OpenMP Memkind: An Extension for Heterogeneous Physical Memories

Xi Wang, John D. Leidel, Yong Chen

August 14, 2017

Data-Intensive Scalable Computing Laboratory (DISCL)
Overview

- Background
- OpenMP Memkind
- LLVM Implementation
- Evaluation & Future Work
Heterogeneous memory systems

BACKGROUND
Heterogeneity in HPC

- LANL Roadrunner (2008)
- ORNL Titan (2011)
- SDSC Gordon (2012)
- NERSC Cori (2016-2017)
- ORNL Summit (2017)

Heterogeneous Compute

Heterogeneous Memory
Heterogeneous Memory in HPC

System Interconnects

Scratchpad Memory

Processing Targets

Memory Targets

OpenCAPI™
...yeah, its fast
...but is it very useful?
Heterogeneous Programming Models

- Heterogeneity in HPC has forced significant changes in the programming models!
  - Heterogeneous processing has become commonplace
  - OpenMP
  - OpenACC
  - OpenCL

- Our dilemma:
  - The increasingly heterogeneous memory landscape has forced us to rethink many of the recent changes to the programming models
  - Memory and processing may be hierarchical, decoupled or entirely disjoint
  - The notion of implicitly coupling the memory placement to the processing placement is now a thing of the past
    - $H$-NUMA?

Research Goal

• Develop a portable set of directives+clauses for a parallel programming model to explicitly allocate data blocks in specific physical memories

• The methodology should be flexible enough to:
  • Support heterogeneity at multiple levels
  • Support *yet-to-be-invented* devices
  • Support user-specific library extensions
  • Fail gracefully in the event of insufficient memory resources
  • Fit within the standard language parameters of the target programming model
Proposed Approach

• Our proposed approach utilizes an early OpenMP “memkind” proposal to implement arbitrarily complex memory placement
  • Extensible to many heterogeneous memory types
  • Utilizes named address spaces from C standard
  • Based upon an early proposal by Intel

• Implemented in the LLVM compiler + OpenMP runtime infrastructure

• Sample implementation for Intel KNL + MCDRAM memory
OpenMP support for heterogeneous memories

OPENMP MEMKIND
Memkind Syntax

```c
#pragma omp memkind (expr-list)

expr-list ::= var-list
- [named-address-space-modifier :][variable modifier] var-list
- [named-address-space-modifier :][variable modifier] var-list, expr-list
- named-address-space-modifier ::= fastmem - persistent - user-defined
- variable modifier ::= ref - val
```
Memkind Syntax Examples

// allocate A in fast memory with variable type
// defaults to HBM
#pragma omp memkind( fastmem : val: A )

// allocate A, B in fast memory with reference type
// defaults to MCDRAM
#pragma omp memkind( fastmem : ref: A, B )

// allocate A, B in fast memory using variable
// allocate C, D in persistent memory using reference
#pragma omp memkind( fastmem : val: A, B, persistent : ref: C, D )

// allocate E, F in user-defined address space using variable
#pragma omp memkind( user-defined: val: E, F )
// from OpenMP 4.5 Examples (4.1.1)

```c
void vec_mult( int N ) {
    int i;
    float p[N], v1[N], v2[N];
    init( v1, v2, N );

    #pragma omp memkind( fastmem:var:p,v1,v2 )
    #pragma omp target
    #pragma omp parallel for private(i)
    for(i=0; i<N; i++)
    {
        p[i] = v1[i] * v2[i];
    }

    output(p, N);
}
```

Allocate \{p,v1,v2\} in “fast” memory

Offload compute to target device

Note: we omit the “map” clause given that we have explicitly placed the memory on the device (HBM)
Memkind Syntax Examples cont.

```c
void driver(int N){
    int i;
    struct seq input[N];
    struct seq reference[BIG_N];

    #pragma omp memkind(fastmem:var:input, persistent:var:reference)

    #pragma omp parallel for private(i)
    for( i=0; i<N; i++ ){
        sequence( input[i], reference );
    }
}
```

Genetic sequencing pseudocode

Allocate \{input\} in “fast” memory

Allocate \{reference\} in persistent memory

Note that we can use memkind without target offloading
Memkind support in LLVM compiler infrastructure

LLVM IMPLEMENTATION
LLVM Implementation

- LLVM was chosen as the target implementation platform
- Why?
  - Codebase is arguably a bit more friendly to hack than GCC
  - Support for several target accelerators from multiple vendors
  - BSD-like license
  - *Much of our adjacent research utilizes LLVM*

- LLVM OpenMP implementation originally committed by Intel
- Most of the work is relegated to the Clang frontend
• Utilizes `threadprivate` as template for the memkind implementation
• Parsing and AST construction are performed alongside all other standard OpenMP directives
• We construct an adjacent backend runtime library that is utilized by the CoreGen to handle the memkind allocations
  • `kmp_memkind`
• Other runtimes can be constructed to support other memory types
  • Memkind library handles the allocations and stores a table of descriptors for the target values
  • Unified thread data management is utilized to efficiently handle memory regions in parallel
  • The actual backend `fastmem` allocation is handled via the `hbwmalloc` interface
LLVM Implementation cont.

