TOOLS FOR IMPROVING CROSS-PLATFORM SOFTWARE DEVELOPMENT

Eric Kelmelis

28 March 2018
**OVERVIEW**

**BACKGROUND**

Evolution of processing hardware

**CROSS-PLATFORM KERNEL DEVELOPMENT**

Write once, target multiple hardware platforms

**SCHEDULED TASK-GRAPH COMPUTING**

Building scalable heterogenous software
Software Design

» Initial codes are demonstrated as single-core implementations

» This does not provide enough performance for many real world problems

» In the past, to solve larger jobs, parallelism added through MPI
MPI-BASED PARALLELIZATION

» By using MPI, parallelism between nodes becomes possible
» Larger problems can be solved
» Processing time is reduced

MPI is designed for inter-node communication
In 2001, there were few processing options offered. Since then, many new architectures have been introduced.

**Platform Experience Includes:** Multicore microprocessors, computer clusters, graphics processing units (GPUs), Xeon Phis, field-programmable gate arrays (FPGAs), digital signal processors (DSPs), hybrid SoCs, Cell Processor, etc.
Developers have numerous processing architectures to choose from:
- Multicore (CPU: x86, ARM, Power, TILE)
- Vector Processor/Massive Multicore (GPU: NVIDIA, Xeon Phi, ATI)
- Programmable Logic (FPGA: Xilinx, Altera)

These architectures are also being combined into hybrid systems
(NVIDIA Tegra, Xilinx Zynq, Intel, etc.)
Modern Computing Node

Computing nodes from large HPCs to small embedded/mobile device are becoming hybrid systems.
The days of writing software for only single-core Intel chips are past

Software must be...
  ... parallel
  ... scalable
  ... cross-platform
  ... hybrid
  ... future compatible
Varying computing platforms and integrated hybrid systems allow for faster, more power-efficient computations

*but...*

...this comes at the price of programming complexity

**Two Approaches...**

1. A team of experts in using each of these new devices
2. Tools that allow programmers to effectively use them without the need to learn and re-learn specialized skills
**Approaches to Using Hybrid Systems**

### Libraries

Experts build optimized software → Users build applications on these tools

![Diagram showing library call and compilation process]

#### Operation Scheduling (Kernels)

Tools analyze software at compile time and efficiently order operations

![Diagram showing code analysis and operation scheduling]

### Directives

Developers annotate their code with hints for the compiler

```c
#pragma less_power
capture_frame();
detect_objects_lightweight(&object_detected);
if(object_detected)
{
    #pragma more_performance
    detect_objects_precise(&object_detected);
}
```

#### Task Scheduling (Applications)

Tasks are scheduled at runtime based on system characteristics

![Diagram showing task scheduling process]
**Efficient Software Required at Two Levels**

**Computational Kernels**
- Must be able to write efficient execution units
- Often require multiple versions to work with multiple devices

**Scalable Applications**
- Application must be able to scale to utilize available hardware
- Often multiple deployment platforms or scenarios to consider
WRITING CROSS-PLATFORM KERNELS
Leveraging industry-standard compilers and linking tools
» Avoid duplication of effort and simplify long-term maintenance
» Allow users to only rewrite the code portions that are problematic
BUILDING PORTABLE COMPUTATIONAL KERNELS

User-Developed Code

DFG Builder

DFG Compiler

EMP Tools

Standard Toolchain

DFG Compiler

Standard Toolchain

Standard Toolchain

DFG Compiler

Standard Toolchain

Standard Toolchain

DFG Compiler

Standard Toolchain

Standard Toolchain

DFG Compiler

Standard Toolchain

Standard Toolchain

DFG Compiler

Standard Toolchain

Standard Toolchain

DFG Compiler

Standard Toolchain

Standard Toolchain

DFG Compiler

Standard Toolchain

Standard Toolchain

DFG Compiler

Standard Toolchain

Standard Toolchain
**Using an Intermediate DFG Approach**

**Front-End**
- Generate a dataflow graph (DFG) representation from a C++ program
  - Function to compute, e.g., FFT

**Back-End**
- Schedule dataflow graph representation on parallel hardware
» Performing code analysis/pre-processing before compilation will allow us to translate user directives and placeholder types into code that is compilable by industry-standard C++ compilers

