inter-related routines?
 - In many compilation units / source files?

What are accelerators good at?

- Simple computation over simple data
 - Lots of data
 - Lots of (repeated) computation
- Is this the code you have?
 - Actually, accelerators can be good at more complex code
 - It takes more programmer effort

Offloading Virtual Methods

- Call graph duplication of late bound calls
 - function pointers / virtual methods
- Offload block 'domains'
 - select functions to duplicate for indirect calls
 - Lookup accelerator implementation via host address

OOP on Accelerators

// A (very simple) class hierarchy with virtual methods

```
struct Object { virtual void update();}
```

```
struct SubClass : public Object { virtual void update();}
```

// A collection of objects to update in simulation

```
Object * objects[N_OBJECTS]; // Objects allocated in global heap
```

```
void update_objects() {
```


// Partition code: Inside the offload block is compiled to accelerator

```
offload [ Object::update, Subclass::update] {
```

```
    for (int i = 0; i < N_OBJECTS; i++)
```

```
        objects[i]->update(); // Invoke virtual method on each object
```

```
}}
```



What code is invoked?
What data is accessed?

OOP on Accelerators (2)

- What was inefficient before?
 - Per iteration: get pointer, lookup function, invoke
- Remove a high latency fetch per iteration
 - Desirable to pre-fetch member data too..

```
offload [ Object::update, Subclass::update] {
```

```
    // Bring collection of pointers into local store
```

```
    Array<Object*> local(objects, N_OBJECTS);
```

```
    for (int i = 0; i < N_OBJECTS; i++)
```

```
        local[i]->update(); // Invoke virtual methods on objects
```

```
}
```

Portable template data
transfer strategy

OOP on Accelerators (3)

- What was inefficient before?
 - vtable lookup, member data access in update()
- To improve:
 - Remove virtual call + ensure object data in accelerator memory

```
offload {  
    for (int i = 0; i < N_OBJECTS; i++) {  
        Subclass local = *objects[i]; // Bring object into local store  
        local.update();                // Call method directly  
        *objects[i]=local;             // Write object back to global memory  
    }  
}
```

Need to be sure of specific type - sort objects by type to achieve this.

Can improve further by double buffering transfer operations

Overview

- High Level View of Accelerator Programming
- Adapting to Accelerators
- Offload C++ Tutorial
 - Some (short) C++ examples
- **C++ on GPU**
- A Demo of C++ for GPU

C++ Accelerated Massive Parallelism

- Open specification: C++ on GPU
- Initial implementation
 - (for MSVS 11)
- Data parallel code
 - Buffer mechanism to get data on GPU
- No call-graph duplication

```
#include <amp.h>
using namespace concurrency;
void AddArrays(int n, int * pA, int * pB, int * pC)
{
    array_view<int,1> a(n, pA);
    array_view<int,1> b(n, pB);
    array_view<int,1> sum(n, pC);

    parallel_for_each( sum.grid,
        [=](index<1> idx) restrict(direct3d) {
            sum[idx] = a[idx] + b[idx];
        }
    );
}
```

From http://www.danielmoth.com/Blog/cppamp_in_1_or2_or3_slides.pptx

Offload C++ vs OffloadCL

- Offload C++ for PS3
 - Mature industrial optimizing C++ compiler
 - Used in AAA games titles e.g. NASCAR on PS3
- OffloadCL
 - Research prototype compiler
 - Not quite the same C++ dialect + libraries
 - Aims at GPU in addition to Cell
 - Aims at compatibility with recent C++ on GPU work
 - E.g. C++AMP from AMD + Microsoft

Our modifications for OffloadCL

```
GpuPointer<float> c1S(&S[0][0], MAX_HEIGHT*MAX_WIDTH);
GpuPointer<float> c1T(&T[0][0], MAX_HEIGHT*MAX_WIDTH);
GpuPointer<float> c1V(&V[0][0], MAX_HEIGHT*MAX_WIDTH);
GpuPointer<const float> c1L(&L[0][0], MAX_HEIGHT*MAX_WIDTH);
GpuPointer<const float> c1D(&D[0][0], MAX_HEIGHT*MAX_WIDTH);
GpuPointer<const float> c1M(&M[0][0], MAX_HEIGHT*MAX_WIDTH);
```

In OffloadCL, we use the `GpuPointer<>` class to map buffers. On GPU, we need to map all our arrays into buffers.

```
static void OffloadCLUpdateVelocityPerf() {
    const int range = (UniverseHeight-1)*UniverseWidth - UniverseWidth-1;
    c1S.push(); c1T.push(); c1L.push(); c1D.push(); c1V.push();
```

