2017 46th International Conference on Parallel Processing Workshops

OpenMP Memkind: An Extension for Heterogeneous Physical Memories

Xi Wang
Department of Computer Science
Texas Tech University
Lubbock, Texas, U.S.
xi.wang@ttu.edu

John D. Leidel
Department of Computer Science
Texas Tech University
Lubbock, Texas, U.S.
john.leidel@ttu.edu

Yong Chen
Department of Computer Science
Texas Tech University
Lubbock, Texas, U.S.
yong.chen@ttu.edu

***Abstract*—Recently, CPU and graphics processors have been increasing the degree of on-chip parallelism in order to combat the decrease in traditional Moore's Law scaling. As a result, these new processors are increasing their appetite for faster memory devices with higher bandwidth. Component manufacturers have resorted to disparate or hierarchical fast memory device architectures such as shared local memory (SLM), scratch pad memory (SPM), and high bandwidth memory (HBM) to provide sufficient bandwidth. Following this trend, the physical memory locality gradually becomes a performance feature that users would like to explicitly manage.**

Inspired by this idea, this research is conducted to create a heterogeneous memory interface based on a new declarative data storage directive, or "memkind", for the OpenMP parallel programming specification to explicitly manage physical memory locality. Our approach is implemented as an OpenMP directive in order to avoid allocating data inside parallel regions, thus avoiding performance degradation due to sequential operating system routines.

We demonstrate our approach as an extension to the LLVM OpenMP implementation, that enables the portability of our approach to be rapidly ported to any LLVM-supported architecture target. Our contributions in this work are a detailed design analysis of the memkind directive as well as a detailed implementation in the LLVM compiler infrastructure. We demonstrate the efficacy of our approach using a synthetic benchmark application that records the execution performance and memory allocation efficiency.

***Keywords*-Memkind; Heterogeneous Memory; OpenMP; LLVM**

I. Introduction

As CPU and graphics chips have grown faster in recent years, the appetite for fast delivery of information (bandwidth) has continued to increase. The latest GDDR5 memory's ability to satisfy those bandwidth and power efficiency demands is beginning to wane as the technology reaches the limits of the core specification. Given this trend, new memory solutions have been designed to meet the bandwidth requirements of discrete and integrated GPUs. These solutions include the shared local memory in GFX, scratch pad memory in GPU's and DSP's, and the high bandwidth memory devices in Intel's Knights Landing (KNL) [1]. As these disparate, fast memory types become widely used, the physical memory locality gradually becomes a performance feature that should be exposed to users.Further, the data and page placement in HPC with heterogeneous memory devices is critical to the performance of applications. Therefore, the capability to explicitly perform the data allocations in specific memory device becomes indispensable. Following this idea, new APIs, such as the high bandwidth memory allocation interface "hbwmalloc", have been developed for users to support disparate physical memory allocation.

However, in multi-threaded programs, the memory allocation within a parallel region does not function as it is lexically written. All threads participating in the parallel region are instructed to perform the memory allocation. However, given that memory allocation on most operating systems is implemented as an atomic operation, the parallel memory allocations are executed sequentially. As such, the parallel threads will execute the allocations sequentially which subsequently decreases the potential of parallel speedup. This phenomenon is further exacerbated by the existence of heterogeneous physical memory devices within a platform's memory hierarchy.

This research is defines a new heterogeneous memory interface based on a declarative data storage directive, deemed "memkind", in the OpenMP shared memory programming specification. This directive will provide users with the capability of managing heterogeneous memory allocations and memory placement. Further, this directive shall have the ancillary benefit of permitting the OpenMP runtime library to perform unified optimizations when allocating the blocks with memkind directives such that they are prescriptively reused across parallel regions.

In order to validate our memkind heterogeneous memory interface design, we first implement our memkind directive to support the specific memory operations for the Intel Knights Landing [1] HBM (high bandwidth memory) and provide extensible entries for other types of memory. The HBM was originally built to provide a higher bandwidth and better power efficiency targeting the existing GDDR5 bottlenecks. Unlike normal DDR memory, HBM stacks memory chips vertically, like floors in a skyscraper. This is similar in principle, but incompatible with the Hybrid Memory Cube interface developed by Micron [2]. The towers connect to the CPU or GPU through an ultra-fast interconnect called the "interposer". Several stacks of HBM are plugged into

1530-2016/17 $31.00 © 2017 IEEE
DOI 10.1109/ICPPW.2017.40

the interposer alongside a CPU or GPU, and that assembled module connects to a circuit board [3].

