Preparing for Extreme Heterogeneity in High Performance Computing

Jeffrey S. Vetter

With many contributions from FTG Group and Colleagues

18th Annual Workshop on Charm++ and its Applications
October 20, 2020
University of Illinois at Urbana-Champaign (Virtual)

http://ft.ornl.gov  vetter@computer.org
### Highlights

**Recent trends in computing paint an ambiguous future for architectures**
- Contemporary systems provide evidence that power constraints are driving architectures to change rapidly
- Multiple architectural dimensions are being (dramatically) redesigned: Processors, node design, memory systems, I/O

- Entering an era of Extreme Heterogeneity
- Complexity is our main challenge

<table>
<thead>
<tr>
<th>Applications and software systems are all reaching a state of crisis</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Applications will not be functionally or performance portable across architectures</td>
</tr>
<tr>
<td>• Programming and operating systems need major redesign to address these architectural changes</td>
</tr>
<tr>
<td>• Procurements, acceptance testing, and operations of today’s new platforms depend on performance prediction and benchmarking.</td>
</tr>
</tbody>
</table>

- This is a crisis!

<table>
<thead>
<tr>
<th>Programming systems must provide performance portability (beyond functional portability)!!</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Strive for ‘Write once, run anywhere’</td>
</tr>
<tr>
<td>• Descriptive models of parallelism and data movement</td>
</tr>
<tr>
<td>• Introspective runtime systems</td>
</tr>
<tr>
<td>• Layered, modular, open source approaches required</td>
</tr>
</tbody>
</table>

- Examples
  - ECP investments in LLVM
  - FORTRAN with GPU offloading
  - Programming FPGAs
  - Without Verilog
  - Memory systems are changing too
  - Language support for NVM
Time for a short poll...
Q: Think back 10 years. How many of you would have predicted that many of our top HPC systems would be heterogeneous (GPU-based) architectures?
Q: Think forward 10 years. How many of you predict that our top 100 HPC systems will have the following architectural features?

Assume general purpose multicore CPU
Q: Now, imagine you are building a new application with an expected ~3M LOC and 20 team members over the next 10 years. What on-node programming model/system do you use to future-proof your app? Assume C and C++ (?)

- Fortran XX
- Metaprogramming, DSEL, etc (e.g., AMP, Kokkos, RAJA, SYCL)
- CUDA, cu**, HIP, OpenCL
- Directives: OpenMP, OpenACC
- Python, Julia, Rust, R, Matlab, etc
- Domain Specific Language (e.g., Claw, Hallide, PySL) or Domain Specific Framework (e.g., PetSc, AMReX)
- Legion, Charm++, HPX, etc
- Some new unknown programming approach
- Some combination of the above
Motivating Trends
Business climate reflects this uncertainty, cost, complexity, consolidation.
Sixth Wave of Computing

http://www.kurzweilai.net/exponential-growth-of-computing

6th wave
Predictions for Transition Period

Optimize Software and Expose New Hierarchical Parallelism

- Redesign software to boost performance on upcoming architectures
- Exploit new levels of parallelism and efficient data movement

Architectural Specialization and Integration

- Use CMOS more effectively for specific workloads
- Integrate components to boost performance and eliminate inefficiencies
- Workload specific memory+storage system design

Emerging Technologies

- Investigate new computational paradigms
  - Quantum
  - Neuromorphic
  - Advanced Digital
  - Emerging Memory Devices
Predictions for Transition Period

Optimize Software and Expose New Hierarchical Parallelism
- Redesign software to boost performance on upcoming architectures
- Exploit new levels of parallelism and efficient data movement

Architectural Specialization and Integration
- Use CMOS more effectively for specific workloads
- Integrate components to boost performance and eliminate inefficiencies
- Workload specific memory+storage system design

Emerging Technologies
- Investigate new computational paradigms
  - Quantum
  - Neuromorphic
  - Advanced Digital
  - Emerging Memory Devices
### Predictions for Transition Period

#### Optimize Software and Expose New Hierarchical Parallelism
- Redesign software to boost performance on upcoming architectures
- Exploit new levels of parallelism and efficient data movement

#### Architectural Specialization and Integration
- Use CMOS more effectively for specific workloads
- Integrate components to boost performance and eliminate inefficiencies
- Workload specific memory+storage system design

#### Emerging Technologies
- Investigate new computational paradigms
  - Quantum
  - Neuromorphic
  - Advanced Digital
  - Emerging Memory Devices
Quantum computing: Qubit design and fabrication have made recent progress but still face challenges.
Fun Question: when was the field effect transistor patented?

