A Productive Framework for Generating High Performance, Portable, Scalable Applications for Heterogeneous computing

Wen-mei W. Hwu

with

Tom Jablin, Chris Rodrigues, Liwen Chang, Steven ShengZhou Wu, Abdul Dakkak

CCoE, University of Illinois at Urbana-Champaign
4,224 Kepler GPUs in Blue Waters

- **NAMD**
  - 100 million atom benchmark with Langevin dynamics and PME once every 4 steps, from launch to finish, all I/O included
  - 768 nodes, Kepler+Interlagos is 3.9X faster over Interlagos-only
  - 768 nodes, XK7 is 1.8X XE6

- **Chroma**
  - Lattice QCD parameters: grid size of $48^3 \times 512$ running at the physical values of the quark masses
  - 768 nodes, Kepler+Interlagos is 4.9X faster over Interlagos-only
  - 768 nodes, XK7 is 2.4X XE6

- **QMCPACK**
  - Full run Graphite 4x4x1 (256 electrons), QMC followed by VMC
  - 700 nodes, Kepler+Interlagos is 4.9X faster over Interlagos-only
  - 700 nodes, XK7 is 2.7X XE6
Two Current Challenges

• At scale use of GPUs
  – Communication costs dominate beyond 2048 nodes
  – E.g., NAMD Limited by PME
  – Insufficient computation work

• Programming Efforts
  – This talk

Blue Waters K7 Nodes NAMD Strong Scaling – 100M Atoms

- CPU
- CPU+GPU

SC13
Writing efficient parallel code is complicated.

Tools can provide focused help or broad help

Planning how to execute an algorithm Implementing the plan

- Choose data structures
- Map work/data into tasks
- Schedule tasks to threads
- Memory allocation
- Data movement
- Pointer operations
- Index arithmetic
- Kernel dimensions
- Thread ID arithmetic
- Synchronization
- Temporary data structures

GMAC
DL
Triolet, X10, Chappel, Nesl, DeLite, Par4All
OpenACC/C++AMP/Thrust
Tangram

SC13
## Levels of GPU Programming Languages

**Current generation**
- CUDA, OpenCL, DirectCompute

**Next generation**
- OpenACC, C++AMP, Thrust, Bolt
  - Simplifies data movement, kernel details and kernel launch
  - Same GPU execution model (but less boilerplate)

**Prototype & in development**
- X10, Chapel, Nesl, Delite, Par4all, Triolet...
  - Implementation manages GPU threading and synchronization invisibly to user
Where should the smarts be for Parallelization and Optimization?

- **General-purpose language + parallelizing compiler**
  - Requires a very intelligent compiler
  - Limited success outside of regular, static array algorithms

- **Domain-specific language + domain-specific compiler**
  - Simplify compiler’s job with language restrictions and extensions
  - Requires customizing a compiler for each domain

- **Parallel meta-library + general-purpose compiler**
  - Library embodies parallelization decisions
  - Uses a general-purpose compiler infrastructure
  - Extensible—just add library functions
  - Historically, library is the area with the most success in parallel computing
Triolet – Composable Library-Driven Parallelization

• EDSL-style library: build, then interpret program packages
• Allows library to collect multiple parallel operations and create an optimized arrangement
  – Lazy evaluation and aggressive inlining
  – Loop fusion to reduce communication and memory traffic
  – Array partitioning to reduce communication overhead
  – Library source-guided parallelism optimization of sequential, shared-memory, and/or distributed algorithms
• Loop-building decisions use information that is often known at compile time
  – By adding typing to Python
def correlation(xs, ys):
    scores = (f(x, y) for x in xs for y in ys)
    return histogram(100, par(scores))
Triolet Compiler
Intermediate Representation

• List comprehension and par build a package containing
  1. Desired parallelism
  2. Input data structures
  3. Loop body
     for each loop level
• Loop structure and parallelism annotations are *statically known*

```latex
correlation \text{xs} \ ys = \begin{align*}
\text{let } i = & \text{IdxNest HintPar} \\
& (\text{arraySlice } \text{xs}) \\
& (\lambda x. \text{IdxFlat HintSeq} \\
& (\text{arraySlice } \text{ys}) \\
& (\lambda y. f x y) ) \\
\end{align*} \\
\text{in histogram 100 } i
```

```
Outer loop
 Inner loop
Body
```
Triolet Meta-Library

- Compiler inlines histogram
- histogram has code paths for handling different loop structures
- Loop structure is known, so compiler can remove unused code paths

```
correlation xs ys =
  case IdxNest HintPar
    (arraySlice xs)
    (λx. IdxFlat HintSeq
      (arraySlice ys)
      (λy. f x y ))
  of IdxNest parhint input body.
  case parhint
    of HintSeq. code for sequential nested histogram
    HintPar. parReduce input
      (λchunk. seqHistogram 100 body chunk)
  of IdxFlat parhint input body. code for flat histogram
```
Example: Correlation Code

