Era of Customization and Specialization for Energy-Efficient Computing

Jason Cong
Director, Center for Domain-Specific Computing
www.cdsc.ucla.edu

Chancellor’s Professor, UCLA Computer Science Department
cong@cs.ucla.edu
Is Semiconductor a Sunset Industry?

- Frequency scaling has stopped a decade ago due to power barrier
- CMOS device scaling is coming to the end soon, with no sure replacement in sight

Source: Shekhar Borkar, Intel
Current Solution to Power Barrier

- 10’s to 100’s cores in a processor
- 1000’s to 10,000’s servers in a data center
**Cost and Energy are Still a Big Issue ...**

Hiding in Plain Sight, Google Seeks More Power

![Image of computing center](image.jpg)

Google is building two computing centers, top and left, each the size of a football field, in The Dalles, Ore.

By JOHN MARKOFF and SAUL HANSELL
Published: June 14, 2006

THE DALLES, Ore., June 8 — On the banks of the windswept Columbia River, Google is working on a secret weapon in its quest to dominate the next generation of Internet computing. But it is hard to keep a secret when it is a computing center as big as two football fields, with twin cooling plants protruding four stories into the sky.
Next Big Opportunity – Customization and Specialization

Adapt the architecture to Application domain.
## Justification 1 – Potential of Customization

<table>
<thead>
<tr>
<th>AES 128bit key 128bit data</th>
<th>Throughput</th>
<th>Power</th>
<th>Figure of Merit (Gb/s/W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.18mm CMOS</td>
<td>3.84 Gbits/sec</td>
<td>350 mW</td>
<td>11 (1/1)</td>
</tr>
<tr>
<td>FPGA [1]</td>
<td>1.32 Gbit/sec</td>
<td>490 mW</td>
<td>2.7 (1/4)</td>
</tr>
<tr>
<td>ASM StrongARM [2]</td>
<td>31 Mbit/sec</td>
<td>240 mW</td>
<td>0.13 (1/85)</td>
</tr>
<tr>
<td>Asm Pentium III [3]</td>
<td>648 Mbits/sec</td>
<td>41.4 W</td>
<td>0.015 (1/800)</td>
</tr>
<tr>
<td>C Emb. Sparc [4]</td>
<td>133 Kbits/sec</td>
<td>120 mW</td>
<td>0.0011 (1/10,000)</td>
</tr>
<tr>
<td>Java [5] Emb. Sparc</td>
<td>450 bits/sec</td>
<td>120 mW</td>
<td>0.0000037 (1/3,000,000)</td>
</tr>
</tbody>
</table>

---

- [1] Amphion CS5230 on Virtex2 + Xilinx Virtex2 Power Estimator
- [4] gcc, 1 mW/MHz @ 120 Mhz Sparc – assumes 0.25 u CMOS
- [5] Java on KVM (Sun J2ME, non-JIT) on 1 mW/MHz @ 120 MHz Sparc – assumes 0.25 u CMOS

Justification 2 -- Advance of Civilization

- For human brain, Moore’s Law scaling has long stopped
  - The number of neurons and their firing speed did not change significantly

- Remarkable advancement of civilization via specialization
  - More advanced societies have a higher degree of specialization
Our Goals

❖ A general, customizable platform
  ▪ Can be customized to a wide-range of applications
    • May focus on one or several given domains
  ▪ Can be massively produced with cost efficiency
  ▪ Can be programmed efficiently with novel compilation and runtime systems

❖ Metric of success
  ▪ A “supercomputer-in-a-box” with 100X performance/power improvement via customization for the intended domain(s)
**Example of Customizable Platforms: FPGAs**

- Configurable logic blocks
- Island-style configurable mesh routing
- Dedicated components
  - Specialization allows optimization
  - Memory/Multiplier
  - I/O, Processor
  - Anything that the FPGA architect wants to put in!

More Opportunities for Customization to be Explored

Key questions: Optimal trade-off between efficiency & customizability
Which options to fix at CHP creation? Which to be set by CHP mapper?

- **Core parameters**
  - Frequency & voltage
  - Datapath bit width
  - Instruction window size
  - Issue width
  - Cache size & configuration
  - Register file organization
  - # of thread contexts
  - ...

- **Cache parameters**
  - Cache size & configuration
  - Cache vs SPM
  - ...

- **NoC parameters**
  - Interconnect topology
  - # of virtual channels
  - Routing policy
  - Link bandwidth
  - Router pipeline depth
  - Number of RF-I enabled routers
  - RF-I channel and bandwidth allocation
  - ...

