Approximating Computer Architecture

Adrian Sampson      Cornell
Some Dubious Advice!

1. Never Listen to Advice
2. Always Listen to Advice
3. It's not naivety; it's Beginner's Mind
4. The best Computer Architecture Research always "cheats"
5. Don't Do Frameworks
6. Start a Blog
7. Talk to Your Friends
Harvey Mudd College (yesterday!)
**Dynamic Grooming Algorithm (DGA)**

**Input:** A connection request \( c \) and a network state function \( U \).

**Output:** Returns true if \( c \) was satisfied and false otherwise. If it was satisfied, \( U \) and \( S(c) \) reflect the new network state.

1. \( S(c) \leftarrow \emptyset \)
2. \( s \leftarrow (L(c), \min\{R(c), L(c) + r\}) \)
3. while \( L(s) \neq R(s) \) do
4. if \( U(s) < C \) then
   // \( s \) is available to be used by \( c \\
5. \quad U(s)++ \\
6. \quad \) let \( S(c) \) contain \( s \\
7. \quad s \leftarrow (R(s), \min\{R(c), R(s) + r\}) \\
8. \) else
   // a conflict occurs with \( s \\
9. \quad s \leftarrow (L(s), R(s) - 1) \\
10. \) end
11. end
12. return \( R(s) == R(c) \) // true iff connection is successfully satisfied
1. NEVER LISTEN TO ADVICE
2. ALWAYS LISTEN TO ADVICE
3. IT'S NOT NAÏVITÉ; IT'S BEGINNER'S MIND
4. THE BEST COMPUTER ARCHITECTURE RESEARCH ALWAYS "CHEATS"
5. DON'T DO FRAMEWORKS
6. START A BLOG
7. TALK TO YOUR FRIENDS
Code-Communication Specifications

What **code** may communicate across threads?

- enqueue ✅
- dequeue
- enqueue ✗
- render

May writes in `enqueue` be read by other threads in `dequeue`?
Is this computer architecture?
Compiler

ld 0x04 r1
ld 0x08 r2
add.a r1 r2 r3
st.a 0x0c r3

Dual-Voltage Core

$V_{DDH}$

$V_{DDL}$
Safety by isolation

[PLDI 2011]
@Approx int a = ...;

@Precise int p = ...;

p = a;

a = p;
EnerJ type qualifiers

@Approx int a = ...;

@Precise int p = ...;

p = a;

a = p;
Operator overloading for approximate computation

@Approx int a = ...;

@Precise int p = ...;

p + p;  \hspace{1cm} \text{precise addition}

p + a;  \hspace{1cm} \text{approximate addition}

a + a;
1. NEVER LISTEN TO ADVICE
2. ALWAYS LISTEN TO ADVICE
3. IT'S NOT NAIVE; IT'S BEGINNER'S MIND
4. THE BEST COMPUTER ARCHITECTURE RESEARCH ALWAYS "CHEATS"
5. DON'T DO FRAMEWORKS
6. START A BLOG
7. TALK TO YOUR FRIENDS
Abstract

Although transistor density continues to increase, voltage scaling has stalled and thus power density is increasing each technology generation. Particularly in mobile devices, which have limited cooling options, these trends lead to a utilization wall in which sustained chip performance is limited primarily by power rather than area. However, many mobile applications do not demand sustained performance; rather they comprise short bursts of computation in response to sporadic user activity.

To improve responsiveness for such applications, this paper introduces a new design technique that uses phase-changing materials (PCMs) to actively control junction temperature. This technique is especially useful for systems with high heat flux and low thermal mass, such as tablets, and smart phones, which, despite relatively large chip area, have thermal budgets that are constrained by the poor heat dissipation of passive convection.