» Leverage existing standard toolchains to shorten development time

```cpp
template <typename T>
__dataflow__
auto FFT(T t)
{
    // Get input size
    auto size = Size(t);

    // Slice and Recurse
    auto even = FFT(t(0, 2, size - 1));
    auto odd = FFT(t(1, 2, size - 1));

    // Make Twiddle Factors
    MakeRange<complex<float> >(0, 1, size / 2 - 1);

    // twid(r) = exp( -2.0j * pi * r / size )
    auto twid = Transform(
        std::exp(-2.0f*std::pi<float>()*1f/size),  // Expression
        complexRange);  // One argument

    // Perform remaining FFT math
    auto todd = odd * twid

    if(size <= one)
        return Transform(TypeCast<complex<float>>, t)
    else
        return Concatenate(even + todd, even - todd);
}
```
EM Photonics’ EDSL looks similar to standard C++ code, making it intuitive for the user.

In the future, automated tools can be created to take properly formatted C++ code directly into our toolchain.
TARGETING DIFFERENT BACK-ENDS

Running a function on the CPU

```
std::array<int, 4> vin1{ 2, 3, 4, 5 };
std::array<int, 4> vin2{ 1, 2, 3, 4 };

using MT = decltype(vin1);

auto minusDFG = dataflow::MakeDFG(MinusTest<ArgType<MT, 0>, ArgType<MT, 1>>, "MinusTestVector");
auto compiledMinus = CompileDFG<decltype(vref)>(MinusTest<ArgType<MT, 0>, ArgType<MT, 1>>, minusDFG, "MinusTestVector");
auto vres = compiledMinus(vin1, vin2);
```

Running a function on the GPU

```
std::array<int, 4> vin1{ 2, 3, 4, 5 };
std::array<int, 4> vin2{ 1, 2, 3, 4 };

using MT = decltype(vin1);

auto minusDFG = dataflow::MakeDFG(MinusTest<ArgType<MT, 0>, ArgType<MT, 1>>, "MinusTestVector");
auto compiledMinus = CompileDFG<decltype(vref)>(MinusTest<ArgType<MT, 0>, ArgType<MT, 1>>, minusDFG, "MinusTestVector");
auto vres = compiledMinus(vin1, vin2);
```

» Changing to the GPU backend requires the user to make minimal changes
» Memory management and copying is handled transparently
» No need to use the CUDA driver or runtime API or write any kernels
We ran our analysis on our base FFT example

370 nodes were divided into 23 work sets

Work units within a set can be scheduled independently

What if we unrolled the recursion?
Goal: schedule several recursive stack frames worth of work at once
   For problems that recurse a lot, this can result in the greater ability to schedule parallel work

Test: We unrolled our FFT example one additional time
   Easily accomplished by setting a depth parameter for compile-time unrolling
   Can also be done via dataflow graph mutation for reduced compile times

Result: 3610 nodes divided into 33 work sets
   Significantly more operations per work set
   Observed trend based on additional unrolling depths: 9-10x more nodes and +10 work sets for each unroll
   Unroll more for highly-parallel target platforms

```cpp
using FftType = ArgType<array<float, 1024>, 0>;
const int unrollDepth = 2;
auto FFTDFG = MakeDFG(FFT<unrollDepth>::call<FftType>, "FFT");
```
We benchmarked our Embedded Domain-Specific Language (EDSL) FFT example code against a reference implementation written in C++

- Compiled with Clang in release with best optimization flags for the C++ reference
- Care was taken with the C++ reference to avoid introducing easily-avoided, unnecessary copies

Achieved performance parity
- Optimizations can still be performed
- Our EDSL version also contains full dependency information that can be exploited for automated parallelism
SCALABLE SOFTWARE FOR HYBRID COMPUTERS
**Work Queue Approach to Scalability**

### Traditional Work Partitioning

- Job is broken up into pieces and distributed to processors in the system.

### Modern Problem Partitioning

- Create a queue of work that can be dynamically distributed to processors.

---

**Job**

**CPU**

**CPU**

**CPU**
A task is a pure function which receives inputs and delivers outputs according to a task graph.

Details of task invocation can be abstracted from the programmer.

Tasks can be developed, tested, and characterized as independent units.