- **Custom instructions & accelerators**
  - Shared vs. private accelerators
  - Choice of accelerators
  - Custom instruction selection
  - Amount of programmable fabric
  - ...

---

Our Proposal: Customizable Heterogeneous Platform (CHP)

- Fixed Core
- Custom Core
- Prog Fabric
- Custom Core
- Fixed Core
- Fixed Core
- Reconfigurable RF-I bus
- Reconfigurable optical bus
- Transceiver/receiver
- Optical interface
Examples of Customization

- Customization of processor cores
- Customization of on-chip memory
- Customization of on-chip interconnects
Example 1 – Customization of Cores

- Large cores or small cores?
- How many each type?
Core spilling – [Cong et al Trans. on Parallel and Distributed Systems 2007]

- CMP systems focus on improving overall throughput
  - Sequential or legacy applications might not see benefits
- Key idea – allow execution to be spilt from one core to next at run-time
  - Simulate increase in register file, instruction queue, ROB and LSQ size
  - Allocate cores intelligently to spilling core

Core A’s resources are exhausted.
Core A sends 62 compiler visible registers, a 24-entry store buffer, the LSQ entries of any stores that are in-flight, and a PC.

Core B begins fetching from the PC sent by Core A. Core A continues execution, and sends any register or store values that were in-flight at the time of the spill as they complete.

Spilling can continue if B’s resources are exhausted.

Eventually, all instructions on Core A will commit (unless there is an exception or branch misprediction) and Core A can be released into the pool of idle cores.
Core spilling – [Cong et al Trans. on Parallel and Distributed Systems 2007]

- **Results**
  - Core spilling achieves more than 50% of the performance of ‘ideal’ 32-issue core by using 4-issue cores for single applications
  - 39% improvement for multiple application workload
  - Up to 40% reduction in latency for changing workloads
Example 2: Customization of On-Chip Memory

- HW controlled cache or SW controlled cache (SPM)?
- How much to allocate for each type?
Customizable Hybrid L1 Cache [ISLPED’2011]

- Cache in conjunction with Scratchpad Memory (SPM) in L1
  - Cache: Hardware-controlled
    - Transparent to software: a fast local copy of the global memory address space
  - SPM: Software-controlled
    - Not transparent to software: a separate address space from the global address space

- Customizable
  - Flexibly size the cache and SPM based on the application requirements
    - Cache: dynamic/random access
    - SPM: regular data access pattern

![Cache memory organization](image1)
![SPM organization](image2)
![Comparison (2KB)](image3)
How to Customize?

- **Way-wise reconfigurable cache**
  - Configure several ways of cache as SPM
  - Column cache [Chiou et al. DAC’00]

- **Block-wise reconfigurable cache**
  - Virtual local store [Cook et al. UCB TR’09]
  - Unified mapping of SPM blocks onto cache blocks

- **Adaptive hybrid cache (AH-Cache)**
  - Dynamically remap SPM blocks from high-demand cache sets to low-demand cache sets.
Challenge 1: Fast SPM Location

- SPM Lookup and access in AH-Cache
  - SPM Mapping Lookup Table (SMLT)
    - Store SPM mapping information
  - Additional SMLT lookup stage
    - May increase the critical path
  - Zero-timing overhead SPM location
    - Hide SMLT lookup in EX pipeline with address generation

- The memory reference instructions to the SPM should be:
  - Base address: the base address of the SPM array
  - Offset: the offset related to the SPM base address
Challenge 2: Efficient Adaptive Mapping

- Circular bouncing effect
  - Cache set A holds an SPM block and becomes hot
  - Cache set A sends the SPM block to a cold set B
    - A enable the cache way as a regular cache block
    - B will evict one of its cache blocks to accommodate the SPM block
  - Cache set B becomes hot and sends the SPM block back to A
  - A->B->A->B....

