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

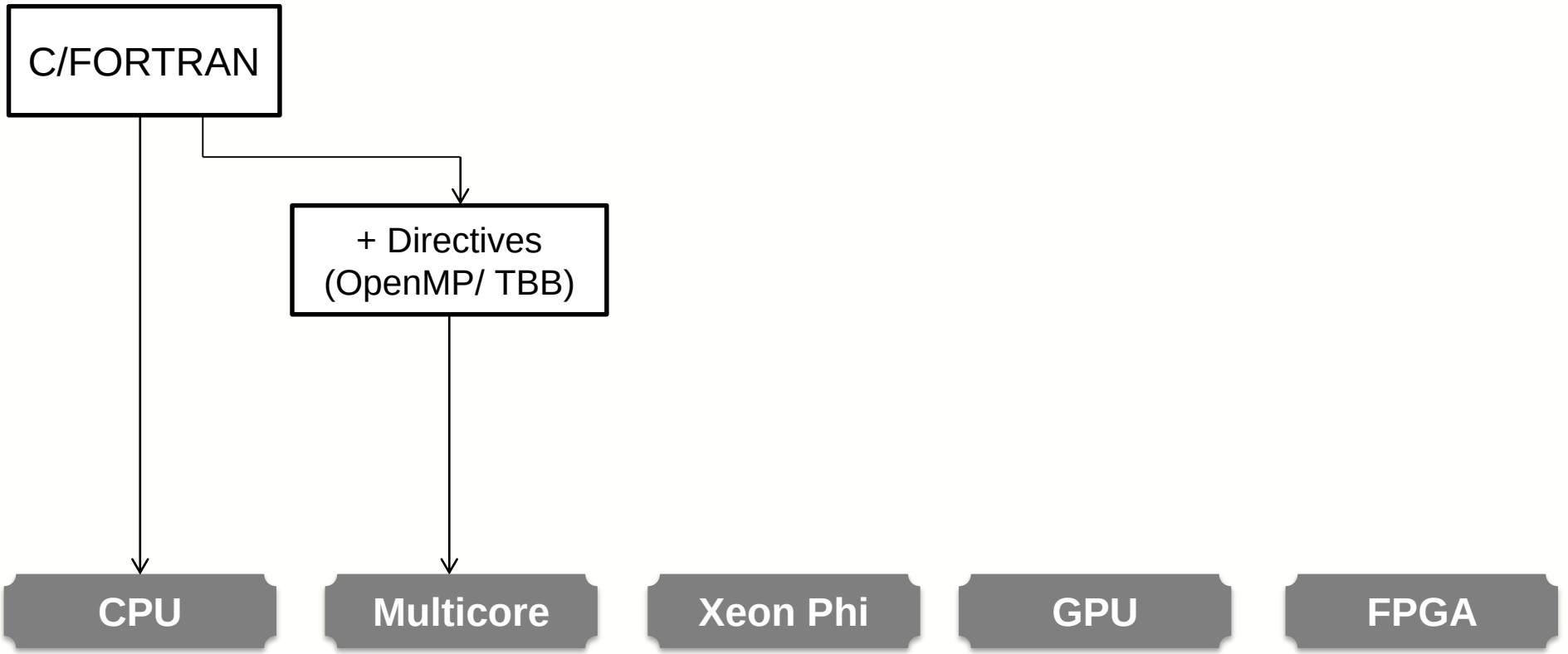
- Reduction trees
- Array packing
- Boundary cell communication