Synchronization between tasks is defined purely by connections in the task graph.
COMPONENTS OF A TASK-BASED APPLICATION

Task Definitions
- Name
- Inputs
- Outputs

Graph

Schedule

Schedule

Copyright © 2018 EM Photonics – EM Photonics Proprietary
IMPLEMENTING TASKS: BUILDING MODULES

Task Description

Kernel K1 for T1
Kernel K1 for T2
Kernel K1 for T3
Kernel K1 for T4

Module Builder

Module 1
- Constants
- T1:K1
- T1:K2
- T1:K3

Module 2
- Constants
- T2:K1
- T2:K2

Module 3
- Constants
- T3:K1

Module 4
- Constants
- T4:K1
- T4:K2

Module Builder

Compilers

T1:K1
T1:K2
T1:K3
T2:K1
T2:K2
T3:K1
T4:K1
T4:K2

Programmer Defined
Existing Tools
New Tools
Implementing Tasks: Module

Develop computational functionality for each targeted processing device

// Call CUDA kernel
dim3 dimGrid( NX/TILE_WIDTH, NZ/TILE_WIDTH );
dim3 dimBlock( TILE_WIDTH, TILE_WIDTH );
primitive_variables_ker <<dimGrid, dimBlock>>>
    (p_cons, p_rho, p_vx, p_vy, p_vz,
     p_p, p_bx, p_by, p_bz, p_psic);

// Call CPU code
primitive_variables_cpu (p_cons, p_rho, p_vx, p_vy,
                        p_vz, p_p, p_bx, p_by, p_bz, p_psic);

// Call Intel Xeon Phi code
primitive_variables_mic (p_cons, p_rho, p_vx, p_vy,
                        p_vz, p_p, p_bx, p_by, p_bz, p_psic);

Module

UpdateVariables()

- CPU Kernel
- CUDA Kernel
- OpenCL Kernel
- Intel MIC Kernel
- [Future architectures]
» Analyze dependencies between tasks
» Dynamically determine best hardware for each task
» Manage data movement between devices
» Live profiling results to better schedule future tasks
**Task Queues & Worker Pools**

- **Task queue** holds tasks to be executed
- **Worker pool** manages the hardware devices available for computation

**Scheduler** assigns tasks to free devices in the worker pool, choosing best device based on worker load, data locality, and performance characteristics.
EXAMPLE: SORAD TASK GRAPH
SAMPLE TASK INTERFACE: getchemistry

interfacc:
  name: getchemistry
  source: chemistry
  type: CPU
  dimensions: [ ncolumns, nlevels ]
  inputs:
    - name: pressure
      type: real
      shape: [nlevels+2, ncolumns]
    - name: temperature
      type: real
      shape: [nlevels+2, ncolumns]
    - name: humidity
      type: real
      shape: [nlevels+2, ncolumns]
    - name: ozone
      type: real
      shape: [nlevels+2, ncolumns]
  outputs:
    - name: dpressure
      type: real
      shape: [ nlevels+2, ncolumns ]
    - name: scaling
      type: real
      shape: [ nlevels+2, ncolumns ]
    - name: water
      type: real
      shape: [ nlevels+2, ncolumns ]
    - name: scaled_ozone
      type: real
      shape: [ nlevels+2, ncolumns ]
CODE TO BUILD SORAD TASK GRAPH (SUBSET)

```cpp
// Connect values for cloudcover
sorad.add_link(fcloud, "fcloud", cloudcover_node, "fclld");
sorad.add_link(layers, "layers", cloudcover_node, "clb");

// Connect values for chemistry
sorad.add_link(pressure, "pressure", chemistry_node, "pressure");
sorad.add_link(temperature, "temperature", chemistry_node, "temperature");
sorad.add_link(humidity, "humidity", chemistry_node, "humidity");
sorad.add_link(ozone, "ozone", chemistry_node, "ozone");

// Connect values for absorption
sorad.add_link(co2, "co2", absorption_node, "co2");
sorad.add_link(chemistry_node, "scaling", absorption_node, "pscale");
sorad.add_link(cosz, "cosz", absorption_node, "cosz");
sorad.add_link(pressure, "pressure", absorption_node, "pressure");
sorad.add_link(chemistry_node, "wh", absorption_node, "swater");
sorad.add_link(cloudcover_node, "cloudcover", cc, "cloudcover");
sorad.add_link(cloudcover_node, "cloudtop", cloudtop, "cloudtop");
sorad.add_link(chemistry_node, "dpressure", dpressure, "dpressure");
sorad.add_link(chemistry_node, "oh", oh, "oh");
sorad.add_link(absorption_node, "absorption", absorp, "absorption");
sorad.connected();

go::runtime_graph run_sorad = sorad.build_runtime(extents{nlevels, ncolumns, 8, 5, 3, 10, 14, 1, 2, 3, 4, 5, 35});
run_sorad.set_output_callback("cloudcover", ccOutWrapper, nullptr);
run_sorad.set_output_callback("cloudtop", cloudtopOutWrapper, nullptr);
run_sorad.set_output_callback("dpressure", dpressureOutWrapper, nullptr);
run_sorad.set_output_callback("oh", ohOutWrapper, nullptr);
run_sorad.set_output_callback("absorption", absorpOutWrapper, nullptr);
```
Dynamic scheduled versions took about 1 second to handle data allocation
Task-Based SORAD