- Key idea to avoid circular bouncing effect
  - Floating-block-holder (FBH) queue: record the cache sets currently holding the floating cache blocks
  - A hot set will not give up its floating blocks once re-inserted in the FBH queue in an adaptation interval

Only 4.4 SPM remapping on average per $10^6$-cycle interval for our benchmarks

- Less than 6% per-cache-access energy overhead
# Impact of Adaptation

- **Design points for comparison:**
  - Design $N$: Non-adaptive hybrid cache, baseline
  - Design $B$: $N +$ Balanced cache [Zhang, ISCA’06]
  - Design $V_p$: $N +$ Victim cache [Jouppi, ISCA’90]
  - Design $V_s$: $N +$ A serially accessed victim cache
  - Design $R$: Phase-reconfigurable hybrid cache [Zhang, et.al., ISLPED’02]
  - Design $AH$: AH-Cache
  - Design $S$: Static optimized hybrid cache (not practical, just check the optimality gap of AH)

## Improvements of AH-Cache

<table>
<thead>
<tr>
<th></th>
<th>$N$</th>
<th>$B$</th>
<th>$V_p$</th>
<th>$V_s$</th>
<th>$R$</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miss rate</td>
<td>52%</td>
<td>19%</td>
<td>22%</td>
<td>22%</td>
<td>33%</td>
<td>-1%</td>
</tr>
<tr>
<td>Run-time</td>
<td>18%</td>
<td>3%</td>
<td>4%</td>
<td>8%</td>
<td>12%</td>
<td>~0%</td>
</tr>
<tr>
<td>Energy</td>
<td>13%</td>
<td>16%</td>
<td>22%</td>
<td>10%</td>
<td>7%</td>
<td>-1%</td>
</tr>
<tr>
<td>ED product</td>
<td>33%</td>
<td>19%</td>
<td>25%</td>
<td>18%</td>
<td>18%</td>
<td>~0%</td>
</tr>
</tbody>
</table>
Example 3: Customization of On-Chip Interconnects

- How many wires to include for on-chip communication?
- Uniform distribution and dedicated connections?
Our Answer: Use of Multiband RF-Interconnect for Customization

- In TX, each mixer up-converts individual baseband streams into specific frequency band (or channel)
- N different data streams (N=6 in exemplary figure above) may transmit simultaneously on the shared transmission medium to achieve higher aggregate data rates
- In RX, individual signals are down-converted by mixer, and recovered after low-pass filter
Terahertz VCO in 65nm CMOS

- Demonstrated an ultra high frequency and low power oscillator structure in CMOS by adding a negative resistance parallel tank, with the fundamental frequency at 217GHz and 16.8 mW DC power consumption.
- The measured 4th and 6th harmonics are about 870GHz and 1.3THz, respectively.

Higher harmonics (4th and 6th harmonics) may be substantially underestimated due to excessive water and oxygen absorption and setup losses at these frequencies.

“Generating Terahertz Signals in 65nm CMOS with Negative-Resistance Resonator Boosting and Selective Harmonic Suppression” Symposium on VLSI Technology and Circuits, June 2010
Mesh Overlaid with RF-I [HPCA ’08]

- 10x10 mesh of pipelined routers
  - NoC runs at 2GHz
  - XY routing
- 64 4GHz 3-wide processor cores
  - Labeled aqua
  - 8KB L1 Data Cache
  - 8KB L1 Instruction Cache
- 32 L2 Cache Banks
  - Labeled pink
  - 256KB each
  - Organized as shared NUCA cache
- 4 Main Memory Interfaces
  - Labeled green
- RF-I transmission line bundle
  - Black thick line spanning mesh
RF-I Logical Organization

• Logically:
  - RF-I behaves as set of N express channels
  - Each channel assigned to src, dest router pair (s,d)

• Reconfigured by:
  - remapping shortcuts to match needs of different applications
**Power Savings [MICRO ’08]**

- We can thin the baseline mesh links
  - From 16B…
  - …to 8B
  - …to 4B

- RF-I makes up the difference in performance while saving overall power!
  - RF-I provides bandwidth where most necessary
  - Baseline RC wires supply the rest

![Diagram showing network connections and bandwidth requirements](image)
Impact of Using RF-Interconnects [MICRO ’08]

- **Adaptive RF-I enabled NoC**
  - Cost Effective in terms of both power and performance
Specialization is also Important

- Specialized accelerators used to be considered “wasteful”
  - A story
- Van Neumann architecture maximizes device reuse
- Utilization wall
  - 6.5% utilization for a 45 nm chip filled with 64bit operators, assuming a power budget of 80 W [ASPLOS’2010]
Our Proposal: Extensive Use of Accelerators [SAW’2011]

- Proposed solution: extensive use of accelerators (customized or implemented using programmable fabric)
  - Sea of accelerators

- Type of accelerators:
  - Tightly vs. loosely coupled

- Benefits
  - Better performance
  - Higher power-efficiency
  - It’s ok to be “wasteful”

- Critical needs:
  - Efficient accelerator management
    - Scheduling
    - Sharing
Using Accelerators with OS Management