We also design and implement the memkind and associated runtime with the portability for users to run the memkind applications anywhere. If an application compiled with memkind directives is executed on a platform without hardware support for the targeted physical memory devices, the memkind directive will subsequently fall back to using the default memory allocation schema. Our design and implementation are based upon the LLVM compiler infrastructure and its associated OpenMP runtime implementation. LLVM is an open-source compilation suite written with the goal of providing a modern static-single-assignment (SSA)-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages [4].

The contribution of this research study is three-fold:

- We design a directive-based heterogeneous memory interface, deemed *memkind*, based upon the OpenMP shared memory programming specification. We provide sample implementations of which for Intel's HBM and other disparate physical memory devices;
- We implement our memkind directives into an upstream version of the LLVM compiler suite and its associated OpenMP runtime library. In addition to the core HBM support, our open source implementation provides all the infrastructure necessary for other researchers to experiment with heterogeneous memory directives [5].
- We construct a synthetic benchmark that compares conventional OpenMP memory allocations with our memkind features. We evaluate our approach using extensive tests with our allocation benchmark.

The remainder of this work is organized as follows. Section II introduces the memkind directive design. A detailed design and constructions of the workflow and memkind runtime library are demonstrated in Section III and Section IV. Evaluations and results are discussed in the Section V. Section VI discusses relevant work and we conclude this research and future work in Section VII.

II. OpenMP Directive Design

The memkind directive design is inspired by the first proposal targeting at memory features for HBM from the OpenMP Architecture Review Board (ARB) (language and accelerator subcommittee) as proposed by Intel [6]. We extend this proposal and design the memkind directive as one potential solution to provide the heterogeneous memory allocation interface support in OpenMP. Given the volatility of the current memkind proposals in the OpenMP ARB committee discussions, we expect that future minor modifications to our *syntax* will be required in order to adhere to the final specification release, though the rationale, semantics, and methodology will remain the same. We make every attempt to match the syntactical format of existing OpenMP directives in order to make inclusion into the future specification and existing runtime implementations. The syntax format of the our introduced memkind directive is designed as follows:

#pragma omp memkind (expr-list) new-line

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

In the aforementioned syntactical description, the *expr-list* consists of named-address-space-modifier, variable modifier, and var-list. Each named-address-space- modifier stands for one type of physical memory device. In this research, three named-address-space-modifiers are introduced and supported in the memkind directive, including *fastmem*, *persistent* or non-volatile storage, and *user-defined*. These modifiers specify the memory space and memory type that users intend to access. The data in *fastmem* clause currently defaults to the high bandwidth memory.

It should be noted that we use the abstract identifiers such as *fastmem*, which is designed to permit the OpenMP runtime implementors to define platform-specific defaults for the physical locality of "fast" memory. This flexible paradigm is consistent with the remainder of the OpenMP specification.

The data in *fastmem* clause currently defaults to the high bandwidth memory and *persistent* type memory refers to a nonvolatile and low-latency memory that could maintain the states on power failure.

Our current memkind implementation has full support for the *fastmem* directive as well as parsing support for the remainder of the memory types. In addition to the memory space names, variable modifiers in the directive include the *ref* and *val*, which are used to distinguish different types of variables. *Ref* is utilized to denote reference type variables and *val* denotes all remaining variable types, including pointers in a specified named-address-space-modifier.

The *expr-list* describes the expression list that may contain a simple variable list or hybrid types that include different memory kind variables. In our respective implementation, the memkind variable list defaults to the high bandwidth memory physical memory placement and associated address space. However, future memkind implementors have the

ability to extend or modify the default allocation mechanisms.

A simple example of the memkind syntax and the associated directive in a user application would resemble the following:

#pragma omp memkind(fastmem : val: a , b, persistent : ref: c, d)

In this directive, variables *a* and *b* will be allocated in the *fastmem* space, which corresponds to the high bandwidth memory device. In addition, reference type variables *c* and *d* are allocated as a persistent type which allocates a file in the host file system and converts all load and store as the read and write operations to that file. All the named-address-space-modifiers, variable modifiers and expr-list are separated by colons. All the variables in the varlist are separated by commas. This memkind directive format provides an extensible path for users to extend new memory types or user-defined memory space in the future. Rather than creating a new directive for the additional memory space, we could simply add a new named-address-space-modifier to specify a special memory space for memory operations. In this manner, future developers and researchers may quickly extend our implementation to include disparate memory types.

