Reconceptualizing to Unshackle Programmers from the Burden of Exascale Hardware Issues

Richard A. Lethin, Reservoir Labs, Inc.
Acknowledgements

Reservoir team: Benoit Meister, Nicolas Vasilache, David Wohlford, Muthu Baskaran, John Ruttenberg, Jordi Ros-Giralt, Pete Szilagyi, Patrick Clancy, Jonathan Springer, Jim Ezick, Ann Johnson, Stefan Freudemberger, Melanie Peters, Nicole Bender, Trevor Serfass

Sponsors: ACS, DARPA, DOE, DOD, Others

Primes and Partners (UHPC team): Intel, UIUC, U Delaware, ETI, SDSC
The Exascale Hardware Opportunity And Burden For Programmers
8nm Process Technology

Core

- CE or XE
  - 0.17 mm² (50%)
- Memory
  - 0.35 MB
  - 0.17 mm² (50%)

~0.6 mm

20 mm

Processor Module

- DRAM Capacity: 128-512 GB
- DRAM Bandwidth: 1 TB/s
- Peak perf.: 10 TF
- DRAM Power: 20 W
- Total Power: 124 to 368 W
- Energy efficiency: 81 to 27 GF/Watt

DRAM Power: 20 W

Total Power: 124 to 368 W

Energy efficiency: 81 to 27 GF/Watt

Goal: 50 GF/W

Goal: 80 GF/W

Processor Node

- DRAM Capacity: 32-128 GB
- DRAM Bandwidth: 64-256 GB
- Peak perf.: 2.5 TF
- DRAM Power: 20 W
- Total Power: 124 to 368 W
- Energy efficiency: 81 to 27 GF/Watt

Goal: 50 GF/W

Goal: 80 GF/W

Logic transistors: 2 M
Core memory: 0.35 MB
Vdd: 0.41 V
Frequency: 1.2 GHz
Peak perf.: 2.4 GF
Power: 0.24 to 0.46 W

Cores/Module: 1152
On-die Memory: 400 MB
Vdd: 0.41 V
Frequency: 1.2 GHz
Peak perf.: 2.5 TF
Power: 26 to 87 W
Energy efficiency: 96 to 29 GF/Watt

8nm Process Technology
Hardware Building Blocks
Runnemede Hardware System

(a) Processor Node
10 cm
10 TF Peak

High Density Interconnect & Package

(b) System Node
19” PCB
160 TF Peak

(c) Cabinet
16 boards/cabinet
~1M cores
~2.5 PF Peak
~51 KW
50 GF/Watt

16 boards x 4” = 64”
Fits in 72” cabinet

9 cm

4”
Voltage Scaling

When designed to voltage scale

Energy Efficiency
Freq
Total Power
Leakage
Near Threshold Logic

H. Kaul et al, 16.6: ISSCC08
Extreme Scale Programming Challenges (Part 1)

Low Power
- Near Threshold Voltage operation => parallelism “1000x”
- Maximizing locality
- Very high variation in transistor performance
- Explicit communications, synchronization
- Heterogeneous, hierarchical architecture

Resilience
- More transistors, smaller transistors, operating at margins

New features
- Reorganized / refactored memory system
- New collectives / programmable operators
New Execution Model “e.g., Codelets”

- Fine-grained, event-driven, non-blocking
- Fuse “intra-node” and “inter-node” abstractions
  - Global memory abstractions, RDMA
- Explicit and implicit communications
  - All operands “ready” when codelet fires, results streamed out after codelet finishes
- Dynamic load balancing, other advanced schedulers
Who/what deals with this complexity?

Not the Programmer!

- Expressing all of these considerations will make the program longer, buggier
- Opaque to any semantic or dependence analysis needed for optimization
- Will over-specify the program and bake it to one architecture, defeating portability
- Exascale programming will be too complicated – VLIW lessons
Who/what deals with this complexity?

Don’t put it in the libraries!

• Optimization through and across library calls is an essential place to get performance
• Cross call fusion for locality
• Libraries cannot be opaque
• If they have this complexity in their code they will be opaque
It can be done in a compiler

Sequential C Kernel Source

Your favorite compiler here.