- Managing accelerator by OS is expensive
- In an accelerator rich CMP, management should be cheaper both in terms of time and energy

<table>
<thead>
<tr>
<th>Operation</th>
<th>Latency (# Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 core</td>
</tr>
<tr>
<td>Invoke</td>
<td>214413</td>
</tr>
<tr>
<td>RD/WR</td>
<td>703</td>
</tr>
</tbody>
</table>
Overall Architecture of AXR-CMP

- Architecture of AXR-CMP:
  - Multiple cores and accelerators
  - Global Accelerator Manager (GAM)
  - Shared L2 cache banks and NoC routers between multiple accelerators
1. The core requests and the GAM responds with a list (lcacc-req).
2. The core reserves (lcacc-rsv) and waits.
3. The core shares a task description to accelerator in memory and starts it (lcacc-cmd), the accelerator reads the task description, and begins working.
4. When the accelerator finishes its current task it notifies the core. The core then sends a message to the GAM freeing the accelerator (lcacc-free).
Light-weight Interrupt Support

◆ To reduce OS interrupt service
  ◆ No need to save context

◆ Two main components added:
  ▪ A table to store ISR info
  ▪ An interrupt controller to queue and prioritize incoming interrupt packets

◆ Each thread registers:
  ▪ Address of the ISR and its arguments
  ▪ lw-int source

◆ Sources of lw-int:
  ▪ GAM responses
    • Accelerator ready
    • Wait time for accelerator
  ▪ Accelerator TLB miss
  ▪ Accelerator task buffer empty

◆ Limitations:
  ▪ Only can be used when running the same thread which LW interrupt belongs to
  ▪ OS-handled interrupt otherwise

---

<table>
<thead>
<tr>
<th>Operation</th>
<th>Latency (# Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 core</td>
</tr>
<tr>
<td>Interrupt</td>
<td>16383</td>
</tr>
</tbody>
</table>

---

**lw-reg x y z**  
Register service routine y to service interrupts arriving from accelerator x. LWI message packet will be written to z

**lw-jump**  
Jump to the service routine for the next pending interrupt. Does nothing if no interrupt is pending.

**lw-ret**  
Return from an interrupt service routine.
Accelerator Chaining and Composition

- **Chaining**
  - To have an efficient accelerator to accelerator communication

- **Composition**
  - To create the virtual feeling of having larger accelerators for the applications
Hardware Overhead Analysis

- **AutoPilot to synthesize Global Accelerator Manager (GAM) module and DMA-C**
  - Area is less than 0.01% of the chip (1cm X 1cm) using 65 nm technology.

<table>
<thead>
<tr>
<th>Module</th>
<th>Clock speed</th>
<th>Area (u²m)</th>
<th>Power (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAM</td>
<td>2 ns</td>
<td>12270</td>
<td>2.64</td>
</tr>
<tr>
<td>DMA-C</td>
<td>2 ns</td>
<td>10071</td>
<td>0.09</td>
</tr>
</tbody>
</table>

C File ➔ AutoPilot ➔ Design Compiler ➔ Gate
**Experimental Results – Performance**

**(N cores, N threads, N accelerators)**

Performance improvement over SW only approaches:
on average 168X, up to 380X

Performance improvement over OS based approaches:
on average 51X, up to 292X
Experimental Results – Energy (N cores, N threads, N accelerators)

Energy gain over SW-only version

Energy gain over OS-based version

Energy improvement over SW only approaches: on average 241X, up to 641X

Energy improvement over OS based approaches: on average 17 X, up to 63X
**Experimental results - Increasing number of accelerators and data size**

* 8 cores, 8 threads
* Step-shape response when increasing the number of accelerators

* Increasing speedup when increasing data size
Experimental Results – Benefit of Light-Weight Interrupt

* Larger benefits for LW-Int when increasing the data size (D*D*D cube)
Experimental Results – Hardware GAM Benefit

* The best results is for registration (almost 2X), since the number of accelerators are more and it receives more requests.
* The lowest is for segmentation since it has only one accelerator, which makes it faster for software to manage (only 10% benefit).
3D Integration for Customization or Specialization

- **Vertical integration:**
  - CMP layer + customization/acceleration layer

- **Accelerators can directly access caches**
  - L1 or L2

- **Low latency**
  - 1 cycle traversal across the TSV bundle
  - 2-3 cycles to get from the TSV bundle to accelerator/controller

