EMBRACING HETEROGENEITY - PARALLEL PROGRAMMING FOR CHANGING HARDWARE WITH THE MERGE FRAMEWORK

Michael D. Linderman, Teresa H. Meng
Dept. of Electrical Engineering, Stanford University
mlinderm@stanford.edu

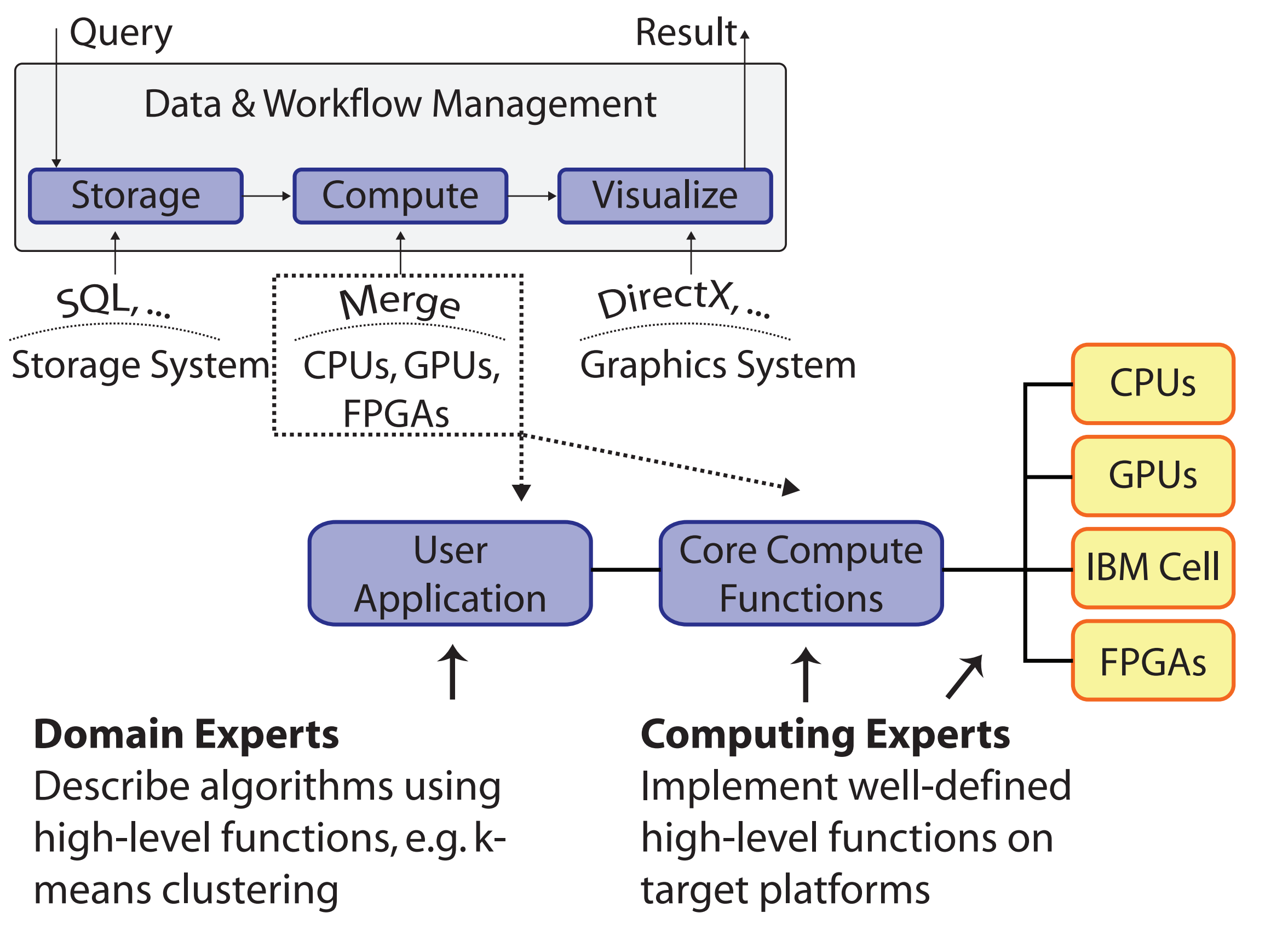
Introduction

The computational requirements of informatics applications are growing at an exponential rate.

Goal: Enable 'productivity' programmers to exploit state-of-the-art heterogeneous multicore computers

Heterogeneous systems, which integrate specialized accelerators such as GPUs or FPGAs alongside multicore general-purpose processors (GPPs), can deliver the scalable performance needed by demanding informatics applications.

Programming such systems is a keen challenge, however. New tools are needed. Unfortunately there is no 'one' compiler that can meet all our needs. The solution lies in integrating many narrowly focused processor, application domain, and vendor-specific toolchains. The Merge framework does exactly this.