III. Memkind Design in LLVM

In order to provide users with the capability to compile and run the applications that take advantage of the memkind declarative data storage directive, we implement our approach into an upstream version of the Clang and LLVM toolchain [7]. We chose an upstream version of the Clang LLVM infrastructure in order to utilize the latest OpenMP specification support that has not yet been merged into the LLVM trunk. These respective patches are directed at supporting the OpenMP 4.5 specification's changes to the accelerator directives utilized to support heterogeneous computing platforms [8]. As a result, our implementation will maintain continuity with the forthcoming LLVM release versions.

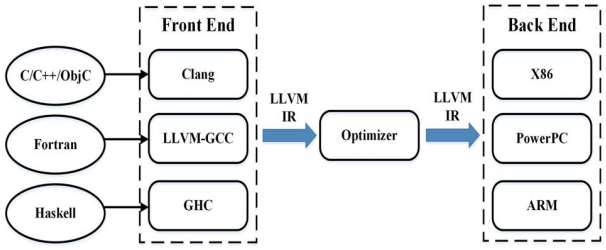

Figure 1. LLVM Architecture.

LLVM is structured as three main modules shown in Figure 1: the front-end, the optimizer and the back-end.

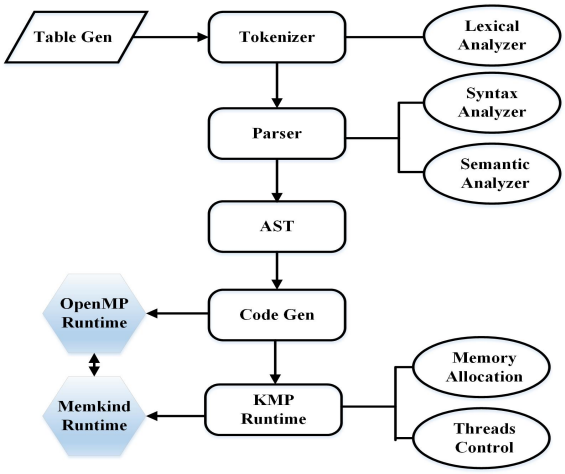

Figure 2. LLVM Deisgn Workflow.

The front-end is responsible for parsing the source code, verifying its correctness and building the intitial intermediate representation (IR One of the most common front-ends for LLVM is Clang, which supports compilation of C/C++ and Objective-C [9].

Given a bundle of IR code, the optimizer is then responsible for target-agnostic optimizations and safety analysis of the target code. The resulting IR from the optimization phase is eventually delivered to one or more target-specific back-ends. These CPU-specific backends perform *lowering* of the IR to target-specific assembly code [10]. Much of our implementation is centered upon the Clang front-end and LLVM OpenMP runtime library.

In order to support the memkind features of OpenMP in the LLVM infrastructure, we modified the following components as demonstrated in Figure 2. There are specific libraries serving OpenMP features in the components of LLVM like parser, abstract syntax tree (AST), code generator and the OpenMP (kmp) runtime. As such, the major work in this section centered upon building memkind support in each of the sub-components and adding the corresponding features into the relative OpenMP libraries of LLVM. Due to the similarity of the syntax and functions between threadprivate and memkind, in this research, we imitate and adopt some features of the *threadprivate* directive.

A. Tokenizer and TableGen

TableGen is a very useful tool that plays a pivotal role in generating the necessary C++ code to drive common tokenizing tasks in the LLVM front-end. TableGen's purpose is to help a human develop and maintain records of domain-specific information. This reduces the amount of duplication in the description, lowers the chance of error, and makes it easier to structure domain specific information [11]. Defining the instruction declarations in the OpenMP TableGen file will automatically generate a new tokenizer

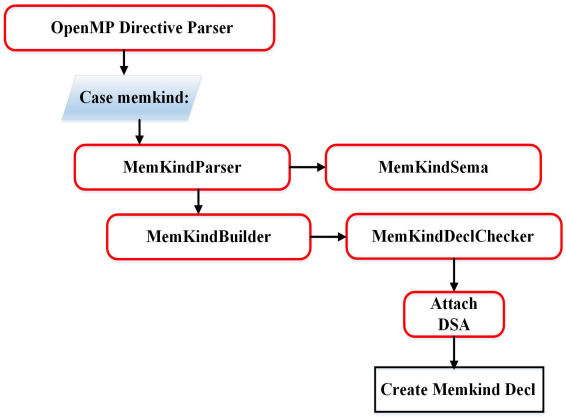

Figure 3. Memkind Parser Workflow.

that will interpret our new directives. In this manner, we take advantage of the TableGen to define the memkind as one declarative node.