Current State of Programming Heterogeneous Systems

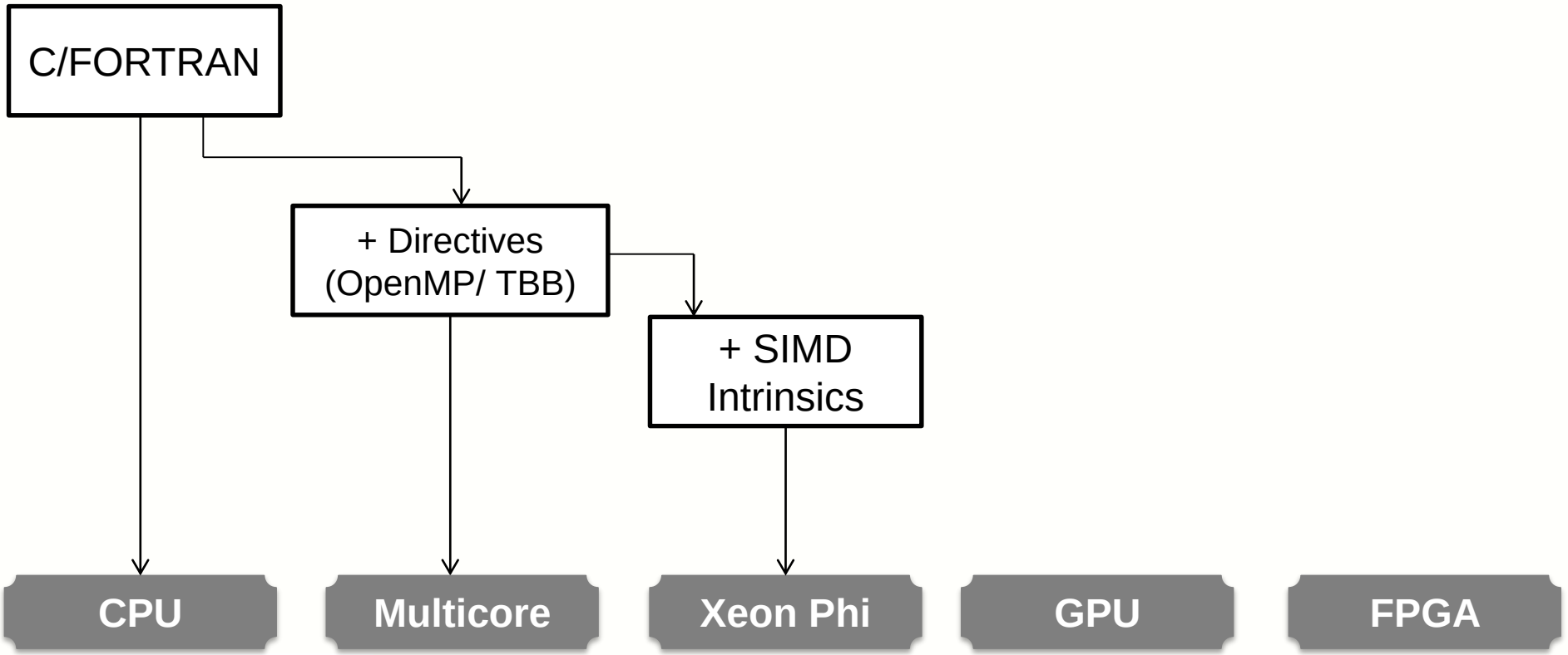
CPU | Multicore | Xeon Phi | GPU | FPGA

Current State of Programming Heterogeneous Systems

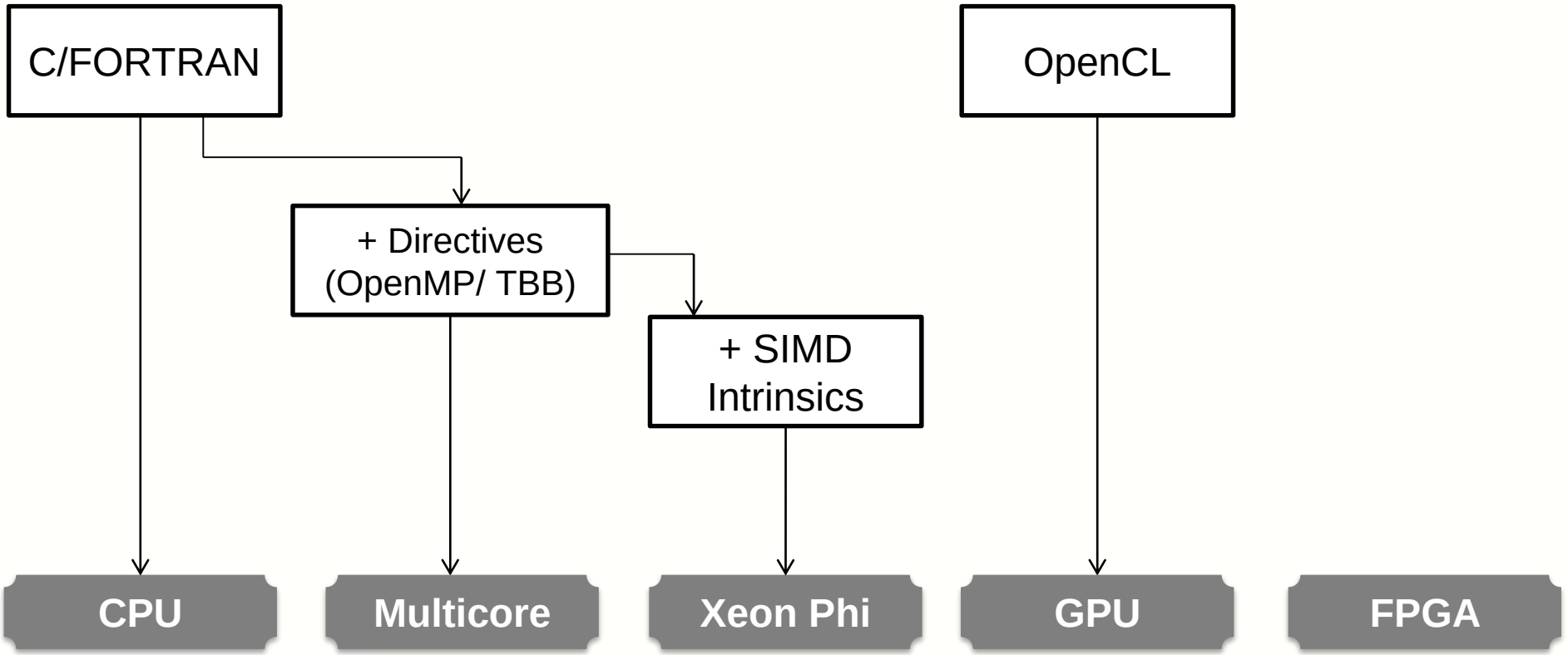
Current State of Programming Heterogeneous Systems



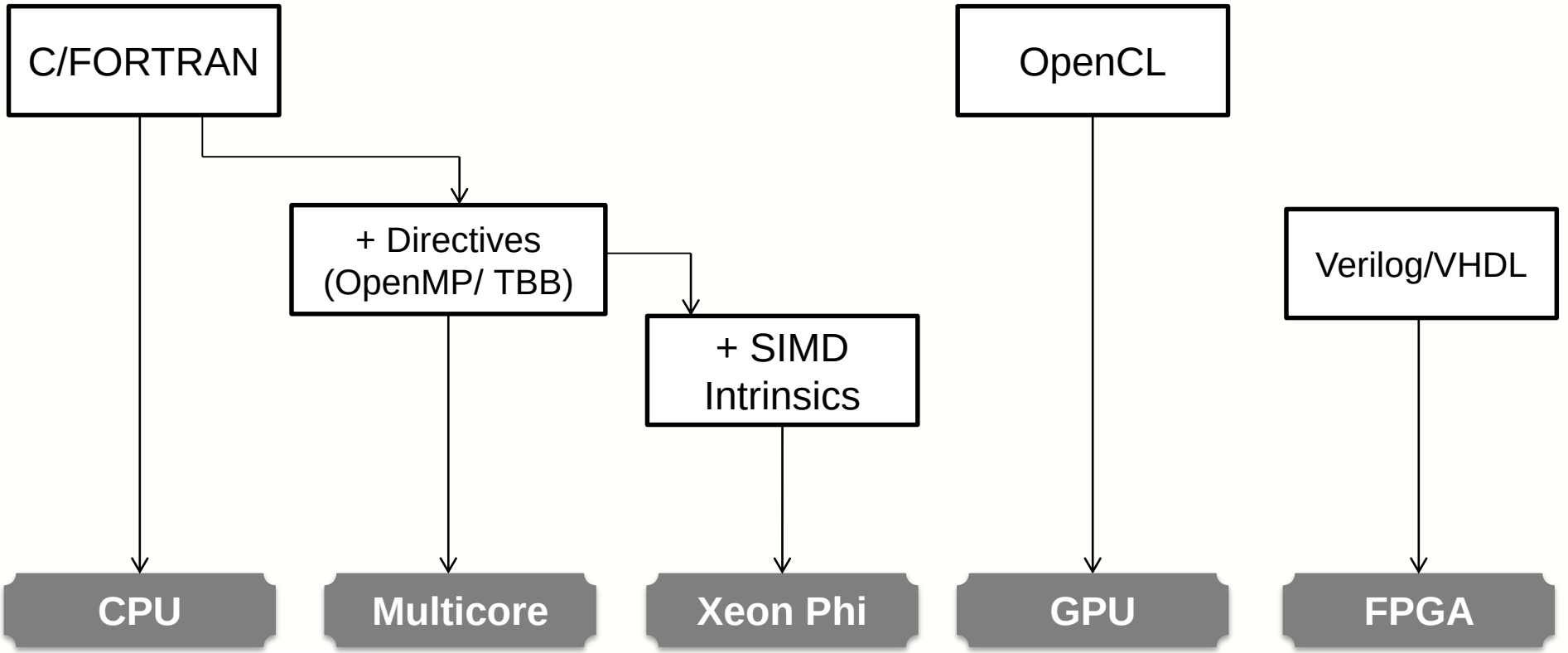
Current State of Programming Heterogeneous Systems