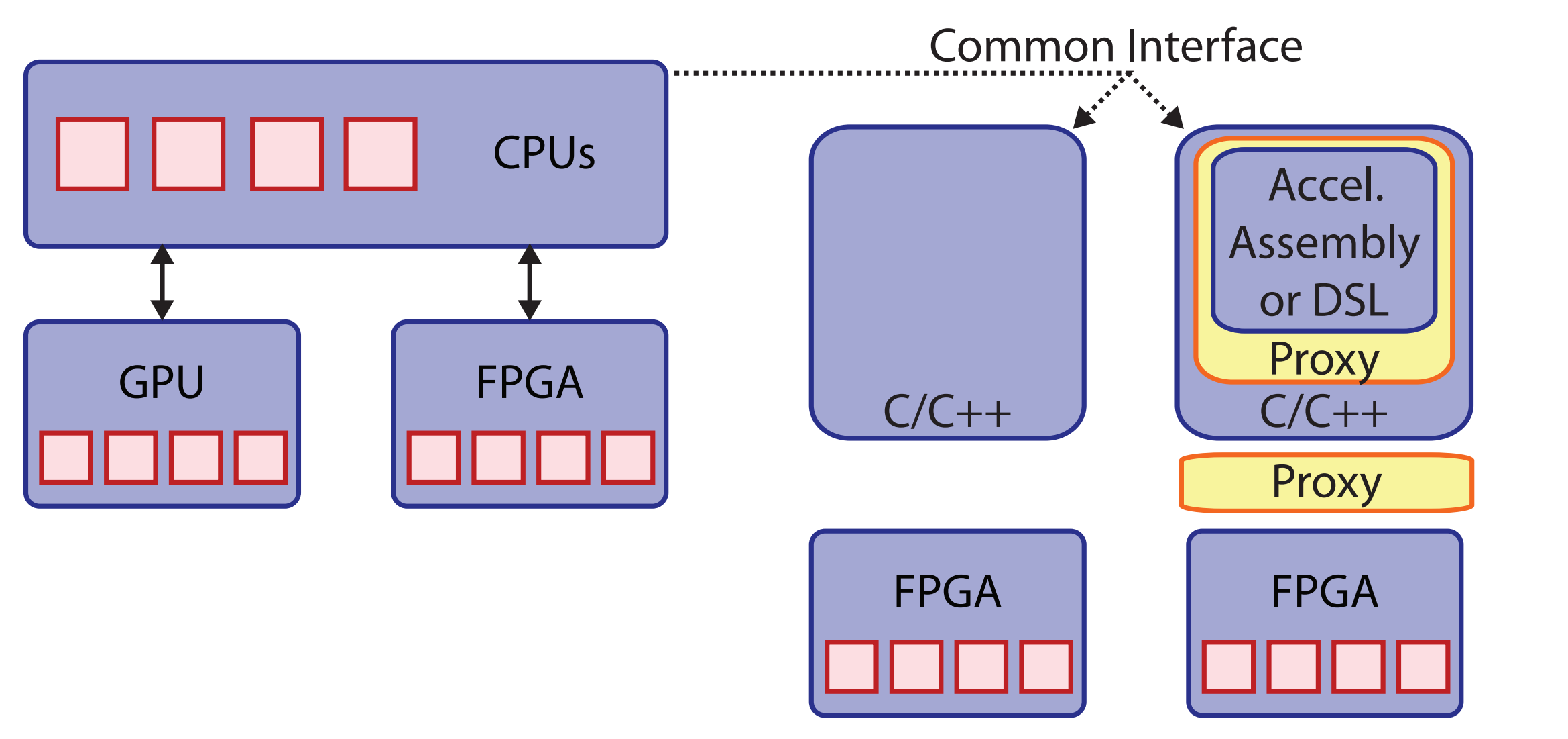


Merge abstracts away the complexity of heterogeneous systems behind a backward-compatible function metaprogramming interface. Applications, written by domain experts, target these processor-agnostic interfaces, which are automatically mapped to compute-expert-created processor-specific modules.

Composition & Coordination

The heart of any integration technique is a common abstraction. Traditional compiler-driven approaches, built around a fixed set of primitives, e.g. short vectors, cannot abstract all the computational features we are interested in. A different, more inclusive, abstraction is needed.

