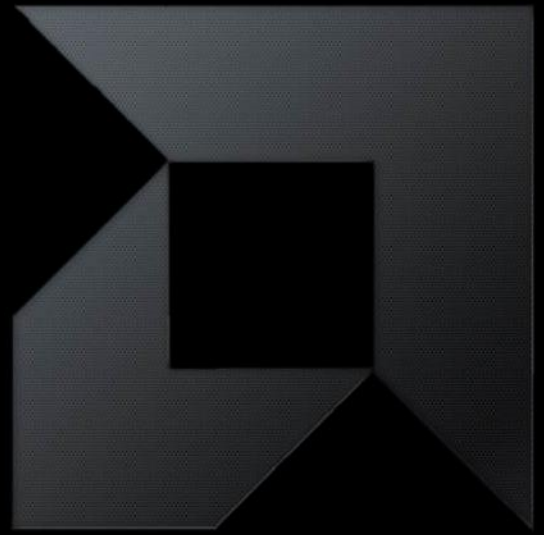
TASKS, FUTURES AND ASYNCHRONOUS PROGRAMMING



AMD

TASK-PARALLELISM

- OpenCL, CUDA, OpenMP (traditionally) and the like are largely data-parallel models
  - Their core unit of parallelism is a data element
  - Parallel dispatches are spread across that data
  - Generally the operation to be performed is closely related if not identical

- Task parallelism is a little more general
  - The tasks may or may not operate on directly-related data
  - Of course, data-parallel executions are, inherently, task parallel at some level

TASKS

- Tasks can be seen as self contained units of execution
  - Usually viewed as scalar and hence distinct from data-parallel executions

- In reality a data-parallel dispatch such as those in OpenCL can be viewed as:
  - a task
  - a tree of tasks
  - a parallel launch of tasks

- It’s all in the way it is interfaced!

- An OpenCL launch is probably best viewed as a single data-parallel task
  - Synchronization and scheduling is at kernel launch granularity.

TASK PARALLEL RUNTIMES

- Designed to efficiently execute tasks as basic units of work
  - Usually implemented on top of some sort of thread pool

- Provide a set of basic functionality:
  - synchronization primitives
  - memory sharing constructs
  - data containers and parallel algorithms

- Work on the basic of queues of tasks
  - Load balancing achieved by work stealing or work donation

MICROSOFT CONCURRENCY RUNTIME (CONCRT)

- A high level concurrency runtime distributed with the visual studio development suite
- Combines with the parallel primitives library to provide a set of useful functionality
- Cooperatively scheduled
  - At the task scheduler level
  - Of course the OS will still preemptively switch the ConcRT threads
  - Uses a work stealing scheduler to balance work
- Uses the latest features of the C++ programming language

WHAT FEATURES? (THIS WILL BE IMPORTANT LATER…)

- C++ really is simplifying the experience for the programmer
  - ConcRT's API is significantly cleaner than earlier task-parallel runtimes as a result

- Some of the earliest adopted C++11 features in compilers are being adopted for this reason:
  - Lambda
  - auto
  - r-value references

ADDITIONS TO THE C++ PROGRAMMING LANGUAGE: LAMBDA FUNCTIONS

- Lambda expressions represent a simplified syntax for constructing closures
  - Code constructs that can be passed around, encapsulating their data requirements

- We could do this in C++ already with function objects
  - Function object syntax is high overhead
  - Short functions are clumsy to write

- Lambdas simplify this

THE PATH TO LAMBDAS

- A function object that multiples an element by a construction-defined constant:

```
struct FunctionObject {
  int m_;
  FunctionObject(int m) : m_(m){}
  int operator()(int a) {
    return a*m_;
  }
}
```

- This could be passed to some sort of parallel for function to square an array:

```
parallel_for_each( vectorIn, vectorOut, FunctionObject(2) );
```

IMPROVING USING LAMBDAS

- The same operation can be achieved using the new lamda syntax more succinctly:

```
int m = 2;
parallel_for( vectorIn, vectorOut, [=](int a){return a*m;} );
```

- Of course that assignment of m was slightly contrived
  - Clearly similar benefits can be achieved on more realistic code

- The lambda syntax may be directly translated by the compiler into something resembling the function object we saw earlier
  - “captured” variables convert to construction-defined fields (by value or reference)
  - Lambda parameters represent the parameters of the () operator overload

- Return type can be inferred in simple cases, or specified using the new post-definition of return type:

```
[=](int a){return a*m;} -> int
```

ADDITIONS TO THE C++ PROGRAMMING LANGUAGE: R-VALUE REFERENCES

- A new type of reference in C++
  - Represented by &&
  - “The best things ever” – Ben Gaster (I’ll let his wife know he said that…)

- Allow the programmer to represent r-values as being different from l-values
  - l-values and r-values follow different rules
  - An l-value persists beyond the defining expression
  - Compilers can optimize r-values more heavily
  - Previously hard to distinguish the two, leading to complicated analysis of code and reduced compositionality

R-VALUE REFERENCES

- Allow the programmer to change behavior based on the type of reference
  - For example, the process of moving data from one vector to another is different from the process of copying

```
class vector {
    vector operator=(vector &rhs) {
        if( size_ != rhs.size_ )
            // resize data_
        memcpy(data_, rhs.data_, size_);
    }
    vector operator=(vector &&rhs) {
        size_ = rhs.size_;
        data_ = rhs.data_;
    }
}
```

Traditional reference assignment needing deep copy

Move assignment where only copying the pointer and size is necessary

BENEFITS TO THE COMPILER AND DEVELOPER

- Note that the information is now encoded in the type system
- The compiler can select the appropriate function to use based on the type
  - And propagate that information through call graphs
- Helps when return value optimization is difficult to achieve
- This reduces the overhead when we are passing C++ objects around and simplifies the set of constructors and assignment operators we need to create
  - More pleasant experience for the programmer!

