Optimizing Data Layouts For Irregular Applications on a Migratory Thread Architecture

Thomas Rolinger\textsuperscript{1,2}, Christopher Krieger\textsuperscript{2}, Alan Sussman\textsuperscript{1}

\textsuperscript{1} University of Maryland
\textsuperscript{2} Laboratory for Physical Sciences
Overview

• Emerging memory-centric architectures
  – fundamentally different from conventional systems
  – require different approaches to obtain high performance

• This work: optimizing data layouts
  – focus on applications with irregular data access patterns
    • makes static optimizations hard
Outline

• Migratory Thread Architecture
  – Emu
  – Data Layouts

• Framework for optimizing data layouts
  – cost model
  – optimization: block placement

• Case Study
  – Sparse Matrix Vector Multiply
Migratory Thread Architecture

• Studied in this work: **Emu**
• “Cheaper” to move program instead of data
• Threads **migrate** to remote data on **reads**
  – migration context: ~ 200 bytes (live registers, PC)
  – stores performed as remote updates (thread does not migrate)
  – no direct analogue to this on conventional systems
• Consequences
  – data layout directly impacts work distribution and hardware load balancing
    • load balance != equally distributing data
    • cannot pin/isolate threads to hardware resources
The Emu Architecture

• Gossamer Core (GC)
  – cache-less
  – supports up to 64 concurrent light-weight threads
• Nodelets combined to form nodes
• Threads move between nodelets
  – intra-node: migration engine
  – inter-node: Serial RapidIO link(s)
• Partitioned Global Address Space (PGAS)
• Migrations performed by hardware
  – no user intervention
The Emu Architecture

- **Gossamer Core (GC)**
  - cache-less
  - supports up to 64 concurrent light-weight threads
- **Nodelets** combined to form **nodes**
- Threads move between nodelets
  - intra-node: migration engine
  - inter-node: Serial RapidIO link(s)
- Partitioned Global Address Space (PGAS)
- Migrations performed by hardware
  - no user intervention

**Single Node with 4 nodelets**
The Emu Architecture

- **Gossamer Core (GC)**
  - cache-less
  - supports up to 64 concurrent lightweight threads
- **Nodelets** combined to form **nodes**
- Threads move between nodelets
  - intra-node: migration engine
  - inter-node: Serial RapidIO link(s)
- Partitioned Global Address Space (PGAS)
- Migrations performed by hardware
  - no user intervention

**System used in our work: Emu Chick**
8 nodes (32 nodelets), Arria10 FPGA hardware
Nodes requiring two hops:

0 ↔ 7  1 ↔ 6  2 ↔ 5  3 ↔ 4
Data Layouts on Emu

• Data layout is everything on Emu
  – data layout is the only “knob” we can control

• Performance metrics to characterize “bad” layouts:
  – induces many thread migrations
    • migrations are expensive
  – induces poor load balancing
    • thread migration hot spots
Problem Statement

• **Question**: How should we lay out our data to achieve high performance?

• **Answer**: it depends on the data access pattern of the application
  – not known until runtime for *irregular applications*

• Proposed framework
  – *profile-driven* data layout optimizations

• Focus of this work: block distributions
  – chunk up data into blocks and distribute blocks to nodelets
  – blocks can have different sizes
Profile-driven Data Layout Optimizations

program \( P \) that accesses data \( D \)

Emu simulator

Data-centric memory profiler

Emu hardware

Optimization(s)

Cost model

specify that we want to optimize \( D \)

memory trace

memory access profile for \( D \)

memory access costs

costs for each block of \( D \) and each nodelet

new layout for \( D \)

test new layout on hardware
Profile-driven Data Layout Optimizations

Focus of this talk

- test new layout on hardware
- Optimization(s)
- Cost model

new layout for $D$

costs for each block of $D$ and each nodelet

In full paper: details of data-centric profiler and block distribution library
Optimization: Block Placement

• Optimization to consider:
  – **INPUT**: existing data layout (mapping of blocks to nodelets)
  – **OUTPUT**: new data layout
  – move a block from its original nodelet to another such that its **total memory access cost is reduced**
  – but also **avoid creating migration hot spots**

• Need a cost model to help guide optimization
  – This talk → high level overview
  – Full paper → more formal description
Cost Model

• **Step 1.** How much does an access cost?

Measure cycles per access between nodelets
Produced by hardware benchmark
Cost Model

• **Step 1.)** How much does an access cost?
Cost Model

- **Step 1.** How much does an access cost?

![Diagram](image)

- **src nodelet**
- **dest nodelet**

- **node 0, nodelets 0 – 3**

- **intra-node migrations**
  - **2x more cycles than locals**

- **locals (diagonal)**
  - **1x**
Cost Model

• **Step 1.** How much does an access cost?

![Diagram showing cost model with nodelets and migrations]

- **locals (diagonal):** 1x
- **intra-node migrations:** 2x more cycles than locals
- **inter-node migrations:** 3x more cycles than locals
Cost Model

• **Step 1.** How much does an access cost?

![Diagram showing cost model with nodes and nodelets labeled.]