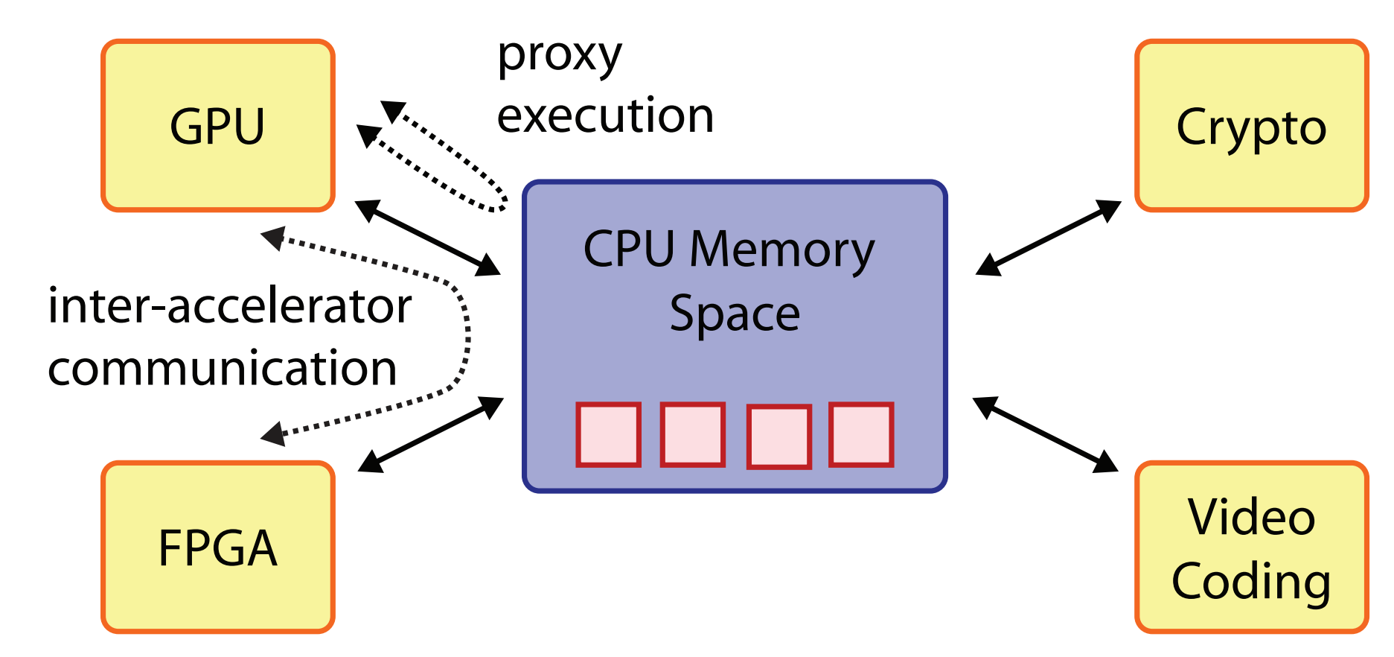
Instead of a particular arithmetic or memory operation, Merge is build around functions, a type of inclusive primitive available in most programming languages, and a common - "hub & spoke" - system organization.



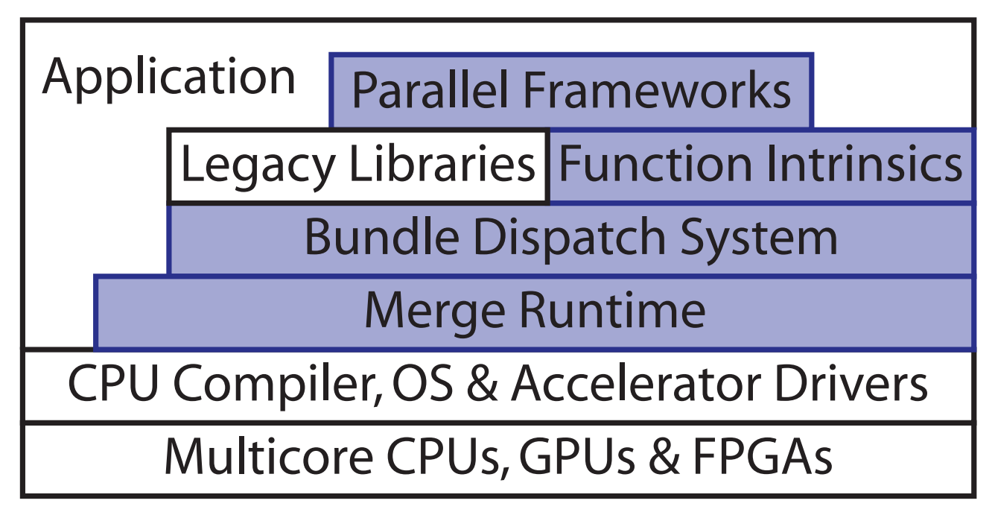
Programmers can implement function APIs using traditional C/C++ code targeting the CPUs, or using processor-specific code targeting a specialized accelerator. Regardless of the approach, both implementations expose the same, legacy-compatible, function interface.

The use of functions (and libraries) facilitates efficient reuse, while the common interface supports composability across a readily extensible set of processors.