VM Abstraction/Back End Compilers

OpenMP Cell DataFlow CUDA

Existing Automated Optimizations

Parallelization
Locality optimization
Tiling
Placement
Distributed local memory opt
Memory promotion
Corrective array expansion
Layout optimization
Reshaping
Communication (DMA) generation
Multi-buffering
Synchronization generation
Thread generation
Hierarchical targets
Heterogeneous targets
Multiple execution models
Loop transformations as scheduling

iteration space of a statement $S(i,j)$

Schedule $\theta$ maps iterations to multi-dimensional time

A feasible schedule must preserve dependencies

Loop transformations/synthesis mean generating code to execution iterations of a loop in the lexicographical order of time
Scheduling state of the art 2011

Joint parallelization + locality + contiguity optimization
Can generate nested parallelism (nested OpenMP)
Explicit management of scratchpad memories
Virtual scratchpads
Explicit communication generation and optimization
Integrated scheduling plus placement/layout optimization
Hierarchical scheduling
Placement
Task formation
Granularity selection
Heterogeneous targets
Hybrid static / dynamic scheduling

... (Reservoir, UIUC, OSU, PSU, Rice, UCB, USC/ISI, CU ...)
The answers

High-level, semantics-rich programming approaches
- Math languages, Ptolemy, ...
- Specify the “what” not the how
- Commutability, reassociability, ...
- Accuracy requirements
- Reformulation algebras
- Numerical methods ontology
- Domain knowledge (symmetry, structure, bounds, ...)

Separate tuning languages from application language
- E.g., Intel Concurrent Collections
- Auto-generate tuning languages
High-level semantics-rich (and architecture-lite) expression provides

Optimizability

Understandability

Verifiability

Longevity

Portability

Composability
This is NOT!

Writing programs in CUDA, OpenMP, ... directly, or hybrids.

- Programmers should not touch these forms – they are TOXIC to automatic portability, optimization, parallelization, and will be costly in the long run.
- These forms should be (and can be EASILY) auto-generated (and auto-tuned) from high-level, semantics-rich high level form.
Migration path for existing codes

Semantic pragmas

Not mapping pragmas

Selective rewrites to high-level, semantics-rich form
Needed exascale research

High-level semantics rich expression languages.

Automated transformations to utilize them.

Automated transformations to address exascale hardware issues.

These benefit all levels of extreme scale systems – Tera, Peta, Exa -- UBIQUITOUS
Verification as a driver

Can system check proof that an application's implementation is correct with respect to dynamism, faults, precision?

Use this as a litmus test for the programming language, annotation system, tools, runtime, and hardware.

E.g., should we off-the-bat be reasoning about asynchrony?

Uncertainty Quantification on steroids

Proof, logic, knowledge
Next Generation Compiler

Applications Spec

- High Level Applications Specification (Math language)
- Mapping Annotations (Hierarchical Tilings, etc.)
- Correctness Proof Assertions

Compiler

- Algorithm/Formulation Exploration
- Irregular / Amorphous Algorithm Compiler
- Polyhedral Optimizer
- Proof Preserving Parallel Intermediate Representation
- Numerical Algorithms Ontology
- Math Ontology
- Proof Assistant

Dynamic Machine Model

Runtime

- Online Profiler
- Online Parameterized Mapping Projector
- Security and Resilience Checker
- Correctness Checker
Supporting New Execution Models

Intel Concurrent Collections (CnC) [Knobe]

- Dynamic single assignment – maximal algorithmic parallelism “Domain Expert”
- Separate tuning language “Tuning Expert”
  - Granularity selection, placement, scheduling
- Runtimes for multi-core, distributed systems
Why CnC?

We've built a proof of concept auto-generator for CnC

- Can express more parallelism than possible with OpenMP
- Get the benefit of adaptive load balancing from runtime
- Expressions in CnC are significantly more succinct than OpenMP

Provides a natural framework to integrate optimization of algorithm structure and data structures (irregular, unstructured)

- Graphs, sparse matrices, meshes

Natural implementation that is fault tolerant
Supporting new directions in tuning languages

CnC Hierarchical affinity groups (Knobe, Sarkar)
Express tuning/mapping as constraints and affinities

Research question: how can we produce schedules more dynamic and less constrained than polyhedral $\theta$

(sally: i) → [w] → (sonia: i, j) → [x] → (Sanjay: i, j) → [y] → (Simon: i, j) → [z] → (sam: i)
Data Layout / Placement / Format / Notations

Hierarchically Tiled Arrays (HTA) (Padua/Garzaran)
Format for dense matrix algs
Associated programming notations
Recursive algorithms
Cache oblivious algorithms

Research challenges
Automatic Scheduling + Data layout optimization?
Dynamic – Massive Challenge
High level notations->recursive formulations
New Data Structures (NDS)
Improved runtime schedulers

J-Machine 8x8x16 = 1024 “cores” circa 1990

Questions about regulation of “clouds” of computation relate to today's efforts to more tightly couple CPU+NW, queues.

Research topic
How can we make modern task scheduler technology incorporate network regulation constraints?

Need compiler
Reconceptualizing

You can't raise the level of abstraction too high.

Semantics-rich, high-level programming is the way to go.

Much research to be done, but the right path is clear:

Each step in the high-level direction facilitates automatic compiler and runtime solutions to increase parallelization, performance, efficiency, confidence, lifetime...

...and will save tons of money, ubiquitously.