- **Higher bandwidth**
  - Almost equal to the bandwidth of the L1/L2 cache

- **No single bottleneck**
  - Each cache can have its own TSV bundle
    - Sharing TSV bundles possible

- **Early results:** medical imaging benchmarks [ASAP’2011]
  - > 7x performance gain
  - > 18x energy gain
Research Scope in CDSC (Center for Domain-Specific Computing)

Customizable Heterogeneous Platform

Domain-specific-modeling
(healthcare applications)

Design once

Invoke many times

CHP creation
Customizable computing engines
Customizable interconnects

CHP mapping
Source-to-source CHP mapper
Reconfiguring & optimizing backend
Adaptive runtime

Domain characterization
Application modeling

Architecture modeling

Customization setting
Center for Domain-Specific Computing (CDSC)
CHP Mapping – Compilation and Runtime Software Systems for Customization

Goals: Efficient mapping of domain-specific specification to customizable hardware

- Adapt the CHP to a given application for drastic performance/power efficiency improvement

Domain-specific applications

Abstract execution

Application characteristics

CHP architecture models

Domain-specific programming model
(Domain-specific coordination graph and domain-specific language extensions)

Programmer

Source-to source CHP Mapper

C/C++ code

Analysis annotations

C/SystemC behavioral spec

RTL Synthesizer (xPilot)

Reconfiguring and optimizing back-end

Binary code for fixed & customized cores

Customized target code

RTL for programmable fabric

Adaptive runtime
Lightweight threads and adaptive configuration

CHP architectural prototypes
(CHP hardware testbeds, CHP simulation testbed, full CHP)
**xPilot: Behavioral-to-RTL Synthesis Flow [SOCC’2006]**

- **Behavioral spec. in C/C++/SystemC**
- **SSDM**
  - **Advanced transformation/optimizations**
    - Loop unrolling/shifting/pipelining
    - Strength reduction / Tree height reduction
    - Bitwidth analysis
    - Memory analysis …
  - **Core behavior synthesis optimizations**
    - Scheduling
    - Resource binding, e.g., functional unit binding register/port binding
  - **Arch-generation & RTL/constraints generation**
    - Verilog/VHDL/SystemC
    - FPGAs: Altera, Xilinx
    - ASICs: Magma, Synopsys, …

- **Platform description**
- **Frontend compiler**
- **RTL + constraints**
- **FPGAs/ASICS**
AutoPilot Compilation Tool (based UCLA xPilot system)

- Platform-based C to FPGA synthesis
- Synthesize pure ANSI-C and C++, GCC-compatible compilation flow
- Full support of IEEE-754 floating point data types & operations
- Efficiently handle bit-accurate fixed-point arithmetic
- More than 10X design productivity gain
- High quality-of-results

Developed by AutoESL, acquired by Xilinx in Jan. 2011
Example: Versatile Scheduling Algorithm Based on SDC (DAC’06)

- Scheduling problem in behavioral synthesis is NP-Complete under general design constraints
- ILP-based solutions are versatile but very inefficient
  - Exponential time complexity
- Our solution: An efficient and versatile scheduler based on SDC (system of difference constraints)
  - Applicable to a broad spectrum of applications
    - Computation/Data-intensive, control-intensive, memory-intensive, partially timed.
    - Scalable to large-size designs (finishes in a few seconds)
  - Amenable to a rich set of scheduling constraints:
    - Resource constraints, latency constraints, frequency constraints, relative IO timing constraints.
  - Capable of a variety of synthesis optimizations:
    - Operation chaining, pipelining, multi-cycle communication, incremental scheduling, etc.
Scheduling Our Approach (DAC’06)

- **Overall approach**
  - **Current objective:** high-performance
  - Use a system of integer difference constraints to express all kinds of scheduling constraints
  - Represent the design objective in a linear function

  ![Diagram]

  - Platform characterization:
    - adder (+/−) 2ns
    - multiplier (*): 5ns
  - Target cycle time: 10ns
  - Resource constraint: Only ONE multiplier is available

- **Dependency constraint**
  - \( v_1 \rightarrow v_3 : x_3 - x_1 \begin{bmatrix} 1 \end{bmatrix} 0 \)
  - \( v_2 \rightarrow v_3 : x_3 - x_2 \begin{bmatrix} 1 \end{bmatrix} 0 \)
  - \( v_3 \rightarrow v_5 : x_4 - x_3 \begin{bmatrix} 1 \end{bmatrix} 0 \)
  - \( v_4 \rightarrow v_5 : x_4 - x_4 \begin{bmatrix} 1 \end{bmatrix} 0 \)