Lilienfeld patents field effect transistor, October 8, 1926
Jessica MacNeil - October 08, 2018
6 Comments

On this day in tech history, JE Lilienfeld filed a patent for a three-electrode structure using copper-sulfide semiconductor material, known today as a field-effect transistor.

Lilienfeld's patent for a "method and apparatus for controlling electric currents" was granted on January 28, 1926.

According to the patent, his invention was for the control of the flow of electric current between two terminals by conducting solely by establishing a third potential that amplifies the oscillating currents like those in radio receivers.

Moral of this story
It may take decades for a new technology to be manufacturable, economical, and usable, if ever.
Predictions for Transition Period

**Optimize Software and Expose New Hierarchical Parallelism**
- Redesign software to boost performance on upcoming architectures
- Exploit new levels of parallelism and efficient data movement

**Architectural Specialization and Integration**
- Use CMOS more effectively for specific workloads
- Integrate components to boost performance and eliminate inefficiencies
- Workload specific memory+storage system design

**Emerging Technologies**
- Investigate new computational paradigms
  - Quantum
  - Neuromorphic
  - Advanced Digital
  - Emerging Memory Devices
Various Markets already Experiencing these Architectural Trends

NVIDIA Jetson AGX Xavier SoC
Experimental Computing Lab (ExC) managed by the ORNL, Future Technologies Group

- NVIDIA Jetson AGX Xavier:
- High-performance system on a chip for autonomous machines
- Heterogeneous SoC contains:
  - Eight-core 64-bit ARMv8.2 CPU cluster (Carmel)
  - 1.4 CUDA TFLOPS (FP32) GPU with additional inference optimizations (Volta)
  - 11.4 DL TOPS (INT8) Deep learning accelerator (NVDLA)
  - 1.7 CV TOPS (INT8) 7-slot VLIW dual-processor Vision accelerator (PVA)
  - A set of multimedia accelerators (stereo, LDC, optical flow)
- Provides researchers access to advanced high-performance SOC environment
## DOE HPC Roadmap to Exascale Systems

<table>
<thead>
<tr>
<th>FY 2012</th>
<th>FY 2016</th>
<th>FY 2018</th>
<th>FY 2021</th>
<th>FY 2022</th>
<th>FY 2023</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Titan</strong></td>
<td><strong>Sequoia</strong></td>
<td><strong>Summit</strong></td>
<td><strong>ORNL</strong></td>
<td><strong>ORNL</strong></td>
<td><strong>ORNL</strong></td>
</tr>
<tr>
<td>ORNL</td>
<td>ORNL</td>
<td>ORNL</td>
<td>HPE/AMD</td>
<td>HPE/AMD</td>
<td>HPE/AMD</td>
</tr>
<tr>
<td>Cray/AMD/NVIDIA</td>
<td>IBM/NVIDIA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Mira</strong></td>
<td><strong>Trinity</strong></td>
<td><strong>Theta</strong></td>
<td><strong>ANL</strong></td>
<td><strong>ANL</strong></td>
<td><strong>ANL</strong></td>
</tr>
<tr>
<td>ANL</td>
<td><strong>LANL/SNL</strong></td>
<td>LBNL</td>
<td>HPE/AMD/NVIDIA</td>
<td>HPE/AMD</td>
<td></td>
</tr>
<tr>
<td>IBM BG/Q</td>
<td>LBNL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cray/Intel Xeon/KNL</td>
<td>Cray/Intel Xeon/KNL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Cori</strong></td>
<td><strong>Sierra</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LBNL</td>
<td><strong>LLNL</strong></td>
<td></td>
<td><strong>LBNL</strong></td>
<td><strong>LANL/SNL</strong></td>
<td><strong>LLNL</strong></td>
</tr>
<tr>
<td></td>
<td>Cray/Intel Xeon/KNL</td>
<td></td>
<td>Cray/Intel Xeon/KNL</td>
<td>HPE/Intel</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>decommissioned</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Exascale Systems</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FY 2022**
- **LLNL**
  - HPE/AMD