- Result is an outer loop specialized for this application
- Process continues for inner loop

\[
\text{correlation } xs \ ys = \parReduce \\
(\text{arraySlice } xs) \\
(\lambda \text{chunk. seqHistogram } 100 \\
(\lambda x. \text{IdxFlat HintSeq } \\
(\text{arraySlice } ys) \\
(\lambda y. f \ x \ y ) \\
\text{chunk})
\]

Parallel reduction; each task processes a chunk of \(xs\)
Task computes a sequential histogram
Inner loop
Body
Cluster-Parallel Performance and Scalability

- **Triolet** delivers large speedup over sequential C
- On par with manually parallelized C for computation-bound code (left)
- Beats similar high-level interfaces on communication-intensive code (right)

A parallel algorithm framework for solving linear recurrence problems
- Scan, tridiagonal matrix solvers, bidiagonal matrix solvers, recursive filters, ...
- Many specialized algorithms in literature

Linear Recurrence - very important for converting sequential algorithms into parallel algorithms
Tangrams Linear Optimizations

• Library operations to simplify application tiling and communication
  – Auto-tuning for each target architecture
• Unified Tiling Space
  – Simple interface for register tiling, scratchpad tiling, and cache tiling
  – Automatic thread fusion as enabler
• Communication Optimization
  – Choice/hybrid of three major types of algorithms
  – Computation vs. communication tradeoff
Linear Recurrence
Algorithms and Communication

Brent-Kung Circuit
Kogge-Stone Circuit
Group Structured
### Tangram Initial Results

#### Prefix scan on Fermi (C2050)

<table>
<thead>
<tr>
<th>Throughput (billions of samples per second)</th>
<th>Problem Size (millions of samples, data type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-32bit</td>
<td>0</td>
</tr>
<tr>
<td>2-32bit</td>
<td>2</td>
</tr>
<tr>
<td>4-32bit</td>
<td>4</td>
</tr>
<tr>
<td>8-32bit</td>
<td>6</td>
</tr>
<tr>
<td>16-32bit</td>
<td>8</td>
</tr>
<tr>
<td>32-32bit</td>
<td>10</td>
</tr>
<tr>
<td>64-32bit</td>
<td>12</td>
</tr>
<tr>
<td>128-32bit</td>
<td>14</td>
</tr>
<tr>
<td>1-64bit</td>
<td>0</td>
</tr>
<tr>
<td>2-64bit</td>
<td>2</td>
</tr>
<tr>
<td>4-64bit</td>
<td>4</td>
</tr>
<tr>
<td>8-64bit</td>
<td>6</td>
</tr>
<tr>
<td>16-64bit</td>
<td>8</td>
</tr>
<tr>
<td>32-64bit</td>
<td>10</td>
</tr>
<tr>
<td>64-64bit</td>
<td>12</td>
</tr>
</tbody>
</table>

#### Prefix scan on Kepler (Titan)

<table>
<thead>
<tr>
<th>Throughput (billions of samples per second)</th>
<th>Problem Size (millions of samples)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>32</td>
<td>10</td>
</tr>
<tr>
<td>64</td>
<td>12</td>
</tr>
</tbody>
</table>

#### IIR Filter on both GPUs

<table>
<thead>
<tr>
<th>Throughput (billions of samples per second)</th>
<th>Order of IIR Filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

#### Tridiagonal solver on both GPUs

<table>
<thead>
<tr>
<th>Throughput (billions of equations per second)</th>
<th>Problem Size (millions of equations)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>2</td>
</tr>
<tr>
<td>3000</td>
<td>4</td>
</tr>
<tr>
<td>4000</td>
<td>6</td>
</tr>
<tr>
<td>5000</td>
<td>8</td>
</tr>
<tr>
<td>6000</td>
<td>10</td>
</tr>
</tbody>
</table>

---

- **StreamScan-Reported**
- **Proposed-Tuned**
- **StreamScan-Tuned**
- **Thrust-1.5**

- **Problem Size (millions of samples, data type)**
- **1-32bit 8-32bit 64-32bit 1-64bit 8-64bit 64-64bit**

- **Throughput (billions of samples per second)**
- **Problem Size (millions of samples)**

- **Prefix scan on Fermi (C2050)**
- **Prefix scan on Kepler (Titan)**

- **IIR Filter on both GPUs**

- **Tridiagonal solver on both GPUs**
Next Steps

• Triolet released as an open source project
  – Develop additional Triolet library functions and their implementations for important application domains
  – Develop Triolet library functions for GPU clusters

• Publish and release Tangram
  – Current tridiagonal solver in CUSPARSE is from UIUC based on the Tangram work
  – Integration with Triolet
THANK YOU!