The proxy layer provides the interface between the CPU and accelerator memory spaces. All computations begin on the CPU, the "hub", and are then dispatched to the "spokes". Using the CPU as the communication ensures compatibility with legacy code that does not know about the accelerators.



Merge Framework



Merge software stack; Merge components hightlighted in blue. Each layer builds on the previous, and all present interfaces compatible with legacy SW infrastructure.

Merge has three key components:

1) Function Intrinsics: Encapsulate specialized code in legacy compatible C/C++ function wrappers

2) Bundling: Function intrinsics for the same computation are collected into "bundles" that provide indirection and introspection. Dispatch system automatically maps applications to particular variants based on programmer-supplied annotations.

3**) Parallel Frameworks**: High-level parallel programming languages built on the encapsulation and bundling capabilities.

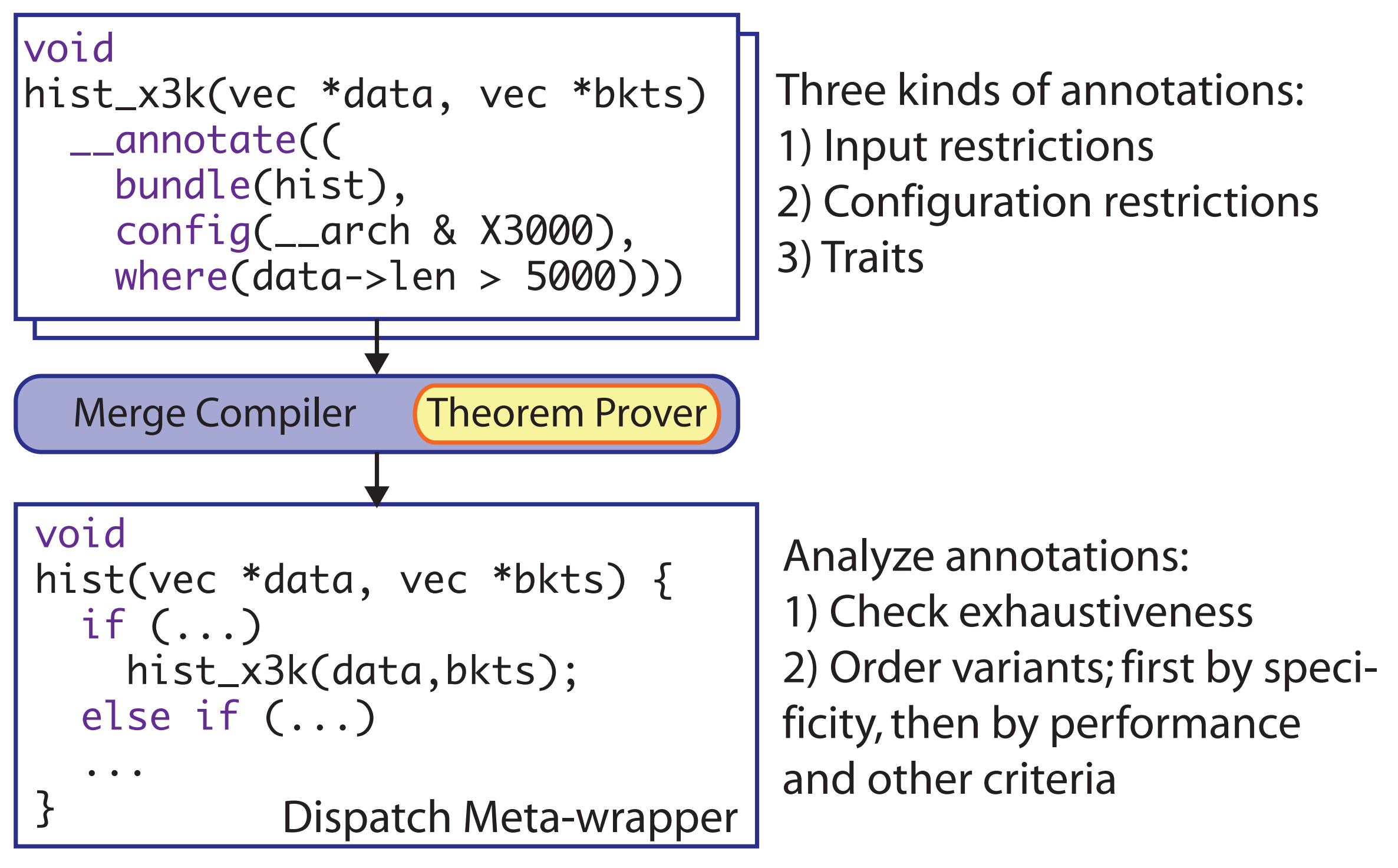
```
void hist_x3k(vec *data, vec *bkts)
  __annotate((
    bundle(hist),
    config(__arch & X3000),
    where(data->len > 5000)))
{
  data_desc = alloc_buffer(...); ...
  #pragma X3000 bind(data_desc,...) {
    // X3000 GPU Implementation
  }
  free(data_desc);
}
```