**FY 2023**
- **LLNL**
  - HPE/AMD

---

*Version 1.6  September 30, 2020*
Future -> Open Source Hardware Enables a Rapid Design of Specialized Chips and Effectively Mass Customization

A. Olofsson, 2018
Take away message ➔
During this Sixth Wave transition, Complexity is our major challenge!

Architecture
- How do we design future systems so that they are better than current systems on important applications?
- Simulation and modeling are more difficult
- Entirely possible that the new system will be slower than the old system!
- Expect ‘disaster’ procurements

Programmability
- How do we design applications with some level of performance portability?
- Software lasts much longer than transient hardware platforms
- Proper abstractions for flexibility and efficiency
- Adapt or die

Final Report on Workshop on Extreme Heterogeneity: https://doi.org/10.2172/1473756
Implication for Applications and Software → Explosion of (incomplete) Programming Systems →

Programming Models
- OpenMP
- OpenMP Offload
- OpenACC
- SYCL
- DPC++
- HIP
- CUDA
- OpenCL
- Kokkos
- Raja
- Many others...

Implementations (SYCL Example)

Fig. 1: Current SYCL implementations and their corresponding API backends.

Inconsistent: feature support, performance, tools ecosystem
Software Strategies for Extreme Heterogeneity
Strategies for Programming Systems in this Era of Rapidly Designed, Diverse Architectures

Goals

• Strive for ‘Write one, run anywhere’

• Descriptive models of parallelism and data movement that enable effective code generation

• Introspective runtime systems

• Layered, modular, open source approaches required
  – One organization can’t do it all

Examples

• Contributing to LLVM
  – FORTRAN with GPU offloading

• Programming FPGAs
  – Without Verilog

• Memory systems are changing too
  – Language support for NVM
Contributing to LLVM
The three technical areas in ECP have the necessary components to meet national goals

**Application Development (AD)**
Develop and enhance the predictive capability of applications critical to the DOE

**Software Technology (ST)**
Produce expanded and vertically integrated software stack to achieve full potential of exascale computing

**Hardware and Integration (HI)**
Integrated delivery of ECP products on targeted systems at leading DOE computing facilities

- **Foster application development**
- **Ease of use**
- **Diverse architectures**
- **HPC leadership**

25 applications ranging from national security, to energy, earth systems, economic security, materials, and data

80+ unique software products spanning programming models and run times, math libraries, data and visualization

6 vendors supported by PathForward focused on memory, node, connectivity advancements; deployment to facilities
ECP is Improving the LLVM Compiler Ecosystem

<table>
<thead>
<tr>
<th>LLVM</th>
<th>+SOLLVE</th>
<th>+PROTEAS-TUNE</th>
<th>+FLANG</th>
<th>+HPCToolkit</th>
<th>+ATDM</th>
<th>Vendors</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Very popular open source compiler infrastructure</td>
<td>- Enhancing the implementation of OpenMP in LLVM</td>
<td>- Core optimization improvements to LLVM</td>
<td>- Developing an open-source, production Fortran frontend</td>
<td>- Improvements to OpenMP profiling interface OMPT</td>
<td>- Enhancing LLVM to optimize template expansion for FlexCSI, Kokkos, RAJA, etc.</td>
<td>- Increasing dependence on LLVM</td>
</tr>
<tr>
<td>- Easily extensible</td>
<td>- Unified memory</td>
<td>- OpenACC capability for LLVM</td>
<td>- Upstream to LLVM public release</td>
<td>- OMPT specification improvements</td>
<td></td>
<td>- Collaborations with many vendors using LLVM</td>
</tr>
<tr>
<td>- Widely used and contributed to in industry</td>
<td>- OMP Optimizations</td>
<td>- Clacc</td>
<td>- Support for OpenMP and OpenACC</td>
<td>- Refine HPCT for OMPT improvements</td>
<td></td>
<td>- AMD</td>
</tr>
<tr>
<td>- Permissive license</td>
<td>- Prototype OMP features for LLVM</td>
<td>- Flacc</td>
<td>- Recently approved by LLVM</td>
<td></td>
<td></td>
<td>- ARM</td>
</tr>
<tr>
<td>- Used for heterogeneous computing</td>
<td>- OMP test suite</td>
<td>- Autotuning for OpenACC and OpenMP in LLVM</td>
<td></td>
<td></td>
<td></td>
<td>- Cray</td>
</tr>
<tr>
<td></td>
<td>- Tracking OMP</td>
<td>- Integration with Tau performance tools</td>
<td></td>
<td></td>
<td></td>
<td>- HPE</td>
</tr>
<tr>
<td></td>
<td>implementation quality</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- IBM</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Intel</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- NVIDIA</td>
</tr>
</tbody>
</table>