ADDITIONS TO THE C++ PROGRAMMING LANGUAGE: AUTO AND DECLTYPE

- C++ is a statically typed language and is famously verbose:
  - std::vector<std::vector<int>> *a = new std::vector<std::vector<int>>(10);
- Looking at that we might wonder why the type of a is necessary at all, it can be inferred can't it?
- C++11 repurposes the `auto` keyword (previously this was a storage qualifier similar to `register`)
  - auto a = new std::vector<std::vector<int>>(10);
- Of course with `using namespace` we can decrease it further

DECLTYPE

- Decltype allows us to request the type of an entity for use

- Let's say we want to construct a lambda that takes a version of the current object as a parameter, we might do:
  - `auto l = [](decltype(this) t){ t->doSomething(); return new decltype(this)(*t); } -> decltype(this);`

- So we:
  - Take a pointer to the current type
  - Apply a function to it
  - Return a copy
  - Specify the type returned using the new return type post-declaration operator

MAKING CONCURRENT PROGRAMMING CLEANER USING C++11 FEATURES

- Introducing this to give some idea how C++11 features can be used in a CPU runtime

- Use some of the early C++11 features
  - Lambdas to make easy task construction
  - auto and decltype to streamline typing

- Cooperative threading:
  - Tasks yield to other tasks and completely occupy the worker thread while running

- Note that TBB 4 has many similar features

SIMPLEST CASE TASKING IN CONCRT IS VERY SIMPLE

- Uses the Concurrency namespace
- Create a task from a function object:
  - `auto concRTTask = Concurrency::make_task([] { std::cout<<"Something printed\n"; })`
- Create a task group that manages task execution:
  - `Concurrency::task_group taskPool;`
- The task group schedules tasks onto the thread pool.
- Tasks can dispatch other tasks and wait on the results – the wait is a cooperative threading construct that pushes the task out of the system.
- Run the task in the pool and wait on its completion:
  - `taskPool.run( concRTTask );`
  - `taskPool.wait();`

PARALLEL ALGORITHMS

- parallel_for
  - Repeatedly performs the same task in parallel
  - Parameterises with an iteration value

```
parallel_for(1, 6,
  [](int value) { wstringstream ss; ss << value << L' '; wcout << ss.str(); });
```

- parallel_for_each
  - Parameterises with a value from a container such as a vector or array

```
parallel_for_each(values.begin(), values.end(),
  [](ValueType value) { wstringstream ss; ss << value << L' '; wcout << ss.str(); });
```

- parallel_invoke
  - Executes a set of tasks in parallel and returns only on completion of the entire set

```
parallel_invoke( [&n] { n = twice(n); },
  [&d] { d = twice(d); }, [&s] { s = twice(s); } );
```

PARALLEL CONTAINERS

- Concurrency safe append and read/dequeue operations:
  - `concurrent_vector`
  - `concurrent_queue`

- Combinable class
  - Offers thread-local storage that is combined into a final result
  - Provides reduction functionality

A COUPLE OF EXTENSIONS

- Let’s add a couple of constructs that will help us later
  - The PPL extras in the sample pack contain related ideas if you’re interested in implementations

- PPL tasks rely on task-groups for waiting. What if instead we could wait on the returned value?
  - Futures are representations of types with asynchronously generated values
  - Data is promised to a future by an executing task, often as the return value

FUTURES

- Conceptually:

```
Task t = make_task(functionThatReturnsAnInt);
Future<int> a = t.run();
…
int b = a.get();
```

- That is that if we have some way to asynchronously launch a unit of work, we might wrap its return type so that we can execute in parallel
  - We only synchronize when we need the return value

CONTINUATIONS

- Another useful construct for concurrent programming

- Continuations are units of code (tasks) that are enqueued to execute after something else:

```
Task a = make_task(……);
Task b = make_task(……);
a.continueWith(b);
Future fA = a.enqueue();
```

- Of course, there is a limitation of writing it this way:
  - How do we cleanly get the future resulting from b?

NESTED ENQUEUE

- The following might be one approach

```
Future FunctionA(Task t) {
  // Body of a
  …
  // Task A enqueues its continuation internally, and returns (no wait)
  return t.enqueue();
}
Task a = make_task(……);
Task b = make_task(……);
Future fA = a.enqueue(b); // A takes a parameter of its continuation
Future fB = fA.get();     // The return value from A was a future
int b = fB.get();
```

- Of course, there is a limitation of writing it this way:
  - How do we cleanly get the future resulting from b?

LIMITATIONS OF CONCRT AND THE PPL

- PPL tasks are grouped, but are not inherently data-parallel

- Data-parallel constructs in the PPL are executed as task trees
  - Such structures are not directly suitable for GPU execution
  - We will show some enhancements…

- The continuation and future mechanisms can be encoded on ConcRT tasks
  - They need to be first-class if we want to execute such constructs on heterogeneous systems
  - They can still be **implemented** the ConcRT way on the CPU

- Ben will later explain how we can apply some of these ideas, but first he'll talk about the current state-of-the-art

Disclaimer & Attribution

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. There is no obligation to update or otherwise correct or revise this information. However, we reserve the right to revise this information and to make changes from time to time to the content hereof without obligation to notify any person of such revisions or changes.

NO REPRESENTATIONS OR WARRANTIES ARE MADE WITH RESPECT TO THE CONTENTS HEREOF AND NO RESPONSIBILITY IS ASSUMED FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

ALL IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED. IN NO EVENT WILL ANY LIABILITY TO ANY PERSON BE INCURRED FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.