C function interface

Annotations used by bundle dispatch system

Encapsulated assembly for X3000 GPU

Function intrinsic for histogram targeting X3000 integrated GPU



```
void
hist_x3k(vec *data, vec *bkts)
  __annotate((
    bundle(hist),
    config(__arch & X3000),
    where(data->len > 5000)))
```

Three kinds of annotations:
1) Input restrictions
2) Configuration restrictions
3) Traits

```
void
hist(vec *data, vec *bkts) {
  if (...)
    hist_x3k(data,bkts);
  else if (...)
  ...
}
```

Analyze annotations:
1) Check exhaustiveness
2) Order variants; first by specificity, then by performance and other criteria

The dispatch meta-wrappers provide a level of indirection between applications and specific function intrinsics that facilitates the automatic selection of implementations for particular input and machine configurations. The ordering represents the Merge compiler's inference of a "good" mapping.

The meta-wrappers support two usage modes:

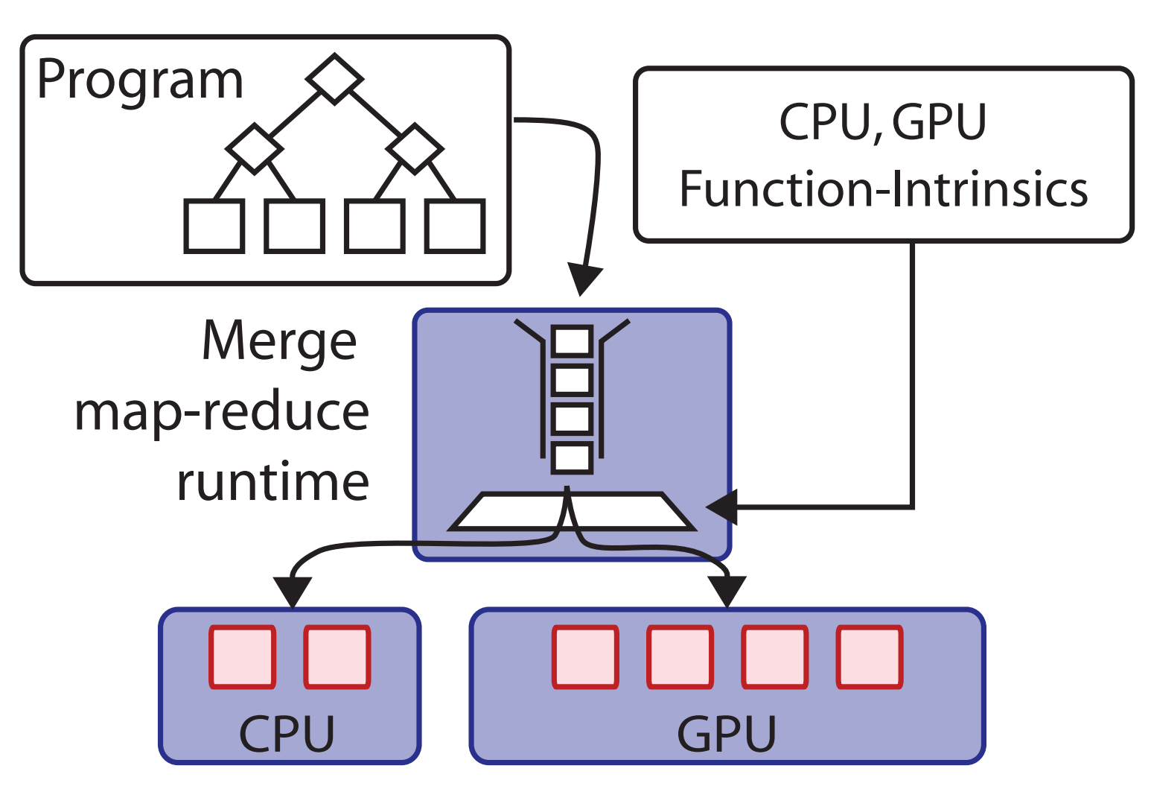
1) Direct Usage: Directly call the simple wrapper show above. Completely hides any heterogeneity from the end-user. Intended for programmers who want the simplest, most backward-compatible programming model.

2) Indirect Usage: Other wrappers enable introspection into the available variants and their dispatch annotations. Programmers can write their own domain-specific schedulers.

