Triolet C++/Python – productive programming in heterogeneous parallel systems

Wen-mei Hwu

Sanders AMD Chair, ECE and CS
University of Illinois, Urbana-Champaign

CTO, MulticoreWare
Agenda

• Performance portability of imperative parallel programming
  – OpenCL

• Algorithm selection, scalability, and efficiency of intentional parallel programming
  – Triolet C++
  – Triolet Python
Essential work in writing efficient parallel code.

Planning how to execute an algorithm

• Distribute computation across
  • cores,
  • hardware threads, and
  • vector processing elements

• Distribute data across
  • discrete GPUs or
  • clusters

• Orchestrate communication for
  • reductions,
  • variable-size list creation,
  • stencils, etc.

Implementing the plan

• Rearrange data for locality
  • Fuse or split loops
  • Map loop iterations onto hardware

• Allocate memory
• Partition data
• Insert data movement code

• Reduction trees
• Array packing
• Boundary cell communication
Current State of Programming Heterogeneous Systems

CPU
Multicore
Xeon Phi
GPU
FPGA

ACS Productivity Workshop, 2014
Current State of Programming Heterogeneous Systems

C/FORTRAN

CPU

Multicore

Xeon Phi

GPU

FPGA
Current State of Programming Heterogeneous Systems

C/FORTRAN

+ Directives (OpenMP/ TBB)

CPU
Multicore
Xeon Phi
GPU
FPGA
Current State of Programming Heterogeneous Systems

C/FORTRAN

+ Directives (OpenMP/ TBB)

Multicore

Xeon Phi

+ SIMD Intrinsics

GPU

FPGA

ACS Productivity Workshop, 2014
Current State of Programming Heterogeneous Systems

C/FORTRAN
- CPU
- Multicore
- Xeon Phi

OpenCL
- GPU
- FPGA

+ Directives (OpenMP/ TBB)
+ SIMD Intrinsics

ACS Productivity Workshop, 2014
Current State of Programming Heterogeneous Systems

C/FORTRAN

- CPU
- Multicore
- Xeon Phi

OpenCL

- GPU

Verilog/VHDL

ACS Productivity Workshop, 2014
Current State of Programming Heterogeneous Systems

Programming heterogeneous systems requires too many versions of code!

C/FORTRAN

- Directives (OpenMP/ TBB)

Multicore

Xeon Phi

- SIMD Intrinsics

CPU

GPU

OpenCL

Verilog/VHDL

FPGA

ACS Productivity Workshop, 2014
Productivity in Programming Heterogeneous Systems

Step #1: Keep only one of the versions and use portability tools to generate the others.
MxPA: Overview

- Optimizes scheduling of work-item execution for **locality** and **vectorization** for the hardware.

- Locality-centric compile-time scheduling selects between:
  - **BFO (Breadth-First Order):** issue each memory access for all work-items first
  - **DFO (Depth-First Order):** issue all memory accesses for a single work-item first

- Kernel fusion run-time scheduling reduces memory traffic for common data flow between kernels

- Dynamic vectorization maintains vector execution in spite of apparent control flow divergence
MxPA: Locality-centric Scheduling

Dependency in executing OpenCL kernels

An example OpenCL code

```
CLKernel() {
    // e.g. SOA
    for (i = 0..N) {
        .. = A[c0*i + wid];
    }
    barrier();
    // e.g. AOS
    for (j = 0..M) {
        .. = B[c1*wid + j];
    }
}
```

MXPA-translated code

```
CLKernel_MXPA() {
    // BFO
    for (i = 0..N) {
        .. = A[c0*i + wid];
    }

    // DFO
    for (wid = 0..LS) {
        for (j = 0..M) {
            .. = B[c1*wid + j];
        }
    }
}
```
MxPA: Dynamic Vectorization of Control Divergent Loops

Effective for Sparse methods and graph algorithms.
MxPA Results: Comparison to Industry

