A Simulator and Compiler Framework for Agile Hardware-Software Co-design Evaluation and Exploration

ICCAD 2020 Opensource Tools and Platforms for Agile Development of Specialized Architectures

Tyler Sorensen
UC Santa Cruz

Aninda Manocha, Esin Tureci, Margaret Martonosi
Princeton University

Juan L. Aragón
Universidad de Murcia
Speaker Bio

• Tyler Sorensen

• **Assistant Professor** at UC Santa Cruz
  Since Summer 2020

• Previously I was a **Post doc** at Princeton with Margaret Martonosi
  when this work was done

• Background is in Programming Languages (GPU programming models), but I was interested in
  peeking under the hood 😊

https://twitter.com/Tyler_UCSC
https://users.soe.ucsc.edu/~tsorensen/
The DECADES Project

• Part of DARPA’s SDH project

• Principle Investigators:
  • David Wentzlaff (Princeton)
  • Luca P. Carloni (Columbia)
  • Margaret Martonosi

• Developing a new tiled, heterogeneous architecture with data-supply and accelerator innovations (tape out planned in near future!)
Building a Chip is a Big Project...

The focus of Professor Martonosi’s group was:

• Programming language support and innovations
  • Support for popular programming languages
  • Modular and extensible design
• Early stage design space exploration
Our Contributions: A Compiler and Simulator

• Compiler: **DEC++**
  • Builds on top of LLVM, Clang frameworks.
  • Kernel-centric parallel programming model (main support for C++)
  • Flexible frontends, backends, and transformations

https://github.com/PrincetonUniversity/DecadesCompiler

• Simulator: **MosaicSim**
  • Early-stage performance estimates (cycle-driven LLVM IR simulation)
  • Tile model support heterogeneous core models (both CPUs and accelerators)
  • *Best Paper Nomination at ISPASS 2020!*

  **MosaicSim: A Lightweight, Modular Simulator for Heterogeneous Systems**

https://github.com/PrincetonUniversity/MosaicSim
The DEC++ Compiler

```c
int main() {
    printf("\n");
    return 0;
}
```
The DEC++ Compiler

int main() {
    printf("\n");
    return 0;
}
The DEC++ Compiler

```c
int main() {
  printf("\n");
  return 0;
}
```

Too complex to develop everything!
Instead we plug into the LLVM toolflow

We require no LLVM source code changes and simply link to public APIs or use tools directly
The DEC++ Compiler

LLVM has mature frontend support (e.g. parsing) that we plug into, e.g. Clang for C++:

```c
int main() {
    printf("\n");
    return 0;
}
```
DEC++ Front End: Programming Model

Kernel function must be identified

Kernel function has two required parameters

```
void _kernel_(float *a, float *b ... int thread_id, int num_threads) {
  for (int i = thread_id; i < SIZE; i+=num_threads) {
    a[i] += b[i];
  }
}
```

Program is written in a thread-agnostic SPMD way
DEC++ Front End: Implementation

Front end implementation must intercept kernel function call and run in SPMD parallel execution

```
1 // Setting up data
2 _kernel_(a, b, ..., DEC_TID, DEC_NUMTHREADS);
3 // Cleaning up

Implemented with Clang Visitor pass

1 #pragma omp parallel for
2 for (int i = 0; i < NUM_THREADS; i++) {
3     _kernel_(a, b, ... i, NUM_THREADS);
4 }
```
The DEC++ Compiler

```c
int main() {
    printf("\n");
    return 0;
}
```
DEC++ Transformations

- Compiler passes over the LLVM AST performing re-writes and analysis.