We also create a class structure for our memkind directive that provides the ability to create memkind declaratives and the associated variable lists. This functionality is also utilized when parsing OpenMP instructions and conducting semantic analysis. We create the corresponding Data Sharing Attributes (DSA) for each memory types in the memkind directive. The memkind DSA is used as default, whereby fastmem and persistent are used by respective named-address-modifiers.

B. Parser and AST

The parser is designed to parse all the tokens produced by the tokenizer and conduct semantic analysis. This includes checking for incompatible types and detecting undefined references. The AST is not only built for storing the declaratives and statements, but also maintains an identifier table and source manager that tracks and navigates the source contexts through core classes such as *Decl*, *DeclContext*, *Stmt*, and *Type*. Figure 3 briefly describes the workflow that represents the proposed approach for the memkind directive design in the parser.

We first build a memkind parsing entry called *MemKindParser* in the OpenMP directive parser, to parse the memkind directive as well as the modifiers and variable list. The function ParseMemKind consumes all the completion tokens, and checks if the token matches the named-address-space-modifier or variable modifier directive specification. We subsequently build a vector structure called "Varlist" to store all the qualified variables from the variable list in the memkind directive after the syntax and semantic analysis. We also create two flags, *nas(named-address-space)_modifier_flag* and *modifier_flag*, to help recognize the
named-address-space-modifier and variable modifier for the corresponding variable.

Named-address-space-modifier and variable modifier are stored to help build and trace the data sharing attributes and variable types in the memkind directive. Syntax checks provided in *ParseMemKind* includes detecting unmatched indents, parentheses, commas and colons. Additionally, the semantic analysis is conducted in a memkind semantic analysis tool named *MemKindSema*. This performs semantic analysis that ensures the memkind directive lexically precedes all references to any of the variables in its list, and variables must be file-scope, namespace-scope, or static block-scope. The definition of the variables is supposed to appear in the same scope as the memkind directive does and so forth.

Next, a memkind handler named *MemKindBuilder* is designed for further semantic analysis and building the object of memkind declaratives, which stores all the information about the respective memkind directive. In addition, a delcarative checker, called *MemKindDeclChecker*, will be executed before the generation of the memkind directive objects. It is utilized to detect the variables that have incomplete types, thread local storage (TLS) types, qualified type (QualType) and syntax errors. Further, it appends the Data Sharing Attribute (DSA) to the DSA stack, which is used to manage all the data sharing attributes for respective variables, according to the named-address-space-modifier flag generated in function *ParseMemKind*. Finally, *MemKindBuilder*will build valid memkind declaratives and add the corresponding declarative objects into the current context.

Eventually, a memkind vistor is designed and attached to the AST libraries to help AST traverse the the memkind declaratives and data recursively. Further, we create the memkind AST reader and writer to trace the memkind declaratives and build statements for the AST, which also provides the users with the diagnosis information when encountering semantic issues.

C. Code Generator

After the memkind directives are successfully parsed, we move to the stage of code generation and the OpenMP runtime. The LLVM code generator is firmly attached to the LLVM runtime libraries in order to provide the runtime features for applications, especially those based upon language extensions. The LLVM OpenMP runtime is the fundamental runtime library designed to support the disparate directives and multithreading features found in OpenMP. This includes features such as initializing threads, variable allocations, thread synchronization and heterogeneous compute regions.