- Top-level Tablegen is modified to include our specific directives + clauses
- Parsed tokens are semantically analyzed for correctness
- AST contains our directed nodes representing our specific directives + clauses
- Note that we do not have any LLVM IR specific passes (yet)
- CodeGen generates the calls to the OpenMP runtime and the KMP runtime.
LLVM MemKind AST
Memkind in practice

EVALUATION & FUTURE WORK
• Crafted a parallel memory allocation benchmark using OpenMP
• Creates blocked arrays commonly found in applications
  • 8 bytes, 16 bytes, 64 bytes, 1KB, 64KB
• Loops execute for 4096 iterations,
  • Allocating private “memkind” memory blocks, or “normal” memory blocks
• Memory is forcibly accessed in the inner loop in order to ensure page faults

• Each target memory size was executed 50 times
• The average time per iteration was derived for each memory size
• Test system:
  • Intel KNL 7210
  • 64 cores @ 1.3Ghz
  • 128GB of MCDRAM
  • HBW_POLICY_PREFERRED was set
    • Fall back to DDR if insufficient memory was available
    • Default OpenMP scheduling policy
Timing Results

• Given the standard OpenMP scheduling and lack of thread placement, the total runtime for each test was variable
  • No hwloc or cpusets were used
• Variable upon:
  • Kernel’s choice of starting core
  • Kernel’s internal scheduling policy
• The further removed from cache line allocations, the higher the runtime variation
Allocation Efficiency

- The memkind allocations exhibit lower runtimes for all memory sizes
- The allocation time is also proportional to the growth in the allocation size
  - 64KB allocations are ~13% more expensive than 8 byte allocations
- As the allocation size grows, the memkind implementation performance advantage also grows
  - 8.75% advantage for 64KB allocations
Future Work

• Port our work forward to the current memkind OpenMP designs

• Future experimentation with user-defined (external) address spaces, especially...
  • Memory mapped file system address spaces
  • Persistent address spaces using NVRAM, 3DXpoint
  • LLVM optimizations based upon memory locality
Future Optimizations: Memory Coalescing

- Can we inject interesting optimizations into our methodology?
- LLVM prevents us from modifying the IR, but we can add metadata
  - Can we express additional locality information such that we can coalesce similar data blocks into larger allocations?

```c
uint64_t A[LEN], B[LEN], R[LEN];
uint64_t i = 0;

#pragma omp memkind( fastmem:var: R,A,B )
#pragma omp parallel for
for( i=0; i<LEN; i++ ){
    R[i] = A[i] * B[i];
}

kmp_memkind Alloc()
```

![Diagram of memory coalescing]

- LEN
  - *R
  - *A
  - *B
Future Optimizations: Hoisting Globals

- Can we determine where globals (or statics) are allocated and hoist them out into the program constructor?
  - LLVM IR metadata has a special Global variable parameter
- This would alleviate potentially redundant calls to the runtime
- Could also improve memory allocation by reducing fragmentation

```llvm
uint64_t MyGlobal;

int main(){
    #pragma omp memkind( fastmem:var: MyGlobal )
    ..... 
}
```

LLVM IR

```
!0 = !DIGlobalVariable(name: "MyGlobal",
    linkageName: "MyGlobal",
    scope: !1, file: !2,
    line: 7, type: !3, isLocal: true, isDefinition: false, variable: i32* @foo, declaration: !4)

@MyGlobal = addrspace(5) constant i64 0,
section "fastmem", align 4
```
• We hope that our approach and associated implementation can be used as a sample guide to modifying LLVM for future LLVM research
Acknowledgements

• National Science Foundation
  • CNS-1162488
  • CNS-1338078
  • IIP-1362134
  • CCF-1409946

• Micron Technology, Inc

• Stony Brook Research Computing and Cyber Infrastructure*

• Institute for Advanced Computational Science @ Stony Brook University*

* Granted access to Llred and Seawulf computing systems for testing
Contacts

• Xi Wang: xi.wang@ttu.edu
• John Leidel: john.leidel@ttu.edu / jleidel@tactcomplabs.com
• Yong Chen: yong.chen@ttu.edu

• Data Intensive Scalable Computing Lab @ TTU
  • http://discl.cs.ttu.edu