**Comparison to Industry**

**LC**  
- **AMD**

+ Locality improves  
= Locality stays the same  
- Locality gets worse

**Speedup** (normalized – higher is better)

---

**Comparison to Intel**

**LC**  
- **Intel**

+ Locality improves  
= Locality stays the same  
- Locality gets worse

**Speedup** (normalized – higher is better)

---

ACS Productivity Workshop, 2014
MxPA Results: Comparison to Industry

<table>
<thead>
<tr>
<th>Metric</th>
<th>LC/AMD</th>
<th>LC/Intel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>3.20x</td>
<td>1.62x</td>
</tr>
<tr>
<td>L1 Data Cache Misses</td>
<td>0.12x</td>
<td>0.36x</td>
</tr>
<tr>
<td>Data TLB Misses</td>
<td>0.23x</td>
<td>0.33x</td>
</tr>
<tr>
<td>LLC Misses</td>
<td>0.91x</td>
<td>0.97x</td>
</tr>
</tbody>
</table>
MxPA Case Study: MOCFE-Bone

Input Configurations:
Nodes = 1, Groups = 25, Angles = 128, MeshScale=10 (Elements=10^3)
HIGH-LEVEL INTERFACE
Who does the hard work in parallelization?

• General-purpose language + parallelizing compiler
  – Requires a very intelligent compiler
  – Limited success outside of regular array algorithms

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

• Triolet - Parallel library + optimizing compiler
  – Library makes parallelization decisions
  – Uses a rich transformation, library aware compiler
  – Extensible—just add library functions
Step #2: Use a higher-level algorithm representation.
Triolet C++

• Goal – design and implement a simple, user-friendly interface for communicating the intended data access and computation patterns to a library-aware C++ compiler

• Technical approach
  – Data objects allow changes to logical data organization and content without touching storage
  – Computations based on map and reduce
  – Aggressive algorithm selection and auto-tuning through hardware specific library implementation
  – Compiler code synthesis technology for target hardware built on MxPA
Triolet/C++ Convolution Code

```cpp
std::vector<int> 2dconvolution(const std::vector<int>& input,
    int x_size, int y_size,
    const std::vector<float>& kernel)
{
    auto input_co = make_matrix<int, 2>(input, x_size, y_size);
}
```

Ask the compiler to treat the C++ input array as a 2D matrix object whose dimensions are given by arguments x_size and y_size.
Triolet/C++ Convolution Code

```cpp
std::vector<int> 2dconvolution(const std::vector<int>& input,
                          int x_size, int y_size,
                          const std::vector<float>& kernel)
{
  auto input_co = make_matrix<int, 2>(input, x_size, y_size);
  auto kernel_co = make_small_vector<int>(kernel);

  Treat the C++ kernel array as a small 1D vector in preparation for dot product.
```
Triolet/C++ Convolution Code

```cpp
std::vector<int> 2dconvolution(const std::vector<int>& input,
                               int x_size, int y_size,
                               const std::vector<float>& kernel)
{
  auto input_co = make_matrix<int, 2>(input, x_size, y_size);
  auto kernel_co = make_small_vector<int>(kernel);
  auto stencil_co = make_stencil_transform<9,9>(input_co);
}
```

Conceptually form a 2D $x_{\text{size}}$ by $y_{\text{size}}$ matrix whose elements are the 9x9 neighbor stencils around the original `input_co` elements.
std::vector<int> 2dconvolution(const std::vector<int>& input, int x_size, int y_size, const std::vector<float>& kernel)
{
    auto input_co = make_matrix<int, 2>(input, x_size, y_size);
    auto kernel_co = make_small_vector<int>(kernel);
    auto stencil_co = make_stencil_transform<9,9>(input_co);
    auto const_co = make_const_transform(kernel_co, x_size*y_size);
}

Conceptually replicate \texttt{kernel\_co} into an \texttt{x\_size} by \texttt{y\_size} stencil matrix
Triolet/C++ Convolution Code

```cpp
std::vector<int> 2dconvolution(const std::vector<int>& input,
    int x_size, int y_size,
    const std::vector<float>& kernel)
{
    auto input_co = make_matrix<int, 2>(input, x_size, y_size);
    auto kernel_co = make_small_vector<int>(kernel);
    auto stencil_co = make_stencil_transform<int, int>(input_co);
    auto const_co = make_const_transform(kernel_co, x_size * y_size);
    auto zip_co = make_zip_transform(stencil_co, const_co);
}

Conceptually form an x_size by y_size matrix where each element is a tuple of one stencil_co element and one const_co element.

ACS Productivity Workshop, 2014
Convolution Example
(5x5 Zipped Matrix)

zip(filters, stencils)
Triolet/C++ Convolution Code