On the first stage of our design, we create a memkind driver in the OpenMP code generator that initializes the memkind directive. It is also responsible for cache memory allocations and global address generation for the constructors and destructors for all the variables in the memkind variable list. Second, a memkind initializer is built in the

runtime library of the code generator to detect and fetch each memkind variable. Then we set the global or static address and corresponding size of the variables as well as the global OpenMP thread number as arguments and emit the memkind driver. Additionally, several memkind handlers are constructed to handle memkind variables and links with the runtime library when encountering the memkind directives. In conjunction with the code generator, we design a new a back-end runtime library for the memkind, called *kmp_memkind*. This runtime library is built specifically for the memkind runtime support and attach it as a part of the kmp runtime library in the LLVM.

IV. Memkind Runtime Library

In the memkind runtime library, we first design a memkind allocator, that allocates the private storage for the memkind data with all the arguments driven by the code generator including, source location, global thread number, size and the address of the data. Alongside each allocation of memkind data, a data descriptor structure is designed, that contains the address, the size of the data, constructors and destructors. The descriptor contents are used for tracking and cleaning during the program runtime.

In addition to the core descriptors, we also build a shared table, called textitmemkind table, to store all the descriptors. In order to make sure the unique index of each descriptors, we hash the data address as the index of in the memkind table. After we construct the thread handlers, we create memkind destructors to destroy the data in memkind directives when the thread terminates. The memkind destructors traces the descriptors stored in the memkind table and destroys the allocated data buffers.

In our memkind runtime library, a unified thread data manager is designed to provide identical thread data management for all clauses in the memkind directive. Regardless of the target memory device, the same thread and data allocation optimizations are conducted. All the data in the memkind directive is maintained across different parallel regions by each thread. This implies that the allocated data in the memkind directive could be reused by parallel regions and thus, the allocation cost is amortized across the scope of the application. We also optimize the data allocation with jemalloc to provide more efficient data allocations and management.

In this research, we implement the fastmem clause as an example implementation of our memkind design. We utilize the hbwmalloc interface to allocate the data into the high bandwidth memory as the default fastmem target. The hwbmalloc API provides the high bandwidth memory interface and is generally considered to be a standard and stable interface [12]. Further, hbwmalloc provides various page size options from 4KB to 1GB in order to optimize for larger allocations in both DDR and HBM memory spaces. By default, we set the the allocation policy for the data in the fastmem clause to HBW_POLIC_PREFERRED. In this case, if insufficient memory is available from the high bandwidth NUMA node at allocation time, the memkind allocator will automatically fall back to standard memory with the smallest NUMA distance.

V. Evaluation

A. Benchmark

In order to validate the design and evaluate the efficiency of our memkind methodology, we created a memkind benchmark and compared the memkind allocations against conventional OpenMP allocations. The benchmark creates arrays with various sizes, including 8 bytes, 16 bytes, 64 bytes, 1KB, 64KB. The selected sizes match three different object sizes defined in jemalloc: small, large and huge. We also build a memkind driver with a loop that iterates 4096 times to allocate private data with a *#pragma omp memkind* directive for each thread and force threads to access their local array in each iteration. Similarly, another base driver is built to perform an identical operation utilizing only the *#pragma omp parallel* directive. Finally, we executed each driver 50 times and calculated the average time cost for a single iteration in both memkind and base driver.

B. Environment

We performed the tests on a 64-core Intel KNL Xeon Phi 7210 with a clock frequency of 1.30 GHz server and 128 GB of MCDRAM (HBM) memory.

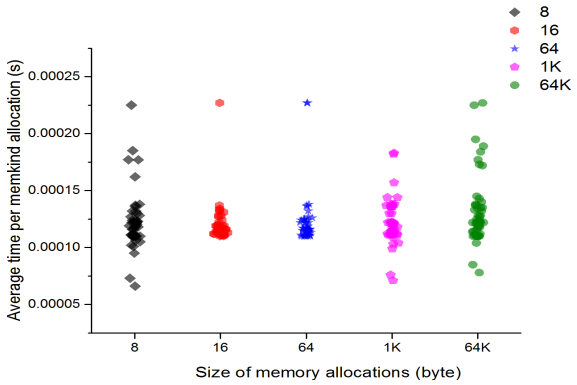

Figure 4. Time cost distribution of small and large memory allocations

C. Results and Analysis

we tested our benchmark on the standard memory by calling hbw_set_policy() to force the the allocation policy to be HBW_POLICY_ PREFERRED. In this case, if there is no sufficient memory available from the high bandwidth NUMA node closest at allocation time, hbw_malloc() automatically falls back to the standard DDR memory. We measured the total time cost for the base and memkind tests as well as derived the average timing for each allocation. As

mentioned in Section V-A, we executed the benchmark 50 times and measured the average time cost for each memory allocation. We forced the default scheduling policy of each respective OpenMP parallel region in the benchmark. As such, the time cost for each test may be quite variable given the kernel's choice of starting core and NUMA policy. Given this, we illustrate the time cost distributions for each respective memkind allocation size in Figure 4.