- **Frequency constraint**
  - \( <v_2, v_5> : x_5 - x_2 \begin{bmatrix} 1 \end{bmatrix} 1 \)

- **Resource constraint**
  - \( <v_2, v_3> : x_3 - x_2 \begin{bmatrix} 1 \end{bmatrix} 1 \)

- **Totally unimodular matrix:** guarantees integral solutions

\[
\begin{bmatrix}
1 & 0 & -1 & 0 & 0 \\
0 & 1 & -1 & 0 & 0 \\
0 & 0 & 1 & -1 & 0 \\
0 & 0 & 0 & 1 & -1 \\
0 & 1 & 0 & 0 & -1
\end{bmatrix}
\begin{bmatrix}
X_1 \\
X_2 \\
X_3 \\
X_4 \\
X_5
\end{bmatrix}
\begin{bmatrix}
0 \\
-1 \\
0 \\
0 \\
-1
\end{bmatrix}
\]
Another Example: Efficient Pattern Mining
[FPGA’08 and DATE’2010]

- Programmers may contain many patterns
- Prior work can only identify exact patterns
- We can efficiently identify “approximate” patterns in large programs
  - Based on the concept of editing distance
  - Use data-mining techniques
  - Efficient subgraph enumeration and pruning
- Highly scalable – can handle programs with 100,000+ lines of code
- Applications:
  - Behavioral synthesis:
    - 20+\% area reduction due to sharing of approximate patterns
  - ASIP synthesis:
    - Identify & extract customized instructions
int amplitude[N]; // Global variable
int state[N];     // Global variable
...
for (i = 0; i < N; ++i)
    if (state[i] & pos)
        d += amplitude[i];

Manual coding ......

int amplitude[N]; // Global variable
int state[N];     // Global variable
int* SPM = &amp1itude[0];

spm_pos(SPM);

spm_size(2*N*sizeof(int));
...
for (i = 0; i < N; ++i)
    if (SPM[N+i] & pos)
        d += SPM[i];

an SPM access
Compiler Support for Customizable Hybrid Cache

- **RASP (Reuse-Aware SPM Prefetching) flow**
  - Prefetch-enabled: hide memory access latency
  - Reuse-enabled: reduce amount of data transfers

**RASP**
- Hybrid Cache Configuration
- C/C++ Program
- Architecture Parameters

**LLVM-2.7 implementation**
- maximal SPM size
- prefetch latency

**Reuse Analysis**
- Reuse Candidate Graphs

**Reuse & Prefetching Co-Optimization**

**Optimized Code for Hybrid Cache**

**Flowchart**
- initiate SPM buffer size
- calculate SPM utilization ratio $r$ for inactive reuse dependency
- activate reuse dep. $u \rightarrow v$ with largest $r$
- update local/reuse regions of downstream vertices of $v$
- exceed SPM size?
  - yes
  - no
Reuse Candidate Graph

- Reuse candidate graph construction
  - Each vertex represents one array reference
  - Each edge represents reuse dependency

\[
\text{for } i = 0 \text{ to } N \\
\text{for } j = 0 \text{ to } M \\
v[i][j] = u[i+1][j+1] + DT\cdot u[i+1][j] + u[i][j+1] + u[i+1][j+2] + u[i+2][j+1],
\]

Diagram:
- Vertices represent array references.
- Edges represent reuse dependencies.
- Directions and distances between vertices correspond to the equation.

52
RASP Experimental Result

Performance and energy comparison

- Average performance gain ~ 15.9%, 12.9% and 18.5%
- Average energy gain ~ 22%, 31.2% and 10%
Application Domain: Medical Image Processing

Medical images exhibit sparsity, and can be sampled at a rate \( << \) classical Shannon - Nyquist theory:

\[
\min_u \sum_{\text{samp. points}} \| A R u - S \|^2 + \lambda \sum_{\text{vols.}} \| \text{grad}(u) \|
\]

compressive sensing

\[
\forall \text{voxel} : u(i) = \frac{1}{Z(i)} e^{-\frac{1}{2} \sum_{\text{voxel j in volume}} (w_{ij} f(j))^2} \sum_{\text{points}} \left( \sum_{s=1}^2 \sqrt{\sum_{k} |x_i^s - z_i^s|^2} \right)^{-\frac{1}{2}}
\]

total variational algorithm

\[
v = \frac{\partial u}{\partial t} + \nu \cdot \nabla u
\]