Processors today (and their heat sinks and energy delivery systems) are designed primarily for sustained performance. Although the focus on sustained performance is the right design choice for some applications (for example, batch-mode high-performance computing), many workloads are interactive in nature and thus demand responsiveness—how long does the user have to wait after initiating a command? In such settings, responsiveness may be more important than sustained performance.
The diagram illustrates the concept of phase change materials (PCMs). On the left, a graph shows the relationship between temperature and heat. The phases are indicated as solid, liquid, and gas, with transitions labeled as melting/freezing and vaporizing/condensing. A PCM is depicted on the right, placed near a CPU, indicating its use in thermal management systems.
1. NEVER LISTEN TO ADVICE
2. ALWAYS LISTEN TO ADVICE
3. IT'S NOT NAÏVE; IT'S BEGINNER'S MIND
4. THE BEST COMPUTER ARCHITECTURE RESEARCH ALWAYS "CHEATS"
5. DON'T DO FRAMEWORKS
6. START A BLOG
7. TALK TO YOUR FRIENDS
Approximate Program

CPU

Accelerator
Approximate Program

Code with Annotations + \( d(y, y') \) quality metric

qualifiers for C & C++

interactive feedback

analysis & optimization

auto-tuning

FPGA accelerator
EnerJ type qualifiers for the Clang compiler

```c
for (int k = k1; k < k2; k++) {
    APPROX float distance =
        dist(points->p[k],
             points->p[0]);
    ...
}
```
Optimization feedback loop

**analysis library** finds coarse-grain, safe-to-approximate regions

**client optimizations** use analysis to relax approximable code

```c
for (int k = k1; k < k2; k++) {
  APPROX float distance =
    dist(points->p[k],
         points->p[0]);
  ...
}
```

$ accept log
loop at streamcluster.cpp:651
can offload to NPU
$

**reports** tell developers where annotations are preventing optimization
Neural acceleration on a commercial FPGA
Auto-tuning for optimal trade-offs
Auto-tuning for optimal trade-offs
ACCEPT speedup over precise execution

Benchmarks are 2.9× faster on average with quality loss under 10%
ACCEPT: A Programmer-Guided Compiler Framework for Practical Approximate Computing

Adrian Sampson  André Baixo  Benjamin Ransford  Thierry Moreau
Joshua Yip  Luis Ceze  Mark Oskin
University of Washington

Abstract
Approximate computing trades off accuracy for better performance and energy efficiency. It offers promising optimization solutions [Rubio-González et al. 2013]; using special low-power hardware structures that produce wrong results probabilistically [Esmailzadeh et al. 2012b; Liu et al. 2011]; and training hardware neural networks to mimic the behavior of costly...
1. NEVER LISTEN TO ADVICE
2. ALWAYS LISTEN TO ADVICE
3. IT'S NOT NAÏVE; IT'S BEGINNER'S MIND
4. THE BEST COMPUTER ARCHITECTURE RESEARCH ALWAYS "CHEATS"
5. DON'T DO FRAMEWORKS
6. START A BLOG
7. TALK TO YOUR FRIENDS
Closed Problems in Approximate Computing
OCTOBER 14, 2017

These are notes for a talk I will give at the NOPE workshop at MICRO 2017, where the title is Approximate Computing Is Dead; Long Live Approximate Computing.

Approximate computing has reached an adolescent phase as a research area. We have picked bushels of low-hanging fruit. While there are many approximation papers left to write, it’s a good time to enumerate the closed problems: research problems that are probably no longer worth pursuing.

Closed Problems in Approximate Hardware

No more approximate functional units. Especially for people who love VLSI work, a natural first step in approximate computing is to design approximate adders, multipliers, and other basic functional units. Cut a carry chain here, drop a block of intermediate results there, or use an automated search to find “unnecessary” gates—there are lots of ways to design an FU that’s mostly right most of the time. Despite dozens of papers in this area, there’s not much new going on here.