Active involvement with broad LLVM community: LLVM Dev, EuroLLVM

https://github.com/llvm-doe-org/llvm-project
Leveraging LLVM Ecosystem to Meet a Critical ECP (community) need: FORTRAN

- Fortran support continues to be an ongoing requirement
- Flang project started in NNSA funding NVIDIA/PGI to open source compiler front-end into LLVM ecosystem
- SOLLVE is improving OpenMP dialect, implementation, and core optimizations
- PROTEAS-TUNE is creating OpenACC dialect and improving MLIR
- ECP projects are contributing many changes upstream to LLVM core, MLIR, etc
- Many others are contributing: backends for processors, optimizations in toolchain, ...
  - Google contributed MLIR

ECP Projects: Flang, SOLLVE, PROTEAS-TUNE
Many other contributors: NNSA, NVIDIA, ARM, Google, …
PROTEAS-TUNE: Clacc – OpenACC in Clang/LLVM

- Develop production-quality, standard-conforming traditional OpenACC compiler and runtime support by extending Clang and LLVM
  - Build on existing OpenMP infrastructure
- Enable research and development of source-level OpenACC tools
  - Design compiler to leverage Clang/LLVM ecosystem extensibility
  - E.g., Pretty printers, analyzers, lint tools, and debugger and editor extensions
- Actively contribute improvements to the OpenACC specification
- Actively contribute upstream all Clang and LLVM improvements that are mutually beneficial
  - Many contributions are already in LLVM
- Open-source with multiple collaborators (vendors, universities)

Programming FPGAs with OpenACC
Challenges in FPGA Computing

• Programmability and Portability Issues
  – Best performance for FPGAs requires writing Hardware Description Languages (HDLs) such as VHDL and Verilog; too complex and low-level
    • HDL requires substantial knowledge on hardware (digital circuits).
    • Programmers must think in terms of a state machine.
    • HDL programming is a kind of digital circuit design.
  – High-Level Synthesis (HLS) to provide better FPGA programmability
    • SRC platforms, Handel-C, Impulse C-to-FPGA compiler, Xilinx Vivado (AutoPilot), FCUDA, etc.
    • None of these use a portable, open standard.
Standard, Portable Programming Models for Heterogeneous Computing

• OpenCL
  – Open standard portable across diverse heterogeneous platforms (e.g., CPUs, GPUs, DSPs, Xeon Phis, FPGAs, etc.)
  – Much higher than HDL, but still complex for typical programmers.

• Directive-based accelerator programming models
  – OpenACC, OpenMP4, etc.
  – Provide higher abstraction than OpenCL.
  – Most of existing OpenACC/OpenMP4 compilers target only specific architectures; none supports FPGAs.
Directive-based Strategy with OpenARC: Open Accelerator Research Compiler

- Open-Sourced, High-Level Intermediate Representation (HIR)-Based, Extensible Compiler Framework.
  - Perform source-to-source translation from OpenACC C to target accelerator models.
    - Support full features of OpenACC V1.0 (+ array reductions and function calls)
    - Support both CUDA and OpenCL as target accelerator models
  - Provide common runtime APIs for various back-ends
  - Can be used as a research framework for various study on directive-based accelerator computing.
    - Built on top of Cetus compiler framework, equipped with various advanced analysis/transformation passes and built-in tuning tools.
    - OpenARC’s IR provides an AST-like syntactic view of the source program, easy to understand, access, and transform the input program.

FPGAs | Approach

• Design and implement an OpenACC-to-FPGA translation framework, which is the first work to use a standard and portable directive-based, high-level programming system for FPGAs.

• Propose FPGA-specific optimizations and novel pragma extensions to improve performance.

• Evaluate the functional and performance portability of the framework across diverse architectures (Altera FPGA, NVIDIA GPU, AMD GPU, and Intel Xeon Phi).