Current State of Programming Heterogeneous Systems

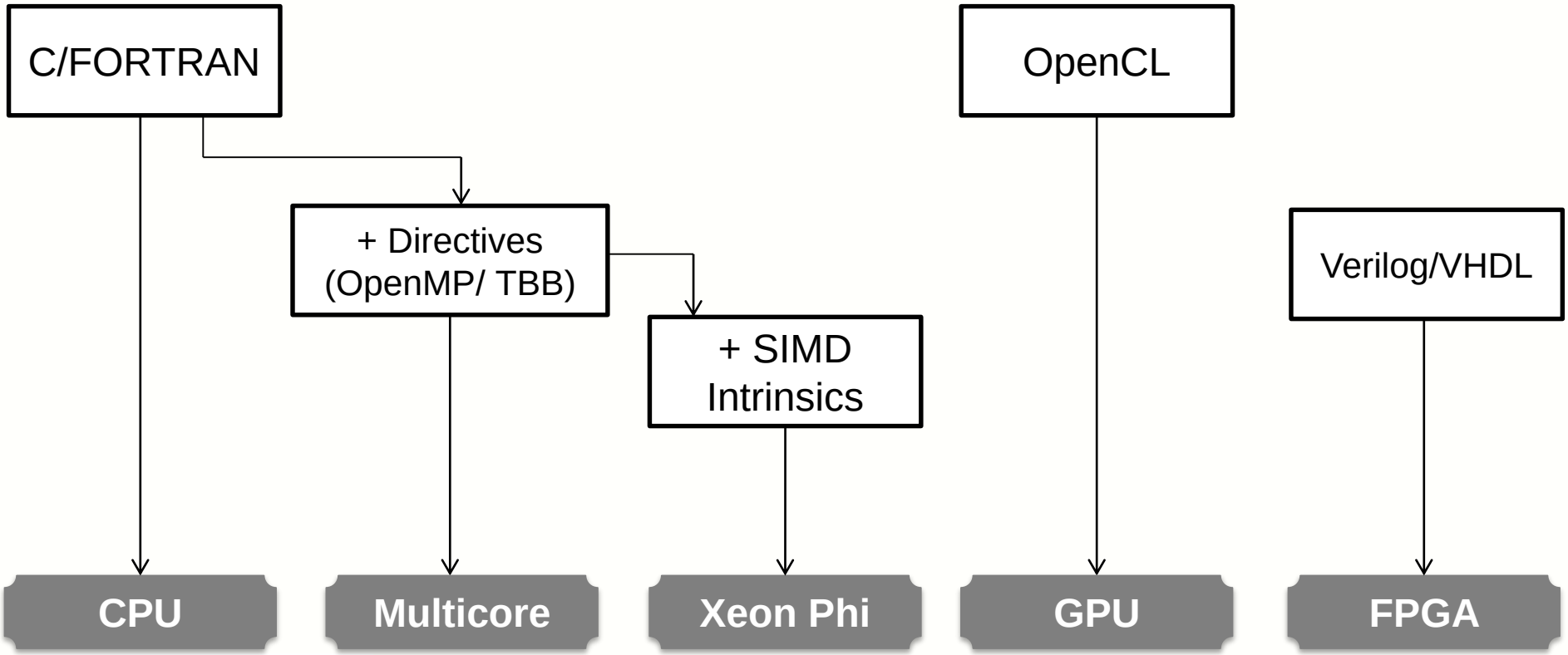


Current State of Programming Heterogeneous Systems



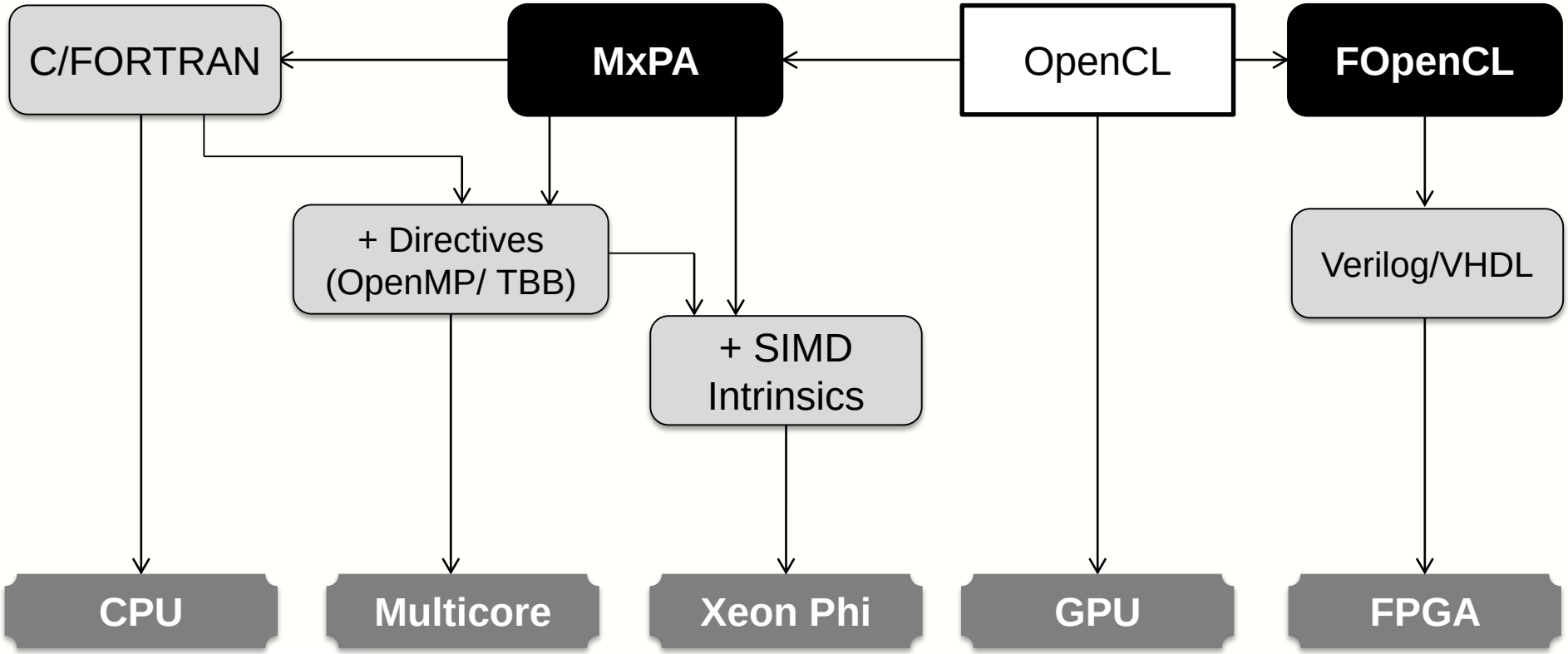
Current State of Programming Heterogeneous Systems

Programming heterogeneous systems requires too many versions of code!



Productivity in Programming Heterogeneous Systems

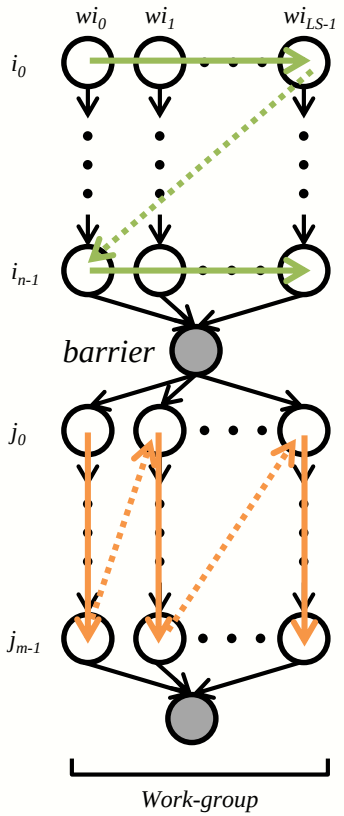
<u>Step #1:</u> Keep only one of the versions and use portability tools to generate the others.