» SORAD broken down into a component tasks and connected in a graph
» **Advantages:** Easier to maintain, test, and extend; Can be scheduled

Statically Scheduled

» An upfront static schedule can be built to execute the SORAD task graph
» **Advantages:** Better performance than regular SORAD; Better portability

Dynamically Scheduled

» Could be scheduled with any framework (e.g. Intel’s TBB, etc.)
» Not enough work to overcome dynamic scheduling overhead
**POINT.solve.5 Selection**

**point.solve.X**
- Iterative method for solving sparse matrix product
  - *point.solve.5*: Operated on 5 x 5 square matrix
  - This was the version that was the focus of this effort
  - *point.solve.n*: Operator on N x N square matrix

**Breaking into Tasks**
- First the monolithic code needed to be broken into tasks
  - Each task has a single responsibility within the code
- Tasks were then wrapped and connected into a task graph which could be executed
POINT_SOLVE_5 TASK GRAPH
MULTIPLE TASK VERSIONS

*point_solve_5* subtask

- **Serial Version**: Used as reference baseline for testing
- **C++ TBB**: Utilizing modern parallelization technology
- **C/FORTRAN + OpenMP**: Able to call existing/developed kernels
» Each PointSolve task calls an underlying subgraph of tasks which perform the full operation
» Handles merging of output
Each PointSolve task calls an underlying subgraph of tasks which perform the full operation

- Handles merging of output
- Each task node marked with data chunk bounds
POINT_SOLVE_5 — SUBGRAPH

» This handles connecting defined tasks for a given chunk of data

Input Data
Input Data Tag
Task
Bookkeeper
Output
CPU Usage: Point Solve 5 with all data

» This trace illustrates the application’s task parallelism separate from the data parallelism

» Where possible, the task-based scheduling framework overlapped task execution
  » The non-colored areas represent the idle time where additional tasks could not be scheduled due to dependencies between tasks
This trace illustrates the application’s task parallelism combined with the data parallelism achieved through the chunking of data.

Where possible, the TBSF overlaps task execution

Much of the idle time seen in the previous trace has been filled in by tasks operating on non-dependent chunks of data.

There is still idle time seen at the end of the iteration

Iteration support now added and being tested.
A comparison was done between the tasks scheduled through the framework and the same tasks utilizing OpenMP for scheduling on the CPU. A speed increase in average execution time was seen in the framework scheduled version for both sets of reference data.
SCALABLE HYBRID COMPUTING

Compute Task
- UpdateVariables()
- Details

CPU Code

GPU Code

HPC Hardware
- Scale performance with hardware upgrades

Diagnostics
- Node 1: OK
- Node 2: OK
- Node 3: OK

Performance

Real time hardware monitoring and fault tolerance

Improve code by added new high performance kernels

45
Copyright © 2018 EM Photonics – EM Photonics Proprietary
Unified Strategy
**EM Photonics Tool Strategy**

**Tools for cross-platform computational kernel development**

- User-Developed Code
  - DFG Builder
  - DFG Compiler

**Accelerated math libraries**
- **CULA**: NVIDIA-only
- **Affinimath**: Cross-platform

**Tools for power efficient programming**
- Dynamic task-based scheduling for heterogeneous computing systems
  - Name
  - Inputs
  - Outputs

- **PowerMP Application**
  - **PowerMP Runtime**
  - Linux Kernel
    - hotplug
    - cpuidle
    - cpufreq
  - Additional PowerMP interfaces
  - Hardware Drivers & BSP
COMBINING TOOLS

Cross Platform Kernels + Dynamic Scheduling + “Off the Shelf” Libraries

Diagram showing the interaction between tasks and library calls.
CONCLUSION

Building Modern Software

» Computing landscape has changed; platform options are diverse
» Tools must account for mixed device deployments

Cross-Platform Kernels

» Developing a tool that allows for cross-platform software development
» Write a single computational kernel and build for different processors

Task-Based Scheduler

» Break application down into a series of interconnected tasks
» Use automated tools to partition those tasks over heterogeneous hardware