**Practice Writing**

**Clarify Your Own Thoughts**

**Save Time Explaining Stuff**

“Market” Your Work

**Get Famous**
1. NEVER LISTEN TO ADVICE
2. ALWAYS LISTEN TO ADVICE
3. IT'S NOT NAIVE; IT'S BEGINNER'S MIND
4. THE BEST COMPUTER ARCHITECTURE RESEARCH ALWAYS "CHEATS"
5. DON'T DO FRAMEWORKS
6. START A BLOG
7. TALK TO YOUR FRIENDS
ZHURU ZHANG
CORNELL
#pragma HLS ARRAY_PARTITION variable=m1 factor=3
#pragma HLS ARRAY_PARTITION variable=m2 factor=3
#pragma HLS ARRAY_PARTITION variable=prod factor=3

int m1[512][512];
int m2[512][512];
int prod[512][512];

for (int i = 0; i < 512; i++) {
    for (int j = 0; j < 512; j++) {
        int sum = 0;
        for (int k = 0; k < 512; k++) {
            #pragma HLS UNROLL factor=3
            sum += m1[i][k] * m2[k][j];
        }
        prod[i][j] = sum;
    }
}
Area–Performance Trade-Offs in Unrolling

- LUTs Used vs. Unrolling Factor
- Runtime (ms) vs. Unrolling Factor
1. NEVER LISTEN TO ADVICE
2. ALWAYS LISTEN TO ADVICE
3. IT'S NOT NAÏVITÉ; IT'S BEGINNER'S MIND
4. THE BEST COMPUTER ARCHITECTURE RESEARCH ALWAYS "CHEATS"
5. DON'T DO FRAMEWORKS
6. START A BLOG
7. TALK TO YOUR FRIENDS
The diagram illustrates the complexity of different programming paradigms. It shows a vertical axis labeled "Complexity" and horizontal axes labeled "NORMAL PROGRAMMING," "HIGH-PERFORMANCE PROGRAMMING," "PARALLEL PROGRAMMING," and "ACCELERATOR DESIGN." The complexity increases from left to right, with "ACCELERATOR DESIGN" being the most complex.

At the top right of the diagram, there is an illustration of a fist with a skull and crossbones, emphasizing the increased complexity and potential danger associated with "ACCELERATOR DESIGN."
Complexity

“Normal” Programming
High-Performance Programming
Parallel Programming
Accelerator Design

- Data Structures
- Memory Management
- Asymptotic Complexity
- ILP
- Memory Models
- Synchronization
- Data Races
- False Sharing
- ???
class AlexNet(nn.Module):

def __init__(self, num_classes: int = 1000) -> None:
    super(AlexNet, self).__init__()

    self.features = nn.Sequential(
        nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=3, stride=2),
        nn.Conv2d(64, 192, kernel_size=5, padding=2),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=3, stride=2),
        nn.Conv2d(192, 384, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.Conv2d(384, 256, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=3, stride=2),
    )

    self.avgpool = nn.AdaptiveAvgPool2d((6, 6))

    self.classifier = nn.Sequential(
        nn.Linear(256 * 6 * 6, 4096),
        nn.ReLU(inplace=True),
        nn.Dropout(),
        nn.Linear(4096, 4096),
        nn.ReLU(inplace=True),
        nn.Dropout(),
        nn.Linear(4096, num_classes),
    )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
assembly: 

:: CUDA:

Python

JAX

mxnet

TensorFlow

PyTorch
assembly: C++ R -> :: HDLs: Accelerator Design Languages
assembly: C++ R :: HDLs: HeteroCL

[Durst et al., PLDI 2020]

[Academic ADLs]

Dahlia

[Nigam et al., PLDI 2020]
Rethinking the system stack for reconfigurable hardware (FPGAs)

Vega et al., PLDI 2021

Type systems for modular reasoning

Nigam et al., PLDI 2023

debuggers

Berlstein et al., ASPLOS 2023

Profilers

Module systems

Assembly

C++, R, Python, OCaml, JavaScript

HDLs
1. NEVER LISTEN TO ADVICE
2. ALWAYS LISTEN TO ADVICE
3. IT'S NOT NAÏVETÉ; IT'S BEGINNER'S MIND
4. THE BEST COMPUTER ARCHITECTURE RESEARCH ALWAYS “CHEATS”
5. DON'T DO FRAMEWORKS
6. START A BLOG
7. TALK TO YOUR FRIENDS