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

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

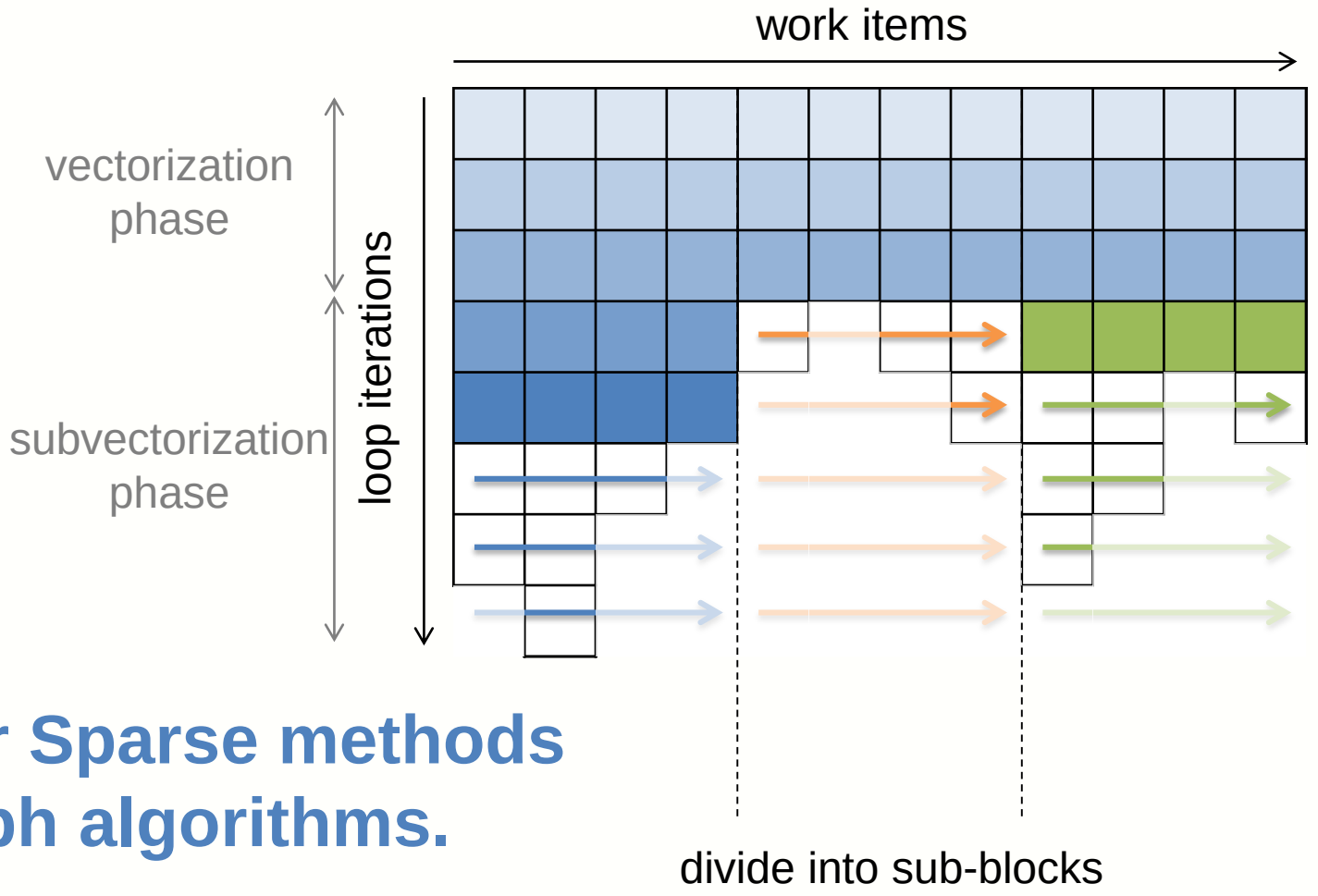
An example OpenCL code

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

  // DFO
  for (wid = 0..LS) {
    for (j = 0..M) {
      .. = B[c1*wid + j];
    }
  }
}
```

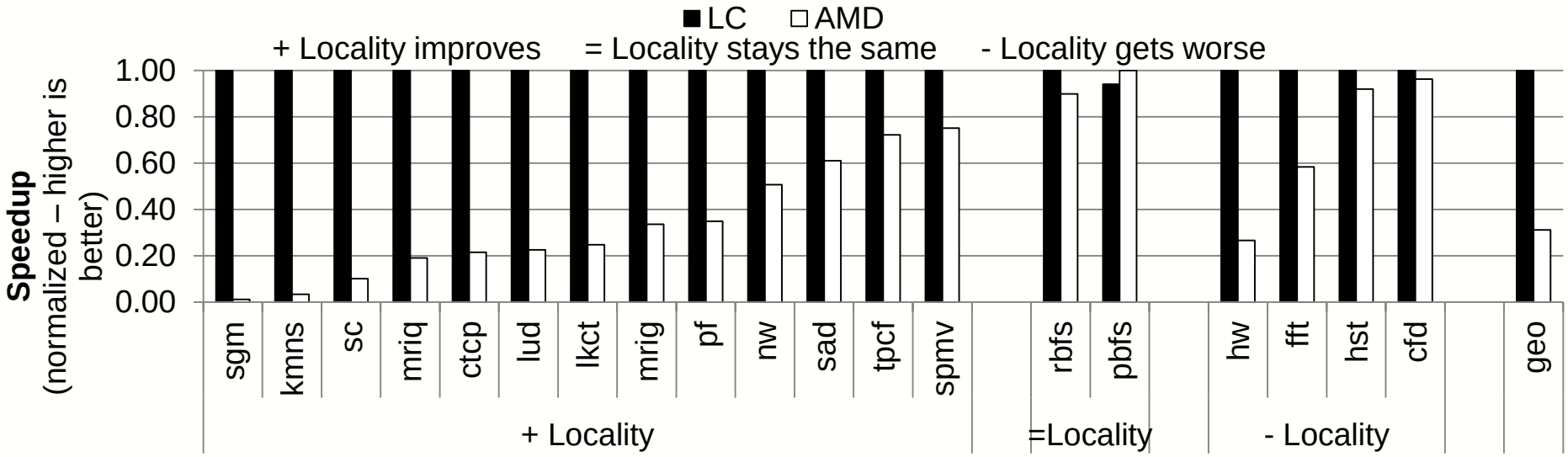
MXPA-translated code

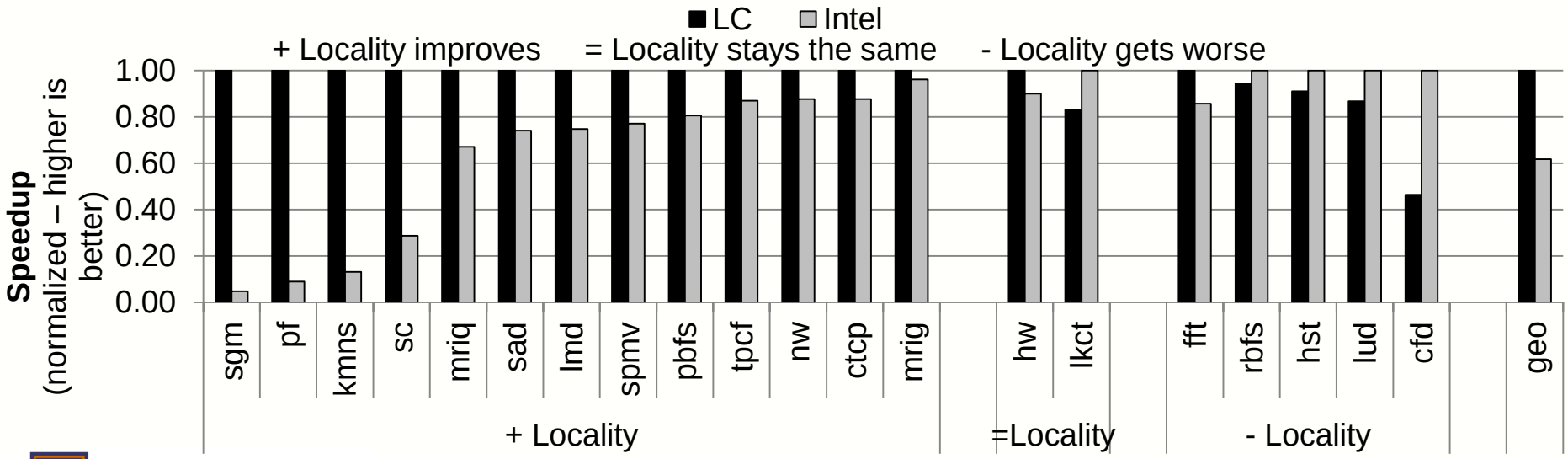
MxPA: Dynamic Vectorization of Control Divergent Loops



Effective for Sparse methods and graph algorithms.

MxPA Results: Comparison to Industry



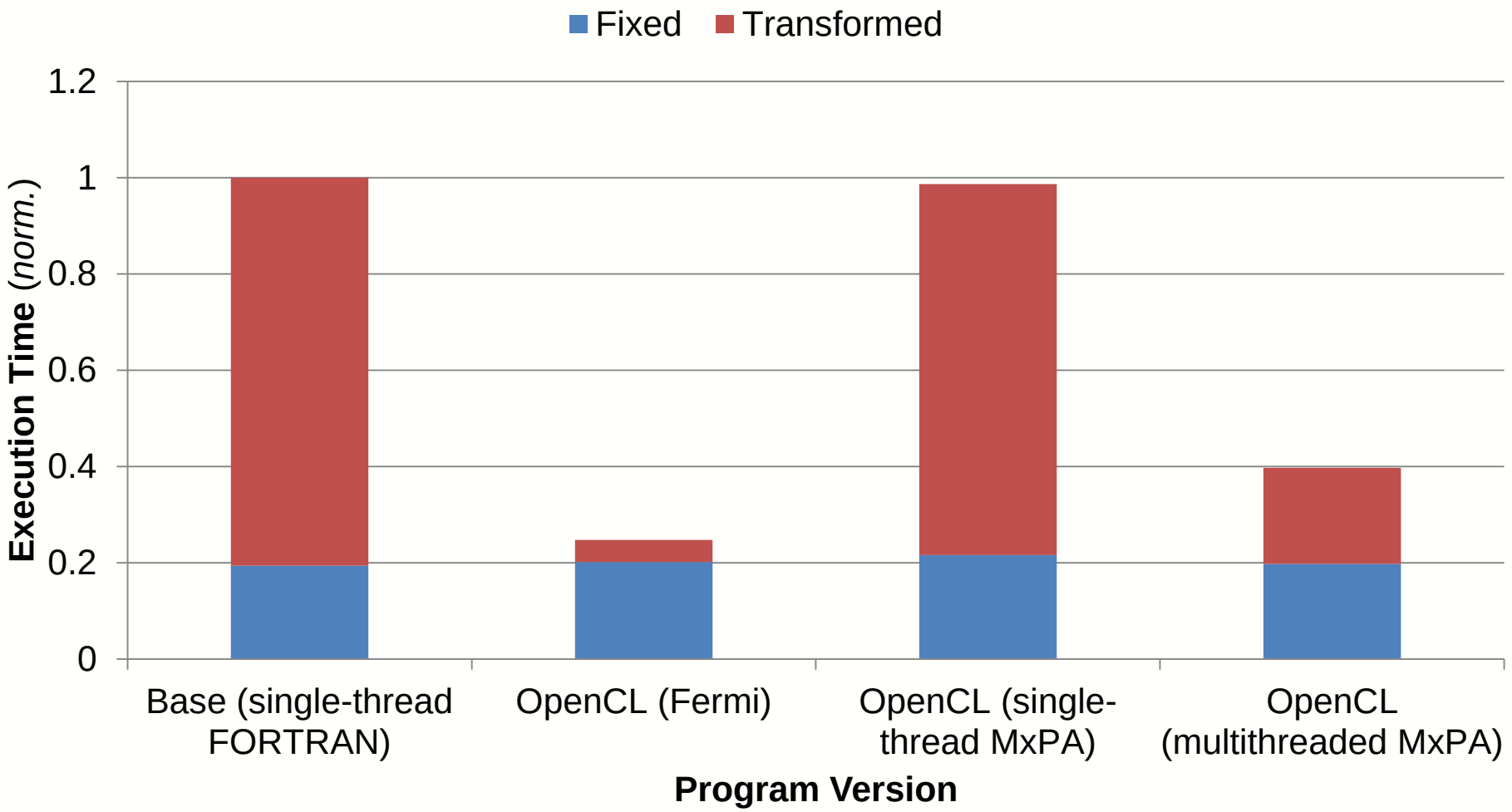


ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

MxPA Results: Comparison to Industry

Metric	LC/AMD	LC/Intel
Speedup	3.20x	1.62x
L1 Data Cache Misses	0.12x	0.36x