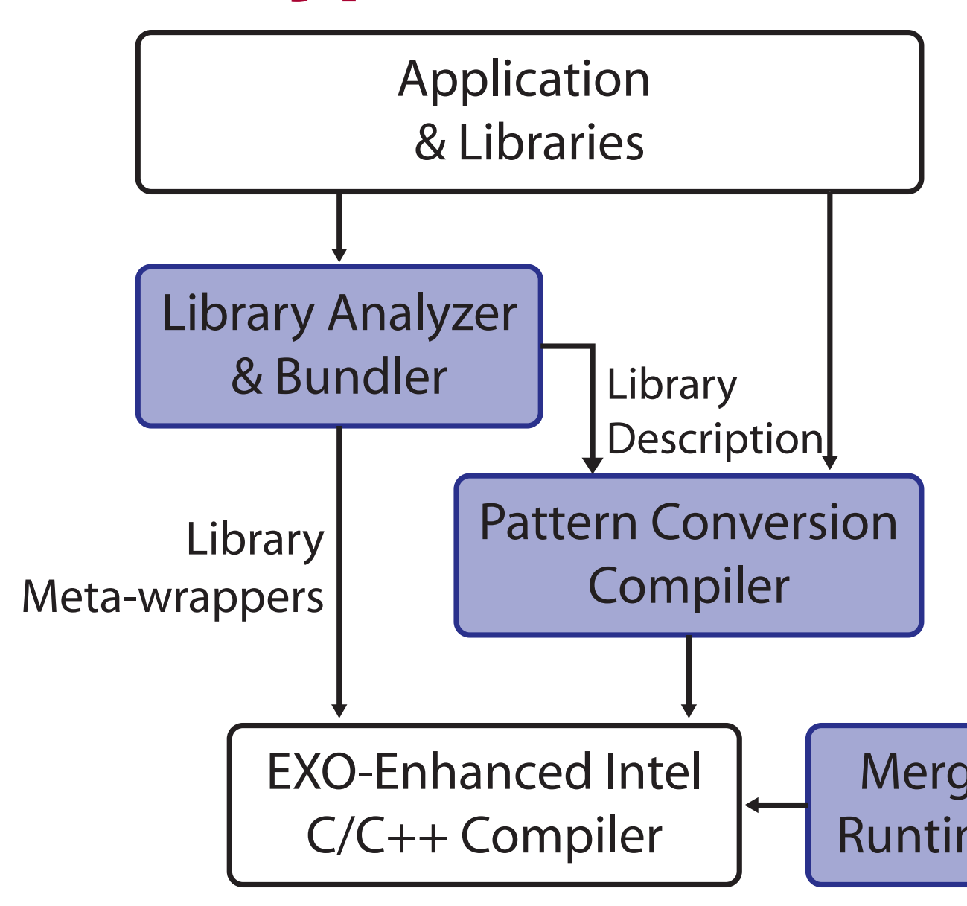
Parallel Frameworks

High-level parallel programming languages help raise the level of abstraction for both expert and non-expert programmers. The Merge prototype includes a parallel language built around map-reduce semantics.

The map-reduce language is encapsulated inside C/C++ functions, and implements a specialized scheduler that attempts to improve performance by dynamically distributing work across different processors in the system.



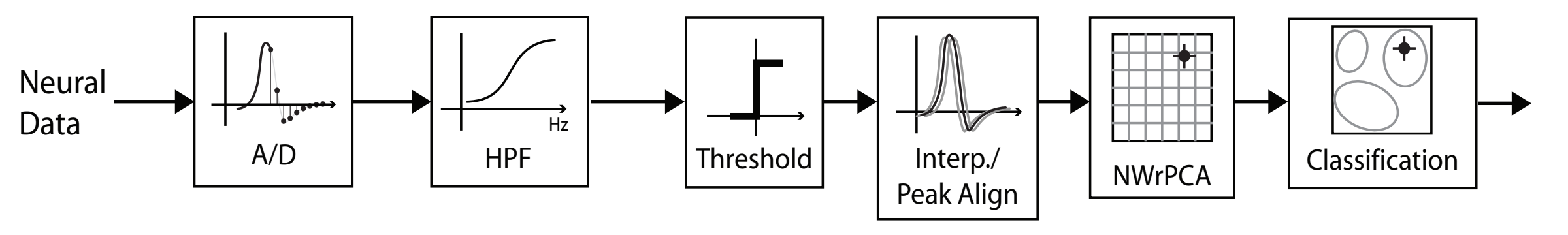
Prototype


Application & Libraries
Library Analyzer & Bundler
Library Description
Library Meta-wrappers
Pattern Conversion Compiler
EXO-Enhanced Intel C/C++ Compiler
Merge Runtime

Two test plaforms:
1) Core 2 Duo CPU & Intel X3000 integrated GPU
2) Unisys 32-way SMP with Intel Xeon processors

Application Example

Real-time Spike Sorting for Neural Prosthetics

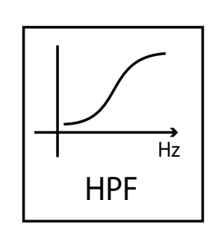


Neural prosthetics systems restore lost communication and movement functionality for patients with spinal cord injuries or neural degenerative diseases.