Baseline Translation of OpenACC-to-FPGA

- Use OpenCL as the output model and the Altera Offline Compiler (AOC) as its backend compiler.
- Translates the input OpenACC program into a host code containing HeteroIR constructs and device-specific kernel codes.
  - Use the same HeteroIR runtime system of the existing OpenCL backends, except for the device initialization.
  - Reuse most of compiler passes for kernel generation.
FPGA OpenCL Architecture

- FPGA
- External DDR Memory
- External DDR Memory
- PCIe
- External Memory Controller and PHY
- External Memory Controller and PHY
- Host Processor
- Global Memory Interconnect
- Number of Replicated Compute Units
- Kernel Pipeline
- Local Memory Interconnect
- Local Memory Interconnect
- Local Memory Interconnect
- Memory
- Memory
- Memory
- Memory
- Memory
- Memory
- Pipeline Depth
- Vector Width
Kernel-Pipelining Transformation Optimization

• Kernel execution model in OpenACC
  – Device kernels can communicate with each other only through the device global memory.
  – Synchronizations between kernels are at the granularity of a kernel execution.

• Altera OpenCL channels
  – Allows passing data between kernels and synchronizing kernels with high efficiency and low latency
Kernel-Pipelining Transformation Optimization (2)

(a) Input OpenACC code

```c
#pragma acc data copyin (a) create (b) copyout (c)
{
    #pragma acc kernels loop gang worker present (a, b)
    for(i=0; i<N; i++) { b[i] = a[i]*a[i]; }
    #pragma acc kernels loop gang worker present (b, c)
    for(i=0; i<N; i++) { c[i] = b[i]; }
}
```

(b) Altera OpenCL code with channels

```c
channel float pipe_b;
__kernel void kernel1(__global float* a) {
    int i = get_global_id(0);
    write_channel_altera(pipe_b, a[i]*a[i]);
}

__kernel void kernel2(__global float* c) {
    int i = get_global_id(0);
    c[i] = read_channel_altera(pipe_b);
}
```
Kernel-Pipelining Transformation Optimization (3)

(a) Input OpenACC code

```c
#pragma acc data copyin (a) create (b) copyout (c)
{
#pragma acc kernels loop gang worker present (a, b)
for(i=0; i<N; i++) { b[i] = a[i]*a[i]; }
#pragma acc kernels loop gang worker present (b, c)
for(i=0; i<N; i++) { c[i] = b[i]; }
}
```

(c) Modified OpenACC code for kernel-pipelining

```c
#pragma acc data copyin (a) pipe (b) copyout (c)
{
#pragma acc kernels loop gang worker pipeout (b) present (a)
for(i=0; i<N; i++) { b[i] = a[i]*a[i]; }
#pragma acc kernels loop gang worker pipein (b) present (c)
for(i=0; i<N; i++) { c[i] = b[i]; }
}
```

Kernel-pipelining transformation is valid under specific conditions.
FPGA-specific Optimizations

- Single work-item
- Collapse
- Reduction
- Sliding window
- (Branch-variant code motion)
- (Custom unrolling)
Overall Performance of OpenARC FPGA Evaluation

FPGAs prefer applications with deep execution pipelines (e.g., FFT-1D and FFT-2D), performing much higher than other accelerators.

For traditional HPC applications with abundant parallel floating-point operations, it seems to be difficult for FPGAs to beat the performance of other accelerators, even though FPGAs can be much more power-efficient.

- Tested FPGA does not contain dedicated, embedded floating-point cores, while others have fully-optimized floating-point computation units.

Current and upcoming high-end FPGAs are equipped with hardened floating-point operators, whose performance will be comparable to other accelerators, while remaining power-efficient.
Emerging Memory Systems
Memory Hierarchy is Specializing too
NVRAM Technology Continues to Improve – Driven by Broad Market Forces
Language support for NVM:
NVL-C - extending C to support NVM
Design Goals: Familiar programming interface

```c
#include <nv1.h>
struct list {
    int value;
    nv1 struct list *next;
};
void add(int k, nv1 struct list *after) {
    nv1 struct list *node
    = nv1_alloc_nv(heap, 1, struct list);
    node->value = k;
    node->next = after->next;
    after->next = node;
}
```

- Small set of C language extensions:
  - Header file
  - Type qualifiers
  - Library API
  - Pragmas

- Existing memory interfaces remain:
  - NVL-C is a superset of C
  - Unqualified types as specified by C
  - Local/global variables stored in volatile memory (DRAM or registers)
  - Use existing C standard libraries for HDD
Design Goals: Avoiding persistent data corruption