Data TLB Misses	0.23x	0.33x
LLC Misses	0.91x	0.97x

MxPA Case Study: MOCFE-Bone



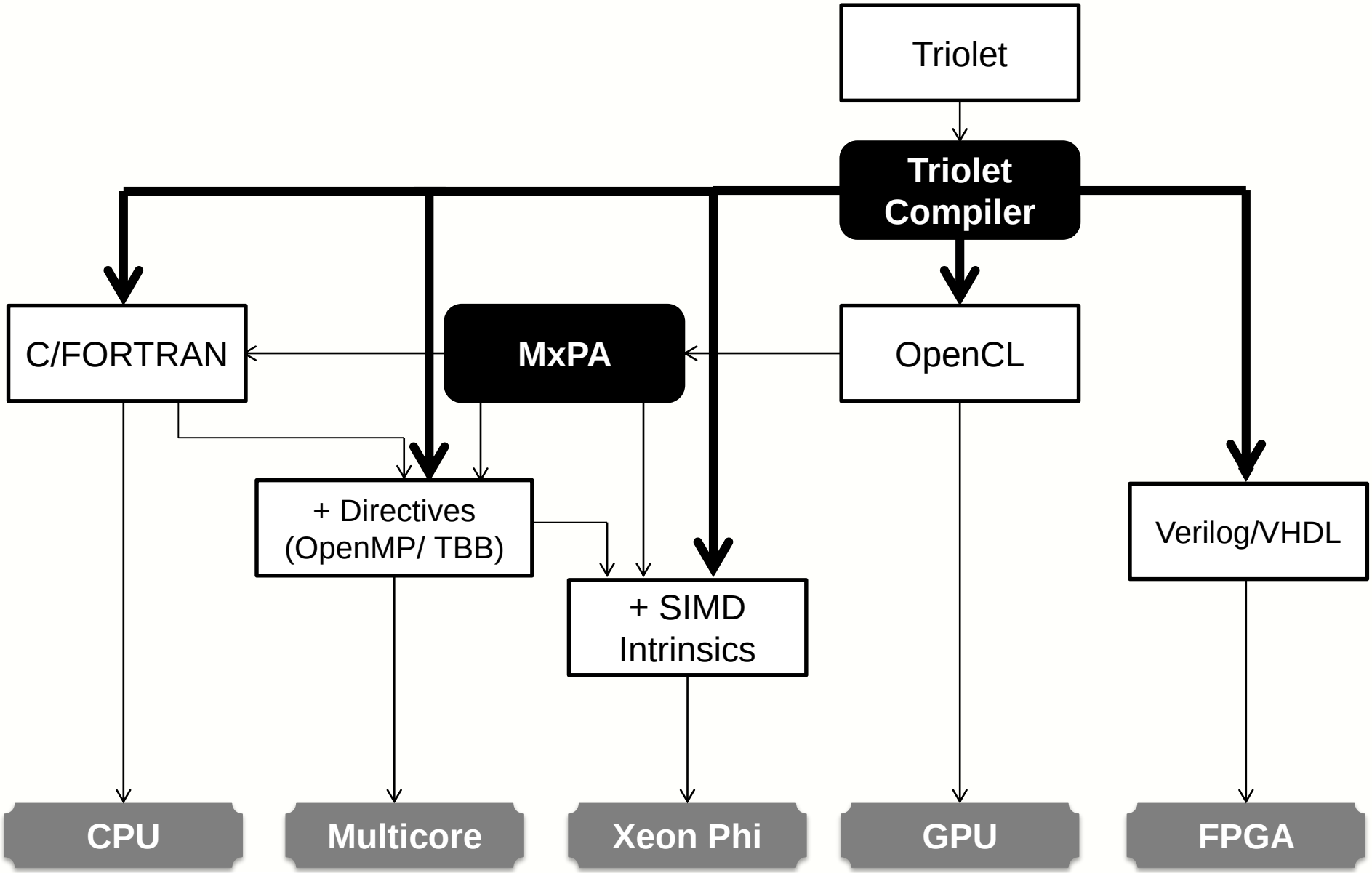
Input Configurations: Nodes = 1, Groups = 25, Angles = 128, MeshScale=10 (Elements=10^3)

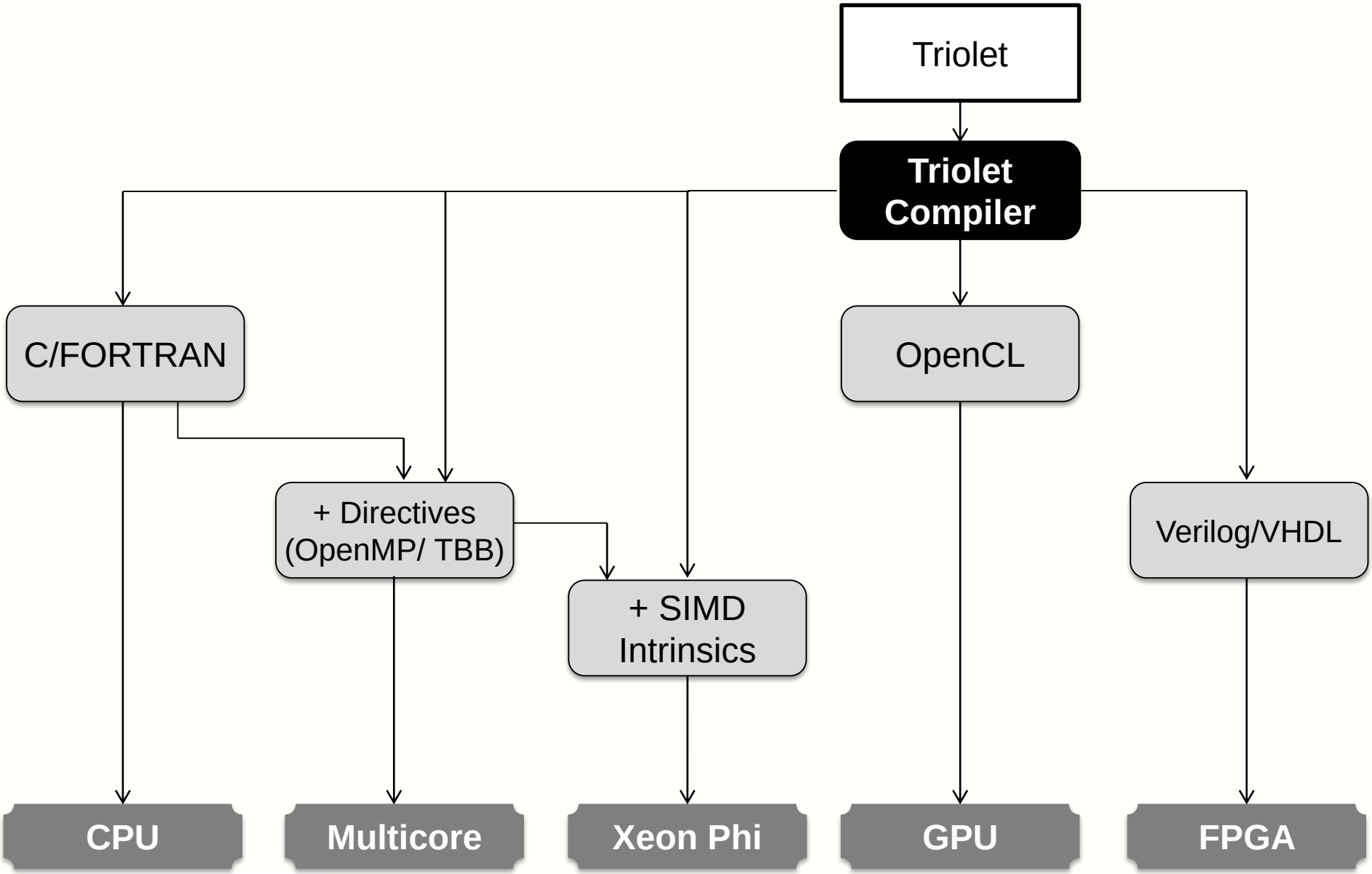
HIGH-LEVEL INTERFACE

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
Triolet Compiler
C/FORTRAN
OpenCL
+ Directives (OpenMP/ TBB)
+ SIMD Intrinsics
Verilog/VHDL
CPU
Multicore
Xeon Phi
GPU
FPGA

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

```
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

```
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

```
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_size by y_size matrix whose elements are the 9x9 neighbor stencils around the original **input_co** elements.