- **Node 0, nodelets 0–3:** inter-node migrations, Two hops, **4x more cycles than locals**
- **Node 7, nodelets 28-31:** inter-node migrations, Single hop, **3x more cycles than locals**
- **Intra-node migrations:** **2x more cycles than locals**
- **Locals (diagonal):** **1x**
Cost Model (cont.)

• **Step 2.** What is the memory access cost for each block?
  – For a given nodelet $i$ and block $b$ on nodelet $j$
    • ($\#$ accesses to $b$) $\times$ (cost of access from $i$ to $j$)
  – Sum up across all nodelets to get a “total” latency for the block, measured in cycles
Cost Model (cont.)

• **Step 2.** What is the memory access cost for each block?
  
  – For a given nodelet $i$ and block $b$ on nodelet $j$
    
    • (# accesses to $b$) $\times$ (cost of access from $i$ to $j$)
  
  – Sum up across all nodelets to get a “total” latency for the block, measured in cycles

Steps 1—2 tell us how to find nodelet that will give the lowest memory access cost for a given block

But need way to consider load balancing of resources
Cost Model (cont.)

• **Step 3.** How are the threads moving around?
  – Memory profiler provides info about how threads access blocks (and nodelets) over time
Step 3.) How are the threads moving around?

– Profiler provides info about how threads access blocks (and nodelets) over time

![Threads Accessing Blocks](chart)

- Maximum threads supported per nodelet → 192
Cost Model (cont.)

• **Step 3.** How are the threads moving around?
  – Profiler provides info about how threads access blocks (and nodelets) over time

![Threads Accessing Blocks](chart)

- **block 0**: heavy load
  - Hard to find acceptable nodelet placement

- **block “load”**: independent of its placement
Cost Model (cont.)

- **Step 3.** How are the threads moving around?
  - Profiler provides info about how threads access blocks (and nodelets) over time.

Active Threads on Nodelets

- **nodelet 0:** heavy “load” → probably not good to relocate blocks there.
- **nodelet “load”** → aggregation of all blocks on nodelet as well as all other activity.
Cost Model (cont.)

• **Step 4.** Compute performance impact of each block → prioritize the blocks
  – Experiments showed that attempting to move all blocks is generally bad
  – Also found that the order in which we attempt to move blocks is crucial

• Considers memory access latency (based on its current placement) as well as block load
Cost Model (cont.)

• **Step 5.** Compute placement cost of block $b$ on nodelet $n$
  
  – Considers both memory access latency and the resulting load on nodelet $n$ **IF** block $b$ were to be placed on nodelet $n$
  
  – Does not require re-running or profiling of application to compute $\rightarrow$ relies on existing profiler data only
Optimization Algorithm

• See full paper for details
• Basic idea:
  – prioritize/sort blocks based on performance impact
  – Place block $b$ on the nodelet $n$ that gives the lowest placement cost
• Update model between placements
  – does not require re-running the program
• Complexity: $O(\log B + B N^2)$
  – $N = \# \text{ nodelets} = 32$ (not data dependent)
  – $B = \# \text{ blocks}$
  – Common case $B = N \rightarrow O(N^3)$
Case Study: SpMV

• Sparse Matrix Vector Multiply
  – fundamental kernel in graph analytics
• $Ax = b$
  – $A \rightarrow$ sparse matrix
  – $x \rightarrow$ dense input vector
  – $b \rightarrow$ dense output vector
• $x$ is split into equal sized blocks
  – default layout (block $i$ on nodelet $i$).
• System: 32 nodelets (8 nodes), 192 threads per nodelet $\rightarrow$ 6,144 migrating threads total
SpMV Performance Gains
New Data Layouts Vs Default

speed-up over default

HEP-th-new  cop20k_A  Dubcova3  web-NotreDame  inline_1  loc-Gowalla  FullChip  mip_1  gupta3  crankseg_1  mycielskian15  vsp_finan512  mycielskian13  parabolic_fem  wiki-Talk  crankseg_2  net150  Hamrle3  patents  human_gene1  Freescale1  patents_main
SpMV Performance Gains

New Data Layouts Vs Default

Already had existing migration hot spot
→ current optimization does not “fix” hot spots
→ best addressed with another optimization
Future Work

• Consider more optimizations
  – copy/replicate blocks
  – adjust block sizes
  – optimize layout for more than one data structure at a time

• More refined cost model
  – better understanding of thread activity
  – consider memory consumption

• Evaluate more applications

• Runtime optimizations
  – not feasible with current Emu hardware
Conclusions

• Data placement is crucial to performance on migratory thread architectures
  – fundamental differences in how to approach data layouts when compared to conventional systems

• Our framework is application independent
  – relies on memory trace analysis and cost model
  – target use cases: iterative applications
    • cost of profiling/optimization can be amortized
• Emerging memory-centric architectures
  – fundamentally different from conventional systems
  – require different approaches to obtain high performance

• This work: optimizing data layouts
  – focus on applications with irregular data access patterns
  – run → profile → model → optimize

Contact:
Thomas Rolinger (tbrolin@cs.umd.edu)