• New categories of pointer bugs:
  – Caused by multiple memory types:
    • E.g., pointer from NVM to volatile memory will become dangling pointer
  – Prevented at compile time or run time

• Automatic reference counting:
  – No need to manually free
  – Avoids leaks and dangling pointers

• Transactions:
  – Avoids persistent data corruption across software and hardware failures

• High performance:
  – Performance penalty from memory management, pointer safety, and transactions
  – Compiler-based optimizations
  – Programmer-specified hints
Design Goals: Modular implementation

- Core is common compiler middle-end
- Multiple compiler front ends for multiple high-level languages:
  - For now, just OpenARC for NVL-C
- Multiple runtime implementations:
  - For now, just Intel’s pmem (pmemobj)
NVL-C: Programming Model

• Minimal, familiar, programming interface:
  – Minimal C language extensions.
  – App can still use DRAM

• Pointer safety:
  – Persistence creates new categories of pointer bugs
  – Best to enforce pointer safety constraints at compile time rather than run time

• Transactions:
  – Prevent corruption of persistent memory in case of application or system failure

• Language extensions enable:
  – Compile-time safety constraints
  – NVM-related compiler analyses and optimizations

• LLVM-based:
  – Core of compiler can be reused for other front ends and languages
  – Can take advantage of LLVM ecosystem

```c
#include <nvl.h>
struct list {
    int value;
    nvl struct list *next;
};
void remove(int k) {
    nvl_heap_t *heap = nvl_open("foo.nvl");
    nvl struct list *a = nvl_get_root(heap, struct list);
    #pragma nvl atomic
    while (a->next != NULL) {
        if (a->next->value == k)
            a->next = a->next->next;
        else
            a = a->next;
    }
    nvl_close(heap);
}
```
#include <nvl.h>
struct list {
    int value;
    nvl struct list *next;
};

void add(int k, nvl struct list *after) {
    struct list *node = nvl_alloc_nv(heap, 1, struct list);
    node->value = k;
    node->next  = after->next;
    after->next = node;
}

• **nvl** type qualifier:
  - Indicates NVM storage
  - On target type, declares NVM pointer
  - No NVM-stored local or global variable

• Stricter type safety for NVM pointers:
  - Does not affect other C types
  - Avoids persistent data corruption
  - Facilitates compiler analysis
  - Needed for automatic reference counting
  - E.g., pointer conversions involving NVM pointers are strictly prohibited

compile-time error explicit cast won’t help
Programming Model: Bare NVM Pointers

• NVM pointers are wide:
  – Facilitates: automatic reference counting, pointer constraints, transactions
  – NVM pointers must be decoded for target architecture’s load/store
  – Bare NVM pointer = virtual address with all NVL-C metadata stripped away

• NVM pointer hoisting is important for performance:
  – Conversion to bare NVM pointer is many instructions longer than load/store
  – In tight loop, the performance penalty is severe
  – If conversion is loop-invariant, it can be hoisted
  – Currently, we implement per application with an informal NVL-C extension
  – Future work: eliminate extension and automate in compiler
Programming Model: Accessing NVM

Volatile Memory (registers, stack, bss, heap)

- `heap`:
- `nvl_heap_t`
- `root`

NVM Heap A ("A.nvl")

How do we access allocations within an NVM heap?

```cpp
nvl_heap_t *heap = nvl_open("A.nvl");
```

```cpp
nvl_T *root = nvl_get_root(heap, T);
```

Checksum error if `T` is incorrect type.

Set root with `nvl_set_root`.

Before first `nvl_set_root`, `nvl_get_root` returns null.
Programming Model: Pointer types (like Coburn et al.)