Triolet/C++ Convolution Code

```
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);
}
```

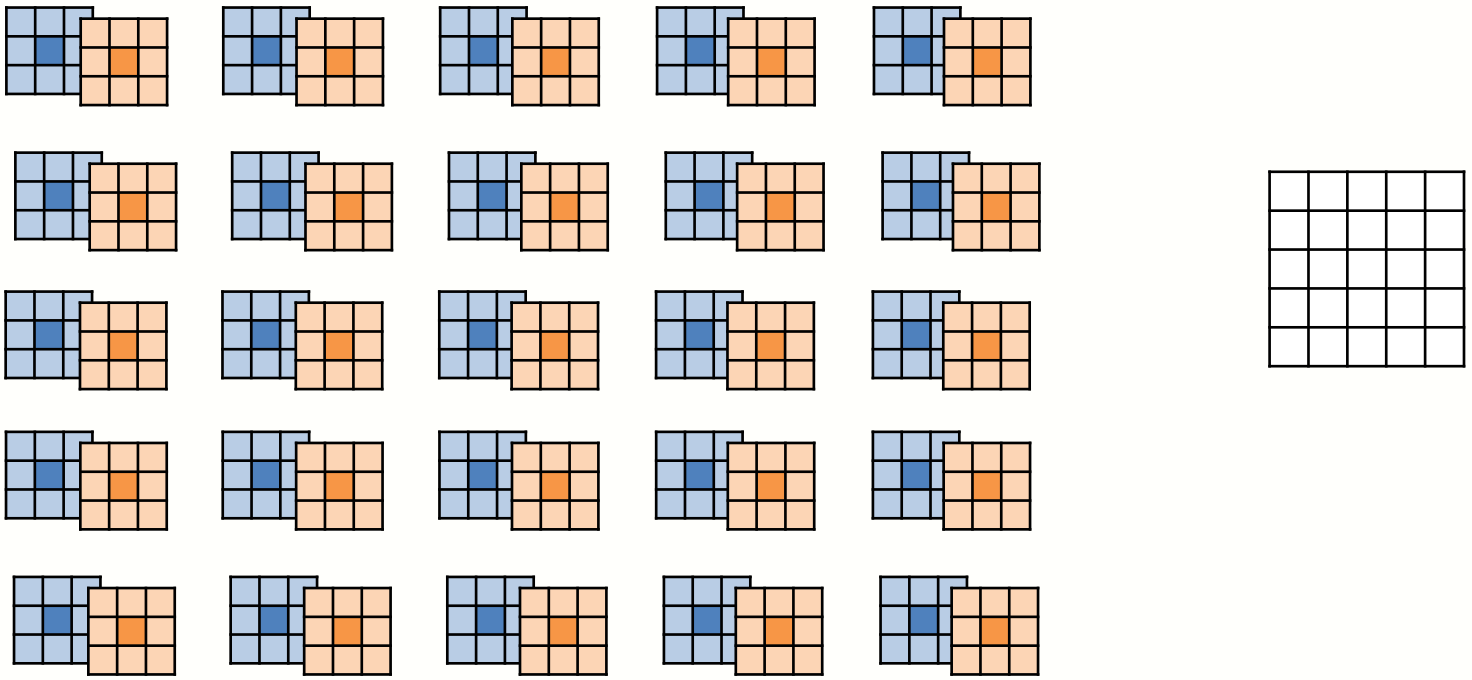
Conceptually replicate **kernel_co** into an x_size by y_size stencil matrix

Triolet/C++ Convolution Code

```
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);
}
```

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.

Convolution Example (5x5 Zipped Matrix)



zip(filters, stencils)

Triolet/C++ Convolution Code

```
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