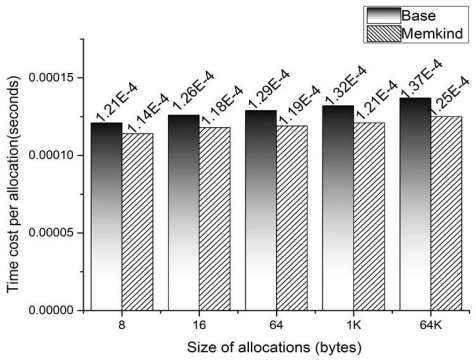

Figure 5. Time cost for small and large memory allocations.

As shown in the Figure 5,the memkind allocations are generally more efficient in comparison with conventional memory allocations for each OpenMP thread, which matches the higher bandwidth advantage of HBM compared with the conventional JEDEC DDR devices.

Further, it can be observed from both Figure 5 and Figure 4 that the average time cost is proportional to the growth of the memory allocation size. As the size of the respective memory allocation grows from 8 bytes to 65536 bytes, the time increased from 1.21E-4s to 1.37E-4s and 1.14E-4 to 1.25E-4 for base test and memkind test, respectively. A 64 KB data allocation consumed 13.223% more on each allocation in comparison with the time to allocate the 8-byte data.

Further, we also measured the improvements for allocations with various sizes when using memkind. As illustrated in the Figure 6, overall the small memkind allocations achieves impressive improvements. As the memory allocation size increases, the allocation improvement shows an ascending trend and peaks at 8.75 in 64 KB allocations%, which indicates the superiority of the data allocation in high bandwidth memory.

The evaluation and test results from the memkind benchmark validate our solution for heterogeneous memory allocations, as well as verify that our hypothesis that jemalloc promotes more efficient memory allocation for the kmp runtime library.

VI. Related Work

Computer architects have recently shifted research to heterogeneous memory architectures (HMA's) in order to

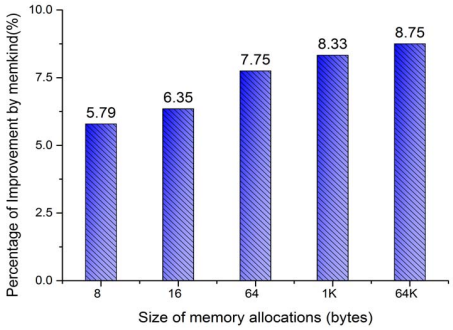

Figure 6. Improvement for each allocation by memkind.

improve memory bandwidth, latency and power consumption. Recently, a heterogeneous main memory has been proposed for future multi-core systems consisting of three memory modules [13]. Each module is optimized for one of the three aforementioned guiding principles. An offline algorithm is also designed that classifies application workloads into one of the three parameters by profiling the L2 miss rate and memory level parallelism characteristics. The operating system then allocates pages of an application in the appropriate memory module at runtime [14]. Further, recent research has also proposed similar approaches that dynamically tune memory allocation layout at runtime [15].

Given the trend of heterogeneous architecture design, the OpenMP specification [16] [17] has gradually extended support for heterogeneous systems such as GPUs, Intel Xeon Phi and FPGAs [18]. For example, a stream programming extension to OpenMP is built to help programmers express concurrency and data locality avoiding non-portable low-level code and early optimizations [19].

The OpenMP 4.1 specification includes offloading constructs that permit execution of user selected regions on generic devices, external to the host processor. This support has since been adopted to integrate GPU support for OpenMP offloading directives in compiler frontends and runtime implementations such as Clang and LLVM [20]. Further, as the latest advances in memory technology strive to increase bandwidth, recent OpenMP compiler efforts have begun to address the heterogeneity created therein. One such effort addressed the efficient use of distributed scratch-pad memories in embedded multiprocessing platforms. This programming framework combines OpenMP with simple language extensions to trigger array data partitioning by exploiting profiling data [21].