Volatile Memory (registers, stack, bss, heap)

NVM Heap A ("A.nvl")
- run-time error
- inter-heap NV-to-NV

NVM Heap B ("B.nvl")
- compile-time error
- intra-heap NV-to-NV

avoids dangling pointers when memory segments close
Programming Model: Transactions: Purpose

• Ensures data consistency

• Handles unexpected application termination:
  – Hardware failure (e.g., power loss)
  – Application or OS failure (e.g., segmentation fault)
  – NVL-C safety constraint violation (e.g., inter-heap NV-to-NV pointer)

• Does not handle concurrent access to NVM:
  – Future work
  – Concurrency is still possible
  – Programmer must safeguard NVM data from concurrent access
Programming Model: Transactions: Undo logs

```c
#include <nvl.h>
void matmul(nvl float a[I][J],
            nvl float b[I][K],
            nvl float c[K][J],
            nvl int *i)
{
    while (*i<I) {
        #pragma nvl atomic heap(heap)
        {
            for (int j=0; j<J; ++j) {
                float sum = 0.0;
                for (int k=0; k<K; ++k)
                    sum += b[*i][k] * c[k][j];
                a[*i][j] = sum;
            }
            ++*i;
        }
        ++*i;
    }
}
```

- Before every **NVM store**, transaction creates undo log to back up old data
- Undo log contains metadata plus old data being overwritten
- Problem: large overhead because an undo log is created for every element of `a` (every iteration of `j` loop)
#include <nvl.h>
void matmul(nvl float a[I][J],
            nvl float b[I][K],
            nvl float c[K][J],
            nvl int *i)
{
    while (*i<I) {
        #pragma nvl atomic heap(heap) \ 
          clobber(a[*i:1])
        {
            for (int j=0; j<J; ++j) {
                float sum = 0.0;
                for (int k=0; k<K; ++k)
                    sum += b[*i][k] * c[k][j];
                a[*i][j] = sum;
            }
            ++*i;
        }
    }
}

- **clobber** clause suppresses undo logs
- Durability after transaction commit is still guaranteed
Evaluation: LULESH

- ExM = use SSD as extended DRAM
- T1 = BSR + transactions
- T2 = T1 + backup clauses
- T3 = T1 + clobber clauses
- BlockNVM = `msync` included
- ByteNVM = `msync` suppressed

- backup is important for performance
- clobber cannot be applied because old data is needed
NVL-C Summary

• Motivated a new programming model for NVM as persistent memory
• Introduced NVL-C, a new programming system for this purpose
  – First class language construct
  – Transactions
• Described several performance optimizations for NVL-C
• Showed performance results for these optimizations on an SSD
• Working on Optane DIMMs now
Recap

Recent trends in computing paint an ambiguous future for architectures

- Contemporary systems provide evidence that power constraints are driving architectures to change rapidly
- Multiple architectural dimensions are being (dramatically) redesigned: Processors, node design, memory systems, I/O
- Entering an era of Extreme Heterogeneity
- Complexity is our main challenge

Applications and software systems are all reaching a state of crisis

- Applications will not be functionally or performance portable across architectures
- Programming and operating systems need major redesign to address these architectural changes
- Procurements, acceptance testing, and operations of today’s new platforms depend on performance prediction and benchmarking.

- This is a crisis!

Programming systems must provide performance portability (beyond functional portability)!!

- Strive for ‘Write once, run anywhere’
- Descriptive models of parallelism and data movement
- Introspective runtime systems
- Layered, modular, open source approaches required
- Examples
  - ECP investments in LLVM
  - FORTRAN with GPU offloading
  - Programming FPGAs
  - Without Verilog
  - Memory systems are changing too
  - Language support for NVM

Visit us—We host interns and other visitors year-round
- Faculty, grad, undergrad, high school, industry

Jobs at ORNL
- Postdoctoral Research Associate in Computer Science
- Software Engineer
- Computer Scientist
- Visit [https://jobs.ornl.gov](https://jobs.ornl.gov)

Contact me vetter@ornl.gov
Final Report on Workshop on Extreme Heterogeneity

1. Maintaining and improving programmer productivity
   - Flexible, expressive, programming models and languages
   - Intelligent, domain-aware compilers and tools
   - Composition of disparate software components

   • Managing resources intelligently
     - Automated methods using introspection and machine learning
     - Optimize for performance, energy efficiency, and availability

   • Modeling & predicting performance
     - Evaluate impact of potential system designs and application mappings
     - Model-automated optimization of applications

   • Enabling reproducible science despite non-determinism & asynchrony
     - Methods for validation on non-deterministic architectures
     - Detection and mitigation of pervasive faults and errors

   • Facilitating Data Management, Analytics, and Workflows
     - Mapping of science workflows to heterogeneous hardware and software services
     - Adapting workflows and services to meet facility-level objectives through learning approaches

https://orau.gov/exheterogeneity2018/

https://doi.org/10.2172/1473756