```
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

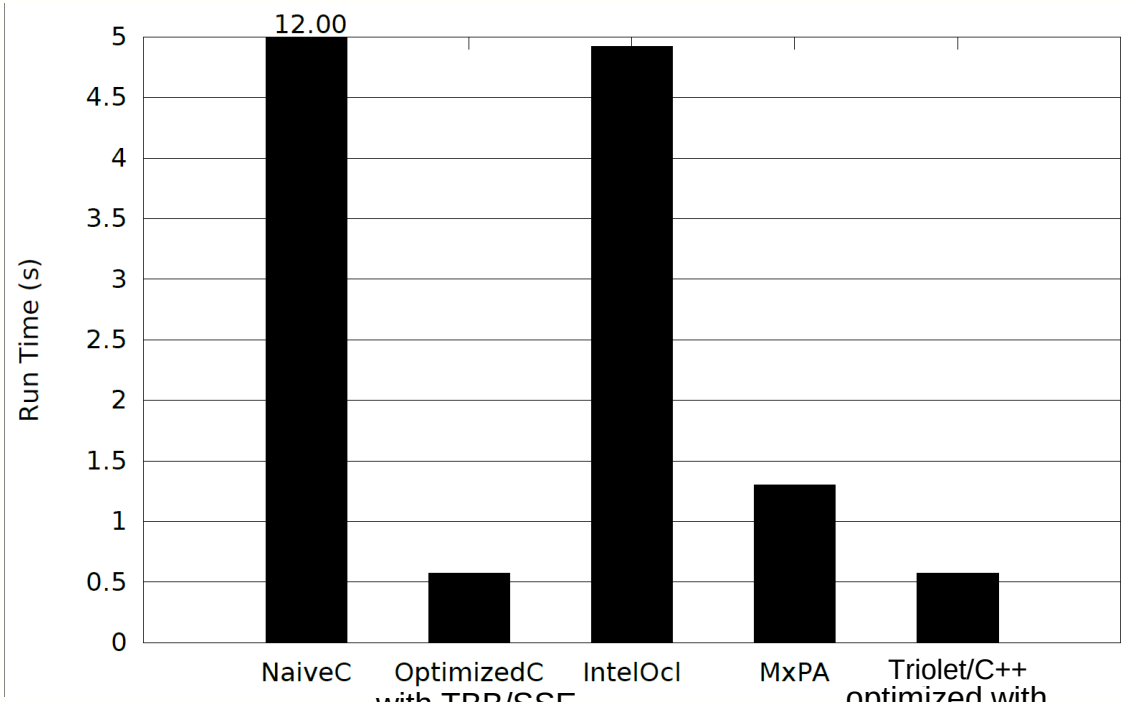
Triolet/C++ Convolution Code

```
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

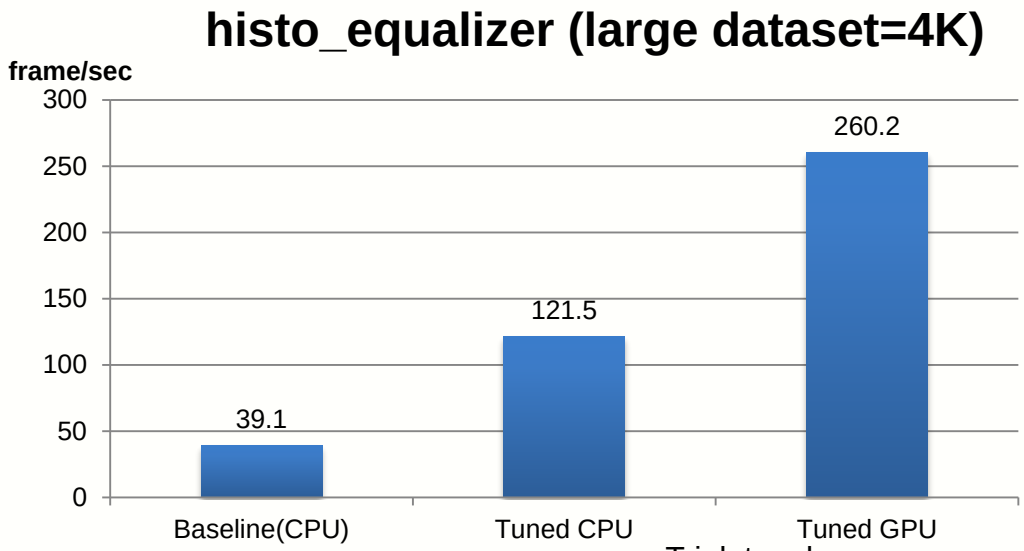
- All versions (except Naive) are multithreaded and vectorized
  - 8x performance difference (Intel vs. Ours)



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

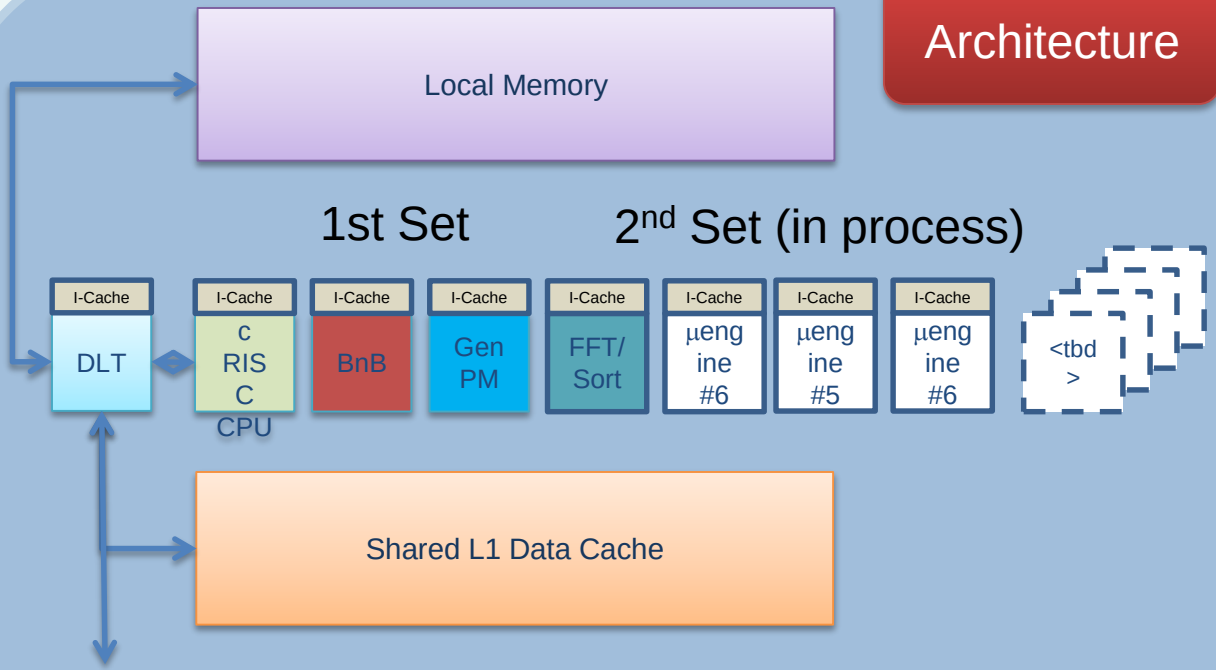


CPU time measured on Intel Xeon E5520 (4-core, hyperthreaded)

GPU time measured on Tesla C2050

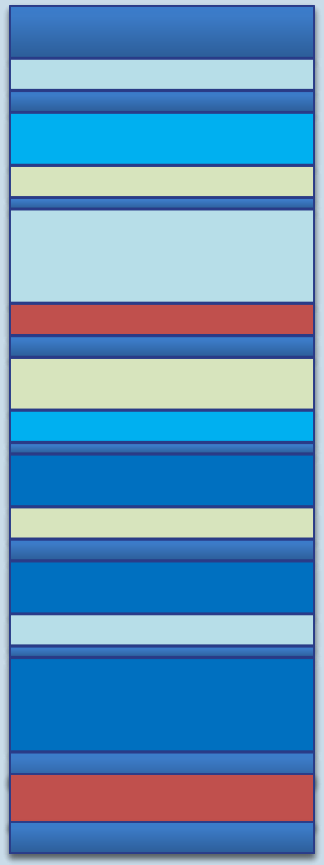
10x10 Heterogeneous Architecture

Architecture



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

Application



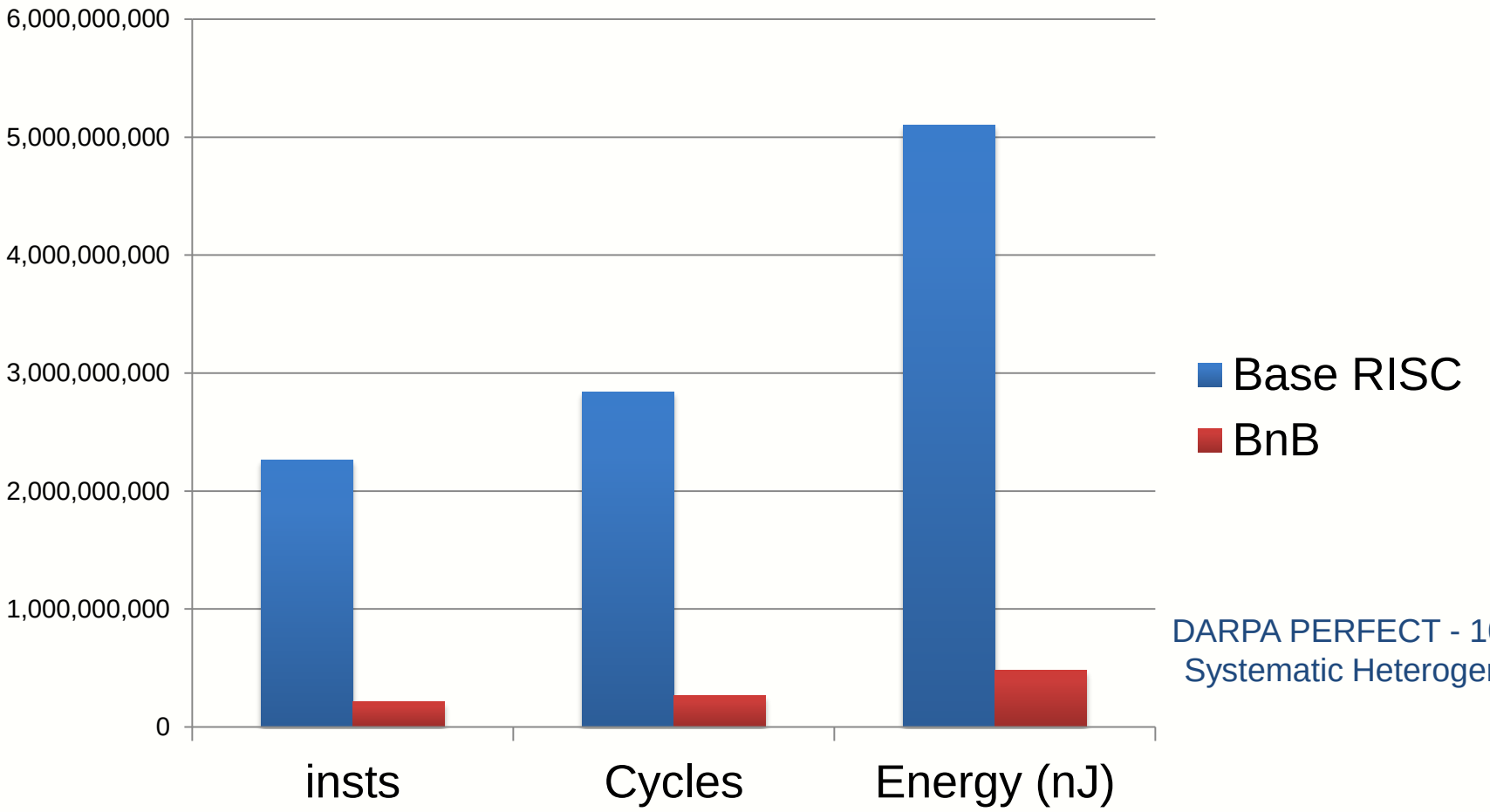
Compiler

- 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



DARPA PERFECT - 10x10: Systematic Heterogeneity

~10.5x improvement across the board
1024x1024 image, HMC memory system

Programming in Triolet Python

Nonuniform FT (real part)

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

Programming in Triolet Python

Nonuniform FT (real part)

Inner loop

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

Inner loop

```
sum(x * cos(r*k) for (x, k) in zip(xs, ks))
```

Programming in Triolet Python

Nonuniform FT (real part)

Inner loop

Outer loop

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

```
ys = [ sum(x * cos(r*k) for (x, k) in zip(xs, ks)) for r in par(rs)]
```

Inner loop

Outer loop

Programming in Triolet Python

Nonuniform FT (real part)

Inner loop

Outer loop

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