We have to push buffers to GPU (but this only pushes if buffer is on host)

```
GpuPointer<float> mS(c1S), mT(c1T), mV(c1V);
GpuPointer<const float> mM(c1M), mL(c1L), mD(c1D);
```

```
parallel_for (range, [=] (const Point &pt)
```

```
{
    int iGID = pt.GetGlobalID();
    mV[iGID] = mD[iGID]*(mV[iGID] + mL[iGID]*(mS[iGID]-mS[iGID-1]+mT[iGID]-mT[iGID-MAX_WIDTH]));
});
```

We use a lambda-function version of `parallel_for`

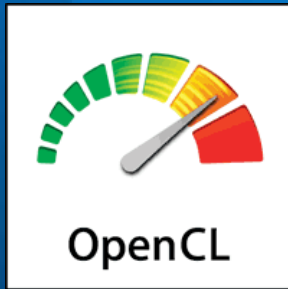
```
c1V.pull();
```

Pull back to host only results we need

```
}
```

OffloadCL / TBB / C++AMP

- Very similar concepts
 - `parallel_for` variants for data parallelism
- Lambda functions are C++11 only
- All array data must be stored in buffer classes
 - `GpuPointer<T>` / `array` / `array_view` classes
 - The name doesn't really matter.
 - OffloadCL will support both C++AMP style and a new upcoming OpenCL C++ standard style



OpenCL

- Low level C99 derived language + API
 - Adds
 - Vector types (SIMD)
 - Memory spaces (__private __global __local __constant)
 - Large set of built-in functions
 - Removes
 - Recursion, Function pointers
 - Mutable global variables
- Multi-vendor standard

OpenCL (2)

- Implicitly parallel
- Dynamic compilation (load, compile, run)
- Portable (CPU, GPU, Cell, DSP, FPGA?)
- Host + Accelerator model
 - Interact via API for
 - Data transfers
 - Program synchronization
 - Compilation
 - Kernel invocation

OpenCL (3)

```
__kernel void square(__global float* input,  
                    __global float* output,  
                    const unsigned count) {  
    int i = get_global_id(0);  
    if(i < count)  
        output[i] = input[i] * input[i];  
}
```