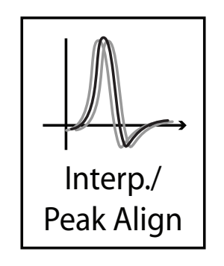
The prosthetic is controlled by neural signals recorded with electrodes implanted in the cortex. As the number of electrodes grows, real-time spike sorting, which disambiguates among different neurons recorded on the same channel, is becoming increasingly computationally demanding.

Each stage implements well-understood kernels that can be parallelized by computing experts within and across channels.

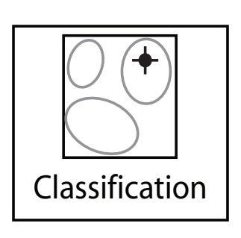
Function Variants



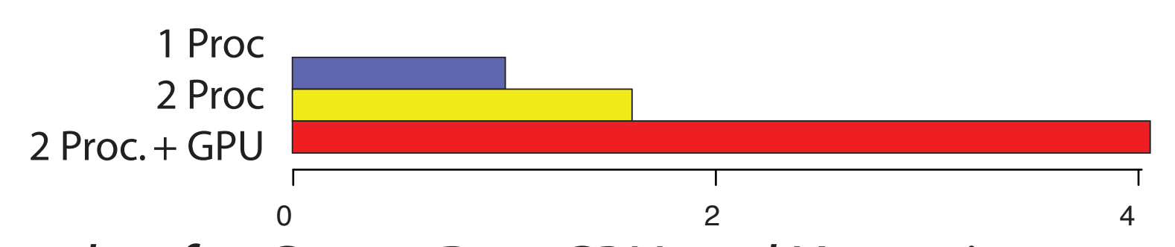
1. Single channel IIR filter for single processor
2. Multi-channel IIR filter for single processor with SSE



1. Single channel FFT for single processor
2. Multi-channel FFT for single processor with SSE



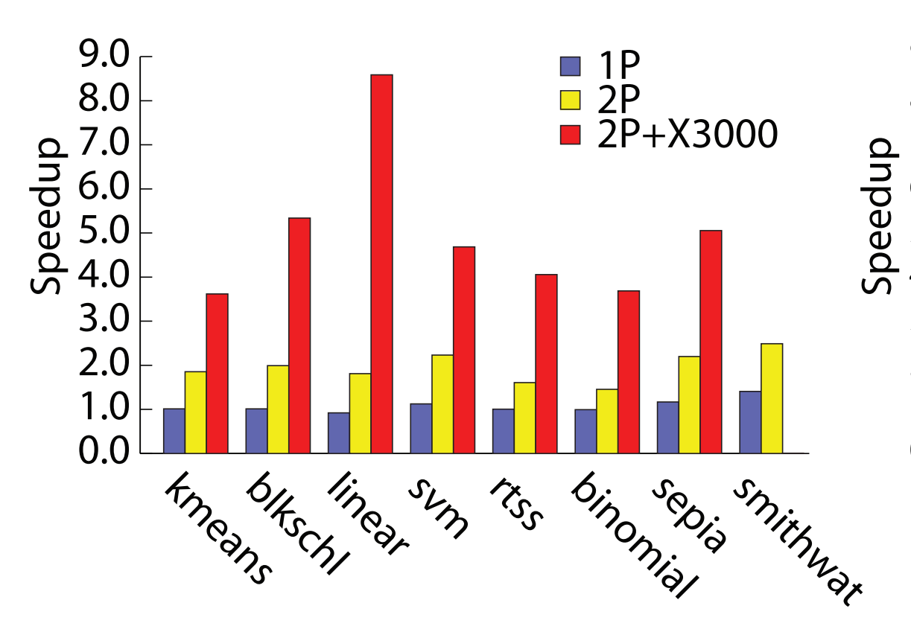
1. Single channel k-means for single processor
2. Multi-channel k-means for GPU