```cpp
std::vector<int> 2dconvolution(const std::vector<int>& input,
   int x_size, int y_size,
   const std::vector<float>& kernel)
{
  auto input_co = make_matrix<int, 2>(input, x_size, y_size);
  auto kernel_co = make_small_vector<int>(kernel);
  auto stencil_co = make_stencil_transform<9,9>(input_co);
  auto const_co = make_const_transform(kernel_co, x_size * y_size);
  auto zip_co = make_zip_transform(stencil_co, const_co);
  auto map_co = make_map_transform(zip_co,map_operation<mul_op>());
}
```

Perform pair-wise multiplication onto all zipped elements
Triolet/C++ Convolution Code

```cpp
std::vector<int> 2dconvolution(const std::vector<int>& input,
   int x_size, int y_size,
   const std::vector<float>& kernel)
{
    auto input_co = make_matrix<int, 2>(input, x_size, y_size);
    auto kernel_co = make_small_vector<int>(kernel);
    auto stencil_co = make_stencil_transform<9,9>(input_co);
    auto const_co = make_const_transform(kernel_co, x_size * y_size);
    auto zip_co = make_zip_transform(stencil_co, const_co);
    auto map_co = make_map_transform(zip_co,map_operation<
        mul_op>();
    auto reduce_co = 
        make_map_transform(map_co,reduce_operation<
            add_op>();
    }
```

Perform vector reduction on all map_co elements, this finishes convolution
```cpp
std::vector<int> 2dconvolution(const std::vector<int>& input,
    int x_size, int y_size,
    const std::vector<float>& kernel)
{
    auto input_co = make_matrix<int, 2>(input, x_size, y_size);
    auto kernel_co = make_small_vector<int>(kernel);
    auto stencil_co = make_stencil_transform<9,9>(input_co);
    auto const_co = make_const_transform(kernel_co, x_size * y_size);
    auto zip_co = make_zip_transform(stencil_co, const_co);
    auto map_co = make_map_transform(zip_co,map_operation<mul_op>());
    auto reduce_co =
        make_map_transform(map_co,reduce_operation<add_op>());
    std::vector<int> output = Evaluate<std::vector, int>(reduce_co);
}
```

The compiler performs actual code synthesis
Convolution Performance

- All versions (except Naive) are multithreaded and vectorized
  - 8x performance difference (Intel vs. Ours)

Run time measured on Intel Core i7-3820 (4-core, hyperthreaded)
Triolet C++ equalize_frames Example

- Baseline is parallel code taken from DARPA PERFECT benchmark suite
- Tuned code outperforms baseline on both architectures
10x10 Heterogeneous Architecture

- **1st set of micro-engines evaluation nearly complete**
- **Developing a set of complementary micro-engines**
- **Composite evaluation: 5x5**

**Compiler**

- **Compilation**: Mapreduce, micro-engine, and robust vectorization and locality management

**GPM**

- **Global Power Management**: power & performance models for core and link frequency changes + opportunity assessment
Compiled Conv2D on BnB

~10.5x improvement across the board
1024x1024 image, HMC memory system
Programming in Triolet Python

Nonuniform FT (real part)

\[ y_i = \sum_{j=0}^{n-1} x_j \cos(r_i k_j) \quad \text{for all } 0 \leq i < m \]
Programming in Triolet Python

Nonuniform FT (real part)

\[ y_i = \sum_{j=0}^{n-1} x_j \cos(r_i k_j) \quad \text{for all } 0 \leq i < m \]

Inner loop

\[
\text{sum}(x \times \cos(r \times k) \text{ for } (x, k) \text{ in zip(xs, ks))}
\]
Programming in Triolet Python

Nonuniform FT (real part)

\[ y_i = \sum_{j=0}^{n-1} x_j \cos(r_i k_j) \quad \text{for all } 0 \leq i < m \]

Inner loop

\[
ys = [\text{sum}(x * \cos(r*k) \text{ for } (x, k) \text{ in zip(xs, ks)))}
\]

Outer loop

for r in par(rs)
Programming in Triolet Python

Nonuniform FT (real part)

$$y_i = \sum_{j=0}^{n-1} x_j \cos(r_i k_j) \quad \text{for all } 0 \leq i < m$$

- “map and reduce” style programming—no new paradigm to learn
- Parallel details are implicit—easy to use
- Automated data partitioning, MPI rank generation, MPI messaging, OpenMP, etc.
- Race-free, type-safe—no crashes or nondeterminism
  (with standard caveat about operator associativity)
Productivity

Triolet

\[ \text{ys} = \left[ \text{sum}(x \times \cos(r \times k) \text{ for } (x, k) \text{ in zip(xs, ks)}) \right] \text{ for } r \text{ in par(rs)} \]

- Library functions factor out data decomposition, parallelism, and communication
Productivity and Performance

**Triplet**

\[ y_s = \text{sum}(x \times \cos(r\times k) \text{ for } (x, k) \text{ in zip(xs, ks)}) \text{ for } r \text{ in par(rs)} \]

- Library functions factor out data decomposition, parallelism, and communication

<table>
<thead>
<tr>
<th>128-way Speedup (16 cores x 8 nodes)</th>
<th>Triplet</th>
<th>C with MPI+OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>99</td>
<td>115</td>
</tr>
</tbody>
</table>

C with MPI+OpenMP