- Simple C/C++ serial computation -->
 - OpenCL kernel + ~100 loc C on host

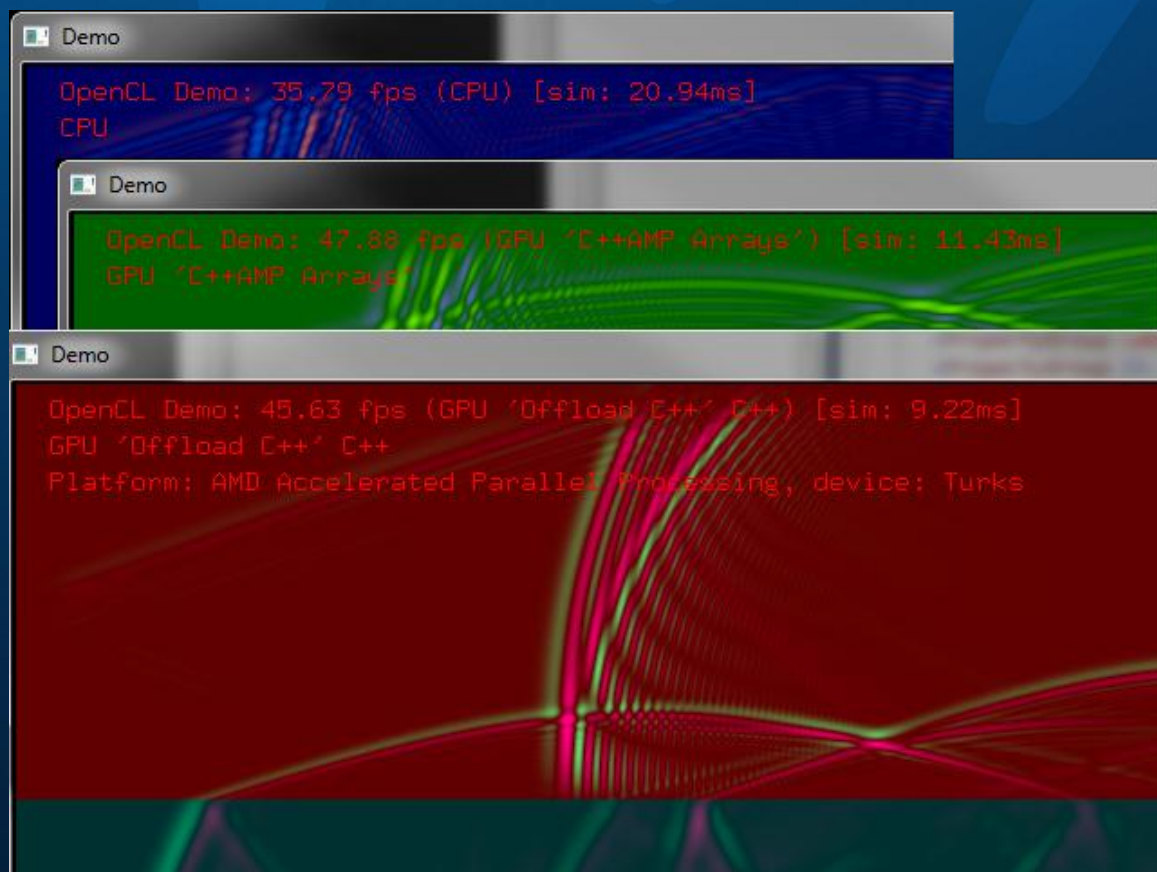
Compiling to OpenCL

- OpenCL is intentionally a low level language
 - Usage requires consideration of many details
- Would rather *compile* to OpenCL
 - Let compiler handle boilerplate
 - Write programs at a higher level
- Feasible
 - Offload C++ on Cell compiles to C99 for SPU/PPU
 - Offload CL compiles C++ to OpenCL 1.1

Overview

- High Level View of Accelerator Programming
- Adapting to Accelerators
- Offload C++ Tutorial
 - Some (short) C++ examples
- C++ on GPU
- **A Demo of C++ for GPU**

Offload CL Demo



- Simulation time
 - Intel Core 2 Duo @2.67Ghz
 - AMD Radeon HD6570
- 20.94ms
 - (CPU)
- 11.43ms
 - (C++ AMP)
- 9.22ms
 - (Offload C++ GPU)

Offload CL Demo

- My laptop
 - Nvidia Geforce 310M
 - (Too low end to run C++AMP / DX11)
 - Intel Core i5 @ 2.4Ghz
 - On “Windows Experience” scores
 - Relative CPU (6.8 vs 6.3)
 - Relative GPU (5.9 vs 6.7)
- Some other variables:
 - OpenCL implementation, etc...

Conclusions + Open Questions...

- Offloading computation to accelerators
 - C++ lambdas + data parallel (the future?)
- Performance portability over many devices?
 - GPU / CPU / Cell
 - Even on more complex code?
 - Are we stuck with GPU inflexibility?

References + Further Reading

- **The Impact of Diverse Memory Architectures on Consumer Software: an Industrial Perspective from the Video Games Domain**
 - Proceedings of the 6th ACM SIGPLAN Workshop on Memory Systems Performance and Correctness, pages 37-42. ACM, 2011.
- **Programming Heterogeneous Multicore Systems using Threading Building Blocks.**
 - Proceedings of the 4th EuroPar Workshop on Highly Parallel Processing on a Chip (HPPC'10), Lecture Notes in Computer Science 6586, pages 117-125. Springer, 2010.
- **Automatic Offloading of C++ for the Cell BE Processor: a Case Study Using Offload.**
 - Proceedings of the 2010 International Workshop on Multi-Core Computing Systems (MuCoCoS'10), pages 901-906. IEEE Computer Society, 2010.
- **Offload - Automating Code Migration to Heterogeneous Multicore Systems.**
 - Proceedings of the 5th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC'10), Lecture Notes in Computer Science 5952, pages 337-352. Springer, 2010.

References + Further Reading

- **Preparing AI for Parallelism: Lessons from NASCAR The Game 2011**
 - Neil Henning <neil@codeplay.com> , Paris Game AI Conference 2011
- **Optimizing Sort Algorithms for the PS3 using Offload™**
 - Pierre-André Saulais <pierre-andre@codeplay.com>
http://codeplay.com/resources/uploaded/Optimizing_Sort_Algorithms_for_the_PS3.pdf.pdf
- **The Codeplay Offload Knowledge Base**
 - FAQs, Tutorials, Case Studies and Performance Tips on Offload C++ for PS3
 - <http://codeplay.com/kb>

Questions + Contact

Find us on the web at
www.codeplay.com

(We're hiring + have internships + doctoral sponsorships)

<http://codeplay.com/jobs>

Email

george@codeplay.com

Any questions?