In addition to the work associated with OpenMP, a great deal of research has been conducted based on OpenCL for heterogeneous systems. FluidiCL is an OpenCL runtime implementation that takes a program written for a single device and uses both the CPU and the GPU to execute it

[22]. In order to reduce the complexity of GPU programming and further optimize the OpenCL code in HPC, the portable compiler based approach is designed to automatically generate optimized OpenCL code from data-parallel OpenMP programs for GPUs [23].

HPC systems are increasingly integrating computational accelerators such as many-core NVIDIA GPUs, Intel Xeon Phi and FPGAs [24][25]. Following this trend, OpenACC was introduced as a programming model to simplify parallel programming for heterogeneous CPU/ GPU systems. Similar to OpenMP, OpenACC utilizes directives as the primary mechanism to accelerate code on both the CPU and GPU [26]. Several compiler infrastructures have been developed to support OpenACC compilation in C, C++ and Fortran[27][28] [29].

Given its wide adoption as a commercial and open source compiler infrastructure, LLVM also attracts noticeable research to optimize performance for both CPU and GPU systems. As an example, new type-safe interface was created to construct backends for the *accelerate* language, design to target both multi-core CPUs and GPUs [30]. Further, performance evaluations of heterogeneous CPU/GPU systems have also become popular, which requires a system-wide view that considers both CPU and GPU components. The system-wide, sampling-based performance analysis capabilities provided by HPCToolKit utilizes CPU-GPU blame shifting in order to accurately describe the real bottlenecks in heterogeneous, GPU-accelerated systems [31][32].

VII. Conclusion and future work

In this work, we have presented a methodology and implementation that supports heterogeneous memory systems using OpenMP directives. We construct our implementation in the LLVM compiler infrastructure, in order to provide users the ability to perform memory operations in specific memory spaces and subsequently boost the overall efficacy of the memory allocation through the design of the memkind library and OpenMP directives.

The memkind directive supported in the OpenMP syntactical structure provides a simple, yet powerful and extensible interface to support mixed physical memory types. We depict a detailed design to create the memkind features in each stage of the LLVM. A memkind benchmark is also created and used to elicit the feasibility and efficacy of our solution with respect to the memory allocation. We demonstrate our approach using various test results from our work including, the average cost distribution for each allocation, comparisons with conventional OpenMP memory allocations and efficiency improvements for disparate memory allocation sizes.

The future direction of this research will focus on the supporting the forthcoming changes to the OpenMP memkind directives in LLVM. Further, the current mem-kind feature will be expanded to support more memory types such as persistent, volatile, teamlocal and user-defined memory types. We will also actively improve our current development based on the new features from future memkind proposals to help contribute to the ongoing LLVM development.

VIII. Acknowledgment

We are thankful to the anonymous reviewers for their valuable feedback. This research is supported in part by the National Science Foundation under grant CNS-1162488, CNS-1338078, IIP-1362134, and CCF-1409946. Further, we sincerely acknowledge the resources and support from Micron Technology, Inc. The authors would also like to thank Stony Brook Research Computing and Cyberinfrastructure, and the Institute for Advanced Computational Science at Stony Brook University for access to the high-performance LIred and SeaWulf computing systems, the latter of which was made possible by a $1.4M National Science Foundation grant (#1531492).

References

[1] Intel Powers the World's Fastest Supercomputer, Reveals New and Future High Performance Computing Technologies. https://newsroom.intel.com/news-releases/intel-powers-the-worlds-fastest-supercomputer-reveals-new-and-future-high-performance-computing-technologies/, August 2011. Accessed: 2016-08-11.

[2] Hybrid Memory Cube Specification 2.0. Technical report, Micron Technology, Inc., July 2015.

[3] High Bandwidth Memory, Reinventing Memory Technology. Technical report, AMD, January 2015.

[4] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[5] Xi Wang, John D. Leidel, and Yong Chen. Memkind: a directive based heterogeneous memory interface in OpenMP. $http : //discl.cs.ttu.edu/gitlab/xiwang/memkind_llvm.git$, 2016.

[6] OpenMP Architecture Review Board. http://openmp.org/wp/. Accessed: 2016-08-09.

[7] Code repository of the upstream version LLVM. https://github.com/clang-ykt/. Accessed: 2016-05-28.

[8] OpenMP Application Programming Interface. Technical report, OpenMP Architecture Review Board, November 2015.