```
ys = [ sum(x * cos(r*k) for (x, k) in zip(xs, ks)) for r in par(rs)]
```

Inner loop

Outer loop

- "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)

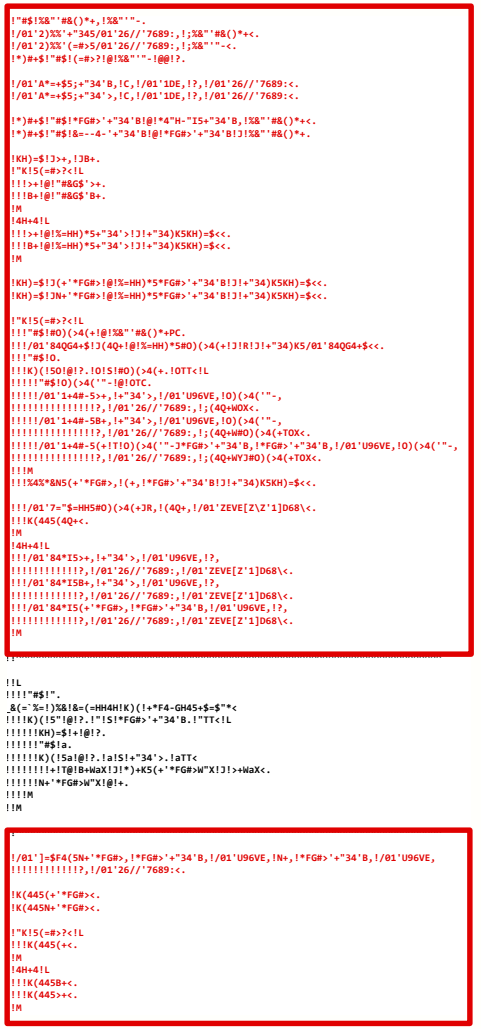
Productivity

Triolet

```
ys = [sum(x * cos(r*k) for (x, k) in zip(xs, ks))
      for r in par(rs)]
```

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

C with MPI+OpenMP



Productivity and Performance

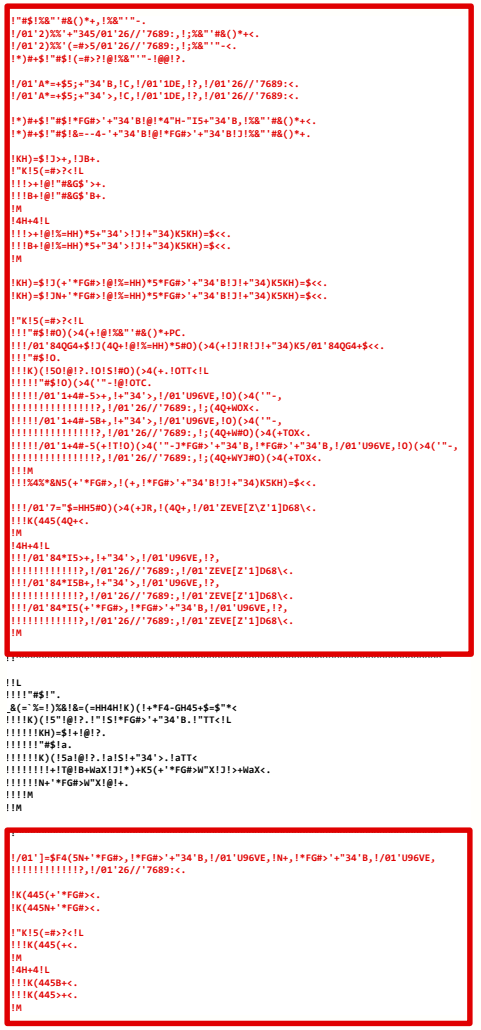
Triolet

```
ys = [sum(x * cos(r*k) for (x, k) in zip(xs, ks))
      for r in par(rs)]
```

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

128-way Speedup (*16 cores × 8 nodes*)	**Triolet**	**C with MPI+OpenMP**
	99	115

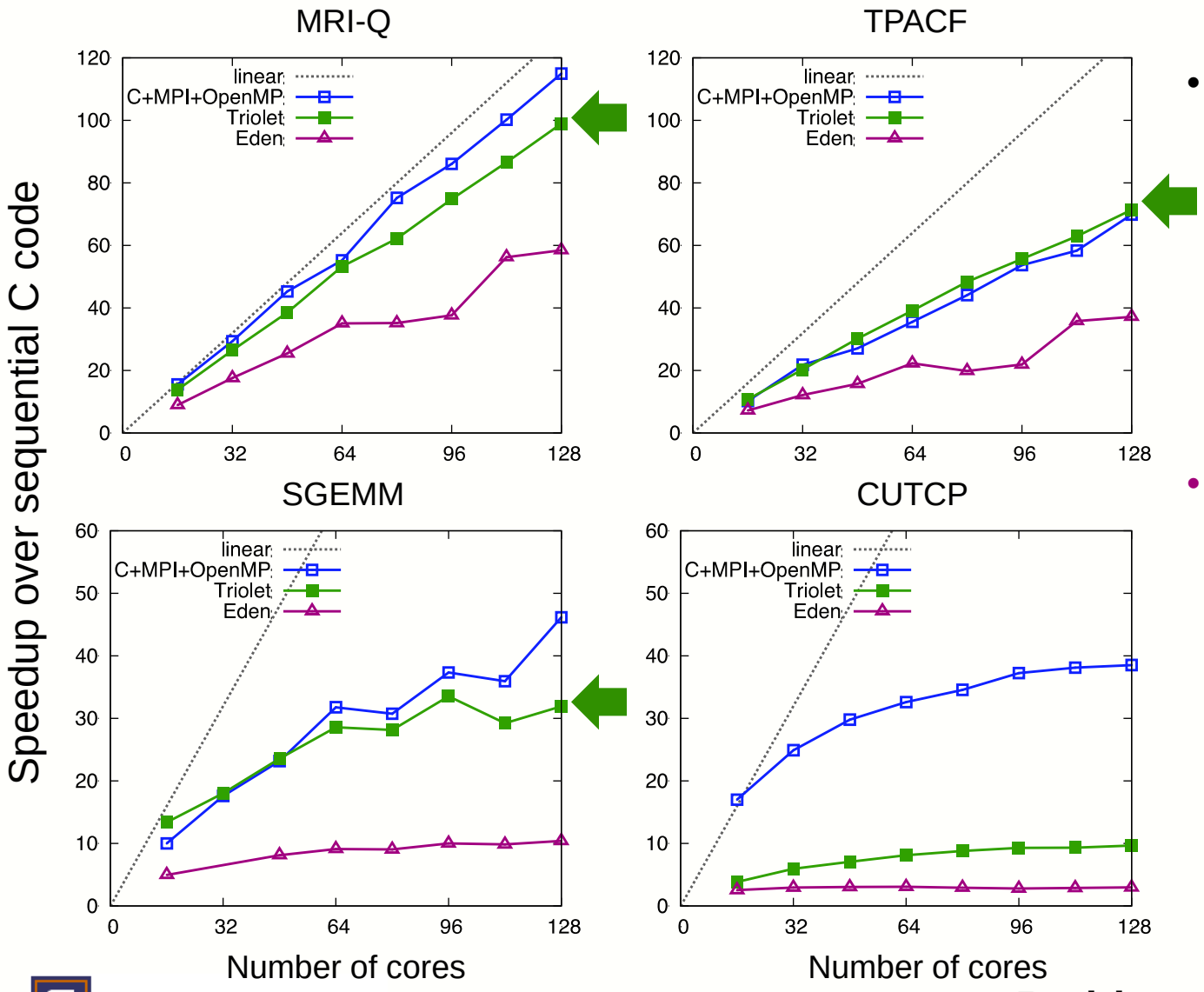
C with MPI+OpenMP



Setup

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

Rodrigues, et al PPoPP 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

Benchmark	Extra Patterns	Vectorizable
Discrete Wavelet Transform	deinterleave, regions	pixels
2 D Convolutions	regions	pixels
Histogram Equalization	histogram, scan, lut	pixels, bins
System Solver	triangular loop	matrix rows
Inner Product	-	channels
Outer Product	triangular loop	channels
Interpolation 1	-	range coords
Interpolation 2	lut	range coords
Back Projection	lut	
Debayer	interleave, regions	pixels
Image Registration	-	pixels
Change Detection	-	-
Sort	sort	array elements
FFT 1D	fft	
FFT 2D	fft	

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, protable 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?