- Lots of opportunity for innovation

- Example: Decoupled Access Execute (DAE)
void _kernel_(float *a, float *b ... int thread_id, int num_threads) {
    for (int i = thread_id; i < SIZE; i+=num_threads) {
        a[i] += b[i];
    }
}

In pseudo LLVM-IR

void _kernel_(float *a, float *b ... int thread_id, int num_threads) {
    for (int i = thread_id; i < SIZE; i+=num_threads) {
        tmp_var0 = load(a[i]);
        tmp_var1 = load(b[i]);
        tmp_var2 = tmp_var0 + tmp_var1;
        store(a[i], tmp_var2);
    }
}
void _kernel_(float *a, float *b ... int thread_id, int num_threads) {
    for (int i = thread_id; i < SIZE; i+=num_threads) {
        tmp_var0 = load(a[i]);
        tmp_var1 = load(b[i]);
        tmp_var2 = tmp_var0 + tmp_var1;
        store(a[i], tmp_var2);
    }
}

Simple code slicing places loads on an Access core and compute on the execute core

Insert API calls for NoC communication
```c
int main() {
    printf("\n");
    return 0;
}
```
DEC++ Backend and Linking

LLVM has backends for many architectures:
- X86: ideal for developing and debugging
- RISC-V: The ISA for the DECADES architecture

But how to deal with new architecture features?
DEC++ Backend and Linking

• We require architecture features to be implemented behind an API with a software emulation implementation:

Provides:
Portable Execution
Documentation
Specification

Loads (load 32 or 64 bits from the stated address and put the data into the queue):
void dec_load32_async(uint64_t qid, uint32_t *addr)
void dec_load64_async(uint64_t qid, uint64_t *addr)

Produce and Consume from Producer to Consumer

Loads that are not asynchronous and performed on the Producer are simply loaded regularly in the Producer tile and then enqueued to the “Produce to Consumer”. They appear on the Producer tile as follows:

void dec_produce32(uint64_t qid, uint32_t data)
void dec_produce64(uint64_t qid, uint64_t data)

The mirror of this interaction is when the Execute consumes data from the queue. These instructions appear on the Execute tile as follows:

uint32_t dec stimulate32(uint64_t qid, uint32_t *data)
uint64_t dec stimulate64(uint64_t qid, uint64_t *data)
Our Contributions: A Compiler and Simulator

**Compiler: DEC++**

- Builds on top of LLVM, Clang frameworks.
- Kernel-centric parallel programming model (main support for C++)
- Flexible frontends, backends, and transformations

https://github.com/PrincetonUniversity/DecadesCompiler

**Simulator: MosaicSim**

- Early-stage performance estimates (cycle-driven LLVM IR simulation)
- Tile model support heterogeneous core models (both CPUs and accelerators)
- *Best Paper Nomination at ISPASS 2020!*
  
  MosaicSim: A Lightweight, Modular Simulator for Heterogeneous Systems

https://github.com/PrincetonUniversity/MosaicSim
How DEC++ interfaces with MosaicSim

Simulator annotates memory instructions to get a memory trace and generates a data-dependency graph

```c
int main() {
    printf("\n");
    return 0;
}
```
MosaicSim: How accurate is simulating LLVM IR?

Raw cycle counts are pretty inaccurate

but why?
instruction mapping mismatches:

1 C instruction maps to:
3 LLVM IR instructions
2 X86 instructions
4 RISC-V instructions
MosaicSim: Scaling Trends

MosaicSim accurately captures trends, which is what is important for early-stage performance modeling.
MosaicSim Extras in ISPASS paper

- modeling ASIC accelerator tiles
- how complex architecture features are efficiently modeled, e.g. RoB, LSQ
- case studies of design space exploration of applications on heterogeneous architectures

Lead author of MosaicSim is Luwa Matthews (now at Apple)
Conclusion

• We present a compiler/simulator framework for hardware-software co-design

• DEC++ is built alongside LLVM, giving it flexibility in frontends and backends.
  • Straightforward to implement innovations at the IR transformation level, e.g. DAE
  • Architecture additions are provided through APIs that support native emulation

• MosaicSim provides early-stage performance estimates. Simulating LLVM is inaccurate at the cycle level, but captures trends and characterizations

*Thanks to the DECADES team and co-authors: Aninda Manocha, Esin Tureci, Marcelo Orenes-Vera, Juan L. Aragón, Margaret Martonosi*

Software:
https://github.com/PrincetonUniversity/DecadesCompiler
https://github.com/PrincetonUniversity/MosaicSim

Tyler Sorensen
https://twitter.com/Tyler_UCSC
https://users.soe.ucsc.edu/~tsorensen/