[9] Chris Lattner. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008.

[10] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM: software protection for the masses. In *Proceedings of the 1st International Workshop on Software Protection*, pages 3–9. IEEE Press, 2015.

[11] LLVM TableGen. http://llvm.org/docs/TableGen/. Accessed: 2016-08-03.

[12] Documentation for high bandwidth memory interface hbwmalloc. http://memkind.github.io/memkind/man_pages-/hbwmalloc.html. Accessed: 2016-08-09.

[13] Manish Gupta, David Roberts, Mitesh Meswani, Vilas Sridharan, Dean Tullsen, and Rajesh Gupta. Reliability and Performance Trade-off Study of Heterogeneous Memories. In *Proceedings of the Second International Symposium on Memory Systems*, pages 395–401. ACM, 2016.

[14] Sujay Phadke and Satish Narayanasamy. Mlp aware heterogeneous memory system. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE, 2011.

[15] Mitesh R Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H Loh. Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 126–136. IEEE, 2015.

[16] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.

[17] Howard W Wong. *The OpenMP Optimization Framework: Using OpenMP to Optimize Structure Grid HPC Applications*. University of California, Irvine, 2013.

[18] Suyang Zhu, Sunita Chandrasekaran, Peng Sun, Barbara Chapman, Marcus Winter, and Tobias Schuele. Exploring Task Parallelism for Heterogeneous Systems Using Multicore Task Management API.

[19] Antoniu Pop and Albert Cohen. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):53, 2013.

[20] Carlo Bertolli, Samuel F Antao, Gheorghe-Teodor Bercea, Arpith C Jacob, Alexandre E Eichenberger, Tong Chen, Zehra Sura, Hyojin Sung, Georgios Rokos, David Appelhans, et al. Integrating GPU support for OpenMP offloading directives into Clang. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, page 5. ACM, 2015.

[21] Andrea Marongiu and Luca Benini. An OpenMP compiler for efficient use of distributed scratchpad memory in MPSoCs. *IEEE Transactions on Computers*, 61(2):222–236, 2012.

[22] Prasanna Pandit and R Govindarajan. Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 273. ACM, 2014.

[23] Michael FP O'Boyle, Zheng Wang, and Dominik Grewe. Portable mapping of data parallel programs to OpenCL for heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–10. IEEE Computer Society, 2013.

[24] Yonghong Yan, Pei-Hung Lin, Chunhua Liao, Bronis R de Supinski, and Daniel J Quinlan. Supporting multiple accelerators in high-level programming models. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 170–180. ACM, 2015.

[25] Jiri Kraus, Michael Schlottke, Andrew Adinetz, and Dirk Pleiter. Accelerating a C++ CFD code with OpenACC. In *Proceedings of the First Workshop on Accelerator Programming using Directives*, pages 47–54. IEEE Press, 2014.

[26] Nvidia, Cray, PGI, and CAPS launch 'OpenACC' programming standard for parallel computing. http://www.theinquirer.net/inquirer/news/2124-878/nvidia-cray-pgi-caps-launch-openacc-programming-standard-parallel-computing, November 2011. Accessed: 2016-08-15.

[27] Xiaonan Tian, Rengan Xu, Yonghong Yan, Zhifeng Yun, Sunita Chandrasekaran, and Barbara Chapman. Compiling a high-level directive-based programming model for GPGPUs. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 105–120. Springer, 2013.

[28] Xiaonan Tian, Rengan Xu, and B Chapman. OpenUH: Open Source OpenACC Compiler. 2014.

[29] OpenARC: Open Accelerator Research Compiler. Technical report, Oak Ridge National Laboratory, Future Technologies Group, 2016.

[30] Trevor L McDonell, Manuel MT Chakravarty, Vinod Grover, and Ryan R Newton. Type-safe runtime code generation: accelerate to LLVM. *ACM SIGPLAN Notices*, 50(12):201–212, 2016.

[31] Milind Chabbi, Karthik Murthy, Michael Fagan, and John Mellor-Crummey. Effective sampling-driven performance tools for GPU-accelerated supercomputers. In *2013 SC-International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12. IEEE, 2013.

[32] Milind Chabbi, Karthik Murthy, Mike Fagan, and John Mellor-Crummey. Hpctoolkit: a tool for performance analysis on heterogeneous supercomputers, 2013.