\[
\mu \Delta v + (\mu + \eta) \nabla (\nabla \cdot v) = -\nabla (T(x - u) - R(x)) \nabla T(x - u)
\]

fluid registration

\[
\frac{\partial \phi}{\partial t} = \nabla \phi \left[ F(\text{data}, \phi) + \lambda \text{div} \left( \frac{\nabla \phi}{\| \nabla \phi \|} \right) \right]
\]

surface(t) = \{voxels x : \phi(x,t) = 0\}

level set methods

\[
\frac{\partial v}{\partial t} + (v \cdot \nabla) v = -\nabla p + \nu \Delta v + f(x,t)
\]

\[
\frac{\partial v_i}{\partial t} + \sum_{j=1}^3 v_j \frac{\partial v_i}{\partial x_j} = -\frac{\partial p}{\partial x_i} + \nu \sum_{j=1}^3 v_j \frac{\partial^2 v_i}{\partial x_j^2} + f_i(x,t)
\]

Navier-Stokes equations
Experimental Platform: Based on Convey HC-1

- Will upgrade to HC-1 ex (with virtex 6) and fermi-based Tesla card
## Performance and Power Consumption

<table>
<thead>
<tr>
<th></th>
<th>CPU (2-thread)</th>
<th>GPU</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Active power</strong></td>
<td>35W (TDP)</td>
<td>200W (TDP)</td>
<td>~90W (4 FPGAs)</td>
</tr>
<tr>
<td><strong>Compressive sensing</strong></td>
<td>1809s</td>
<td>330s (5.5x)</td>
<td>Hybrid: 178s (EM on FPGA) + 57s (TV on GPU) (7.7x)</td>
</tr>
<tr>
<td></td>
<td>63315J</td>
<td>66000J (0.96x)</td>
<td>27420J (2.3x)</td>
</tr>
<tr>
<td><strong>Denoise</strong></td>
<td>0.9s per iteration</td>
<td>0.02s per iteration (45x)</td>
<td>0.09s per iteration (10x)</td>
</tr>
<tr>
<td></td>
<td>31.5J per iteration</td>
<td>4J per iteration (7.9x)</td>
<td>8.1J per iteration (3.9x)</td>
</tr>
<tr>
<td><strong>Registration</strong></td>
<td>2.8s per iteration</td>
<td>0.18s per iteration (15.6x)</td>
<td>0.15s per iteration (18.7x)</td>
</tr>
<tr>
<td></td>
<td>98J per iteration</td>
<td>36J per iteration (2.7x)</td>
<td>13.5J per iteration (7.2x)</td>
</tr>
<tr>
<td><strong>Segmentation</strong></td>
<td>1.9s per iteration</td>
<td>0.05s per iteration (38x)</td>
<td>0.28s per iteration (6.8x)</td>
</tr>
<tr>
<td></td>
<td>66.5J per iteration</td>
<td>10J per iteration (6.7x)</td>
<td>25.2J per iteration (2.6x)</td>
</tr>
</tbody>
</table>

Compressive sensing uses $128^3$ dataset; all others use $256^3$ dataset. FPGA-based compressive sensing and registration uses **fixed-point**.
Concluding Remarks

- Despite of end of scaling, there is plenty of opportunity with customization and specialization for energy efficient computing

- Many opportunities and challenges for architecture support
  - Cores
  - Accelerators
  - Memory
  - Network-on-chips

- Software support is also critical
Acknowledgements

• A highly collaborative effort
  • thanks to all my co-PIs in four universities – UCLA, Rice, Ohio-State, and UC Santa Barbara
• Thanks the support from the National Science Foundation, GSRC, Intel, and Xilinx

Aberle (UCLA)  Baraniuk (Rice)  Bui (UCLA)  Chang (UCLA)  Cheng (UCSB)  Cong (Director) (UCLA)

Palsberg (UCLA)  Potkonjak (UCLA)  Reinman (UCLA)  Sadayappan (Ohio-State)  Sarkar (Associate Dir) (Rice)  Vese (UCLA)
Backup Slides
An overview of RASP

Less than 3KB SPM, reduce ~60% data transfers!