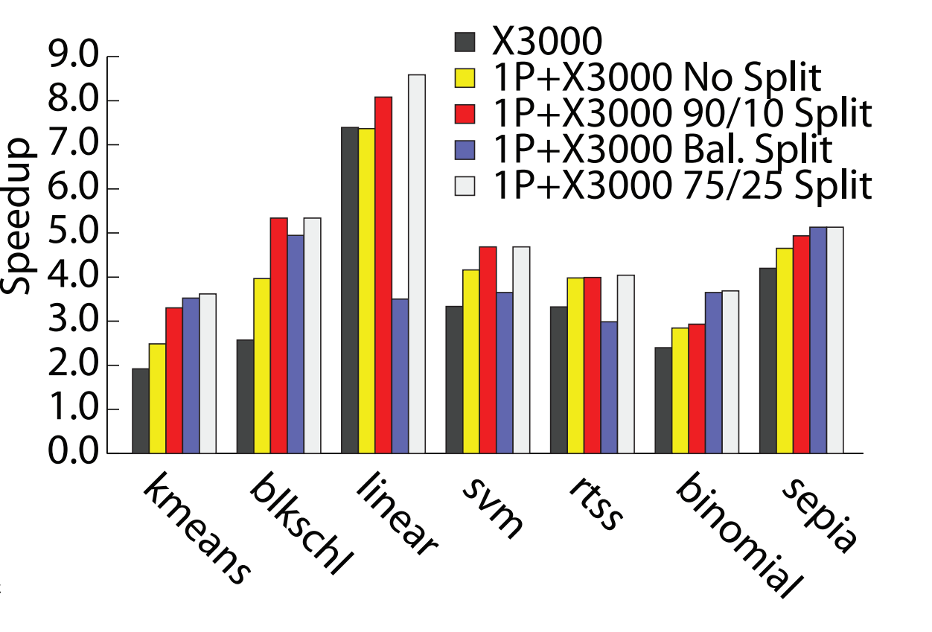
Speedup for Core 2 Duo CPU and X3000 integrated GPU

Results

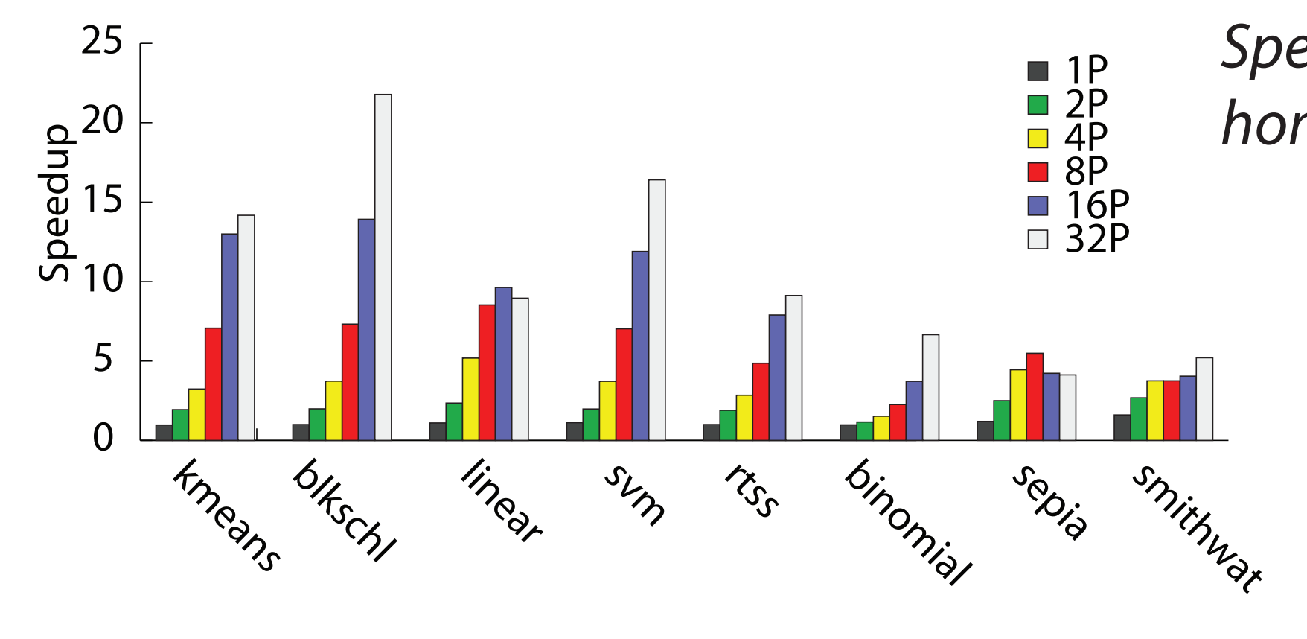
We implemented a set of kernels on both heterogeneous and homogeneous test platforms. Significant speedups relative to the single-core reference implementation are achieved on both platforms using the same source program.



Speedup for Core 2 Duo CPU and X3000 Integrated GPU



Speedup for different work distribution strategies



Speedup on 32-way homogeneous SMP

Further Reading

M. D. Linderman, J. Collins, H. Wang, T. H. Meng, "Merge: A programming model for heterogeneous multicore systems", *Proc. of ASPLOS*, 2008.
M. D. Linderman, J. Balfour, T. H. Meng, W. J. Dally, "Embracing heterogeneity - Parallel programming for changing hardware", *Proc. of HOTPAR*, 2009.

Acknowledgements

This work is funded in part by the FCRP Fcocus Centerfor Circuit and Systems Solutions (C2S2), under contract 2003-CT-888, and the Intel Corporation.