```
$ time python triplet.py
real    0m0.253s   user    0m0.240s   sys     0m0.013s

$ time python mtriplet.py
real    0m0.000s   user    0m0.000s   sys     0m0.000s

$ time python mtriplet.py -t 10
real    0m0.000s   user    0m0.000s   sys     0m0.000s
```

Setup

```
import numpy as np
import mpmath as mp

mp.dps = 50
mp.pretty = True

# Define the function
def triplet(x, k, r):
    return np.sum(x * np.cos(r * k))

# Generate some random data
np.random.seed(0)
x = np.random.rand(1000)
k = np.random.rand(1000)
r = np.random.rand(1000)

# Call the function
result = triplet(x, k, r)
```

Cleanup
Cluster-Parallel Performance and Scalability

• **Triolet** delivers large speedup over sequential C
  – On par with manually parallelized C
  – Except in CUTCP; needs better GC policy for large arrays

• Similar high-level interfaces incur additional overhead
  – Message passing
  – Array split/merge
  – Run time variability

---

**MRI-Q**

**SGEMM**

**TPACF**

**CUTCP**

---

Rodrigues, et al. PPoPP 2014
ACS Productivity Workshop, 2014
Triolet Pattern Domain Coverage Example

- Benchmarks suitable for HOF library
  - Each loop has one output
  - Outer parallelizable map in many benchmarks
  - Small set of computation patterns used repeatedly
    - Loops: map, reduce
    - Data merging: zip, outer product
    - array reshaping
- 8 of 15 benchmarks already ported to Triolet Python

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Extra Patterns</th>
<th>Vectorizable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discrete Wavelet Transform</td>
<td>deinterleave, regions</td>
<td>pixels</td>
</tr>
<tr>
<td>2 D Convolutions</td>
<td>regions</td>
<td>pixels</td>
</tr>
<tr>
<td>Histogram Equalization</td>
<td>histogram, scan, lut</td>
<td>pixels, bins</td>
</tr>
<tr>
<td>System Solver</td>
<td>triangular loop</td>
<td>matrix rows</td>
</tr>
<tr>
<td>Inner Product</td>
<td>-</td>
<td>channels</td>
</tr>
<tr>
<td>Outer Product</td>
<td>triangular loop</td>
<td>channels</td>
</tr>
<tr>
<td>Interpolation 1</td>
<td>-</td>
<td>range coords</td>
</tr>
<tr>
<td>Interpolation 2</td>
<td>lut</td>
<td>range coords</td>
</tr>
<tr>
<td>Back Projection</td>
<td>lut</td>
<td></td>
</tr>
<tr>
<td>Debayer</td>
<td>interleave, regions</td>
<td>pixels</td>
</tr>
<tr>
<td>Change Detection</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Sort</td>
<td>sort</td>
<td>array elements</td>
</tr>
<tr>
<td>FFT 1D</td>
<td>fft</td>
<td></td>
</tr>
<tr>
<td>FFT 2D</td>
<td>fft</td>
<td></td>
</tr>
</tbody>
</table>
Conclusion

- Near-term impact
  - MxPA locality-centric scheduling, kernel fusion scheduling, and dynamic vectorization make OpenCL kernel performance portability a reality – ready for industry impact
  - MulticoreWare MxPA product with several customers including Movidius and Samsung

- Medium term impact
  - Triolet C++ brings intentional programming into C++, giving CUDA/OpenCL/OpenMP/OpenACC developers a much more productive, maintainable, portable new option
  - Immediate commercial opportunity in mobile and server SOCs

- Long-term outlook
  - Triolet Python further brings intentional programming into heterogeneous computing server clusters and distributed computing
  - Triolet Java?