<table>
<thead>
<tr>
<th>SPM</th>
<th>u[i+2][j+1]</th>
<th>u[i+1][j+2]</th>
<th>u[i+1][j+1]</th>
<th>u[i+1][j]</th>
<th>u[i][j+1]</th>
<th>SPM size</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPM₀</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>105</td>
</tr>
<tr>
<td>SPM₁</td>
<td>21</td>
<td>22</td>
<td>1</td>
<td>21</td>
<td>21</td>
<td>86</td>
</tr>
<tr>
<td>SPM₂</td>
<td>21</td>
<td>23</td>
<td>1</td>
<td>2</td>
<td>21</td>
<td>68</td>
</tr>
<tr>
<td>SPM₃</td>
<td>21</td>
<td>278</td>
<td>1</td>
<td>2</td>
<td>21</td>
<td>323</td>
</tr>
</tbody>
</table>

(N, M) = (256, 256)
Prefetch latency: 20

SPMₘₐₓ: 3KB
CDSC Simulation Framework

Hierarchical RF-I enabled coherence protocol

RF-I shortcut

Scratchpad Memory

Dynamic NUCA

Heterogeneity

Tightly-coupled accelerator

Loosely-coupled accelerator

SIMICS (Functional model)

OPAL (Timing model of processor)

WATTCH (Power model of processor)

GARNET (Timing model of interconnection)

Customized routing for irregular NoC

Processor Power/Area Model (Interface to McPAT)

Accelerator Power/Area Model (Interface to AutoPilot)
Automatic Memory Partitioning [ICCAD09]

- Memory system is critical for high performance and low power design
  - Memory bottleneck limits maximum parallelism
  - Memory system accounts for a significant portion of total power consumption

- Goal
  - Given platform information (memory port, power, etc.), behavioral specification, and throughput constraints
    - Partition memories automatically
    - Meet throughput constraints
    - Minimize power consumption

\[
\text{for}\ (\text{int}\ i = 0;\ i < n;\ i++) \\
... = A[i]+A[i+1]
\]
Automatic Memory Partitioning (AMP)

**Techniques**
- Capture array access confliction in conflict graph for throughput optimization
- Model the loop kernel in parametric polytopes to obtain array frequency

**Contributions**
- Automatic approach for design space exploration
- Cycle-accurate
- Handle irregular array accesses
- Light-weight profiling for power optimization

![Diagram of AMP process]

```
Loop Nest
   ↓
Array Subscripts Analysis
   ↓
Partition Candidate Generation
   ↓
Try Partition Candidate C_i,
   Minimize Accesses on Each Bank
   ↓
Meet Port Limitation
   Y: Power Optimization
   N: Throughput Optimization
   ↓
Loop Pipelining and Scheduling
   ↓
Pipeline Results
```
Automatic Memory Partitioning (AMP)

- About 6x throughput improvement on average with 45% area overhead

- In addition, power optimization can further reduced 30% of power after throughput optimization

<table>
<thead>
<tr>
<th></th>
<th>Original II</th>
<th>Partition II</th>
<th>Original SLICES</th>
<th>Partition SLICES</th>
<th>Area Comparsion</th>
<th>Power Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>fir</td>
<td>3</td>
<td>1</td>
<td>241</td>
<td>510</td>
<td>2.12</td>
<td>26.82%</td>
</tr>
<tr>
<td>idct</td>
<td>4</td>
<td>1</td>
<td>354</td>
<td>359</td>
<td>1.01</td>
<td>44.23%</td>
</tr>
<tr>
<td>litho</td>
<td>16</td>
<td>1</td>
<td>1220</td>
<td>2066</td>
<td>1.69</td>
<td>31.58%</td>
</tr>
<tr>
<td>matmul</td>
<td>4</td>
<td>1</td>
<td>211</td>
<td>406</td>
<td>1.92</td>
<td>77.64%</td>
</tr>
<tr>
<td>motionEst</td>
<td>5</td>
<td>1</td>
<td>832</td>
<td>961</td>
<td>1.16</td>
<td>10.53%</td>
</tr>
<tr>
<td>palindrome</td>
<td>2</td>
<td>1</td>
<td>84</td>
<td>65</td>
<td>0.77</td>
<td>0.00%</td>
</tr>
<tr>
<td>avg</td>
<td>5.67x</td>
<td></td>
<td></td>
<td></td>
<td>1.45</td>
<td>31.80%</td>
</tr>
</tbody>
</table>
**Experimental Results - Increasing request for accelerator**

**Effect of increasing number of threads**

- 8 cores, 1 accelerator
- Slowdown due to accelerators sharing

**Flat speedup compare to OS-based approach when increasing the number of request for accelerators**
Experimental results – Estimation Errors

* Very low estimation error for both core and GAM (maximum 6% error, average 3% for core, 2% for GAM)
* Decreasing GAM estimation error because of the canceling effect