Towards Automatic HBM Allocation using LLVM: A Case Study with Knights Landing

Dounia Khaldi
Institute for Advanced Computational Science
Stony Brook University
Stony Brook, NY
Email: dounia.khaldi@stonybrook.edu

Barbara Chapman
Institute for Advanced Computational Science
Stony Brook University
Stony Brook, NY
Email: barbara.chapman@stonybrook.edu

***Abstract*—In this paper, we introduce a new LLVM analysis, called Bandwidth-Critical Data Analysis (BCDA), to decide when it is beneficial to allocate data in High-Bandwidth Memory (HBM) and then transform allocation calls into specific HBM allocation calls, for increased performance in parallel systems. High-Bandwidth Memory (HBM) is a new memory technology that features stacked 3D chips on processor dies.**

The well-known SSA-based compilation infrastructure for sequential and parallel languages LLVM will be used to detect frequently used data and patterns of memory accesses in order to decide on which level to allocate the data: HBM or DDR. BCDA core analysis counts the number of data uses and detects irregular and simultaneous accesses, generating a priority value for every variable. Using this priority value information, LLVM will generate `memkind_alloc` function calls, to transform `mallocs` to HBM allocations if HBM is present and a sufficient size of HBM is available.

As a use case for validating our approach, we show how the Conjugate Gradient (CG) benchmark from the NAS Parallel suite can be optimized for the use of MCDRAM, as the HBM on the Knights Landing Xeon Phi processors is called. We implement BCDA in the LLVM compiler and apply it on CG to detect when it is beneficial to allocate data in the HBM. Then, we allocate the data in the MCDRAM using *hbwmalloc* library calls. Using the priority generated by BCDA, we achieved a 2.29x performance improvement using the LLVM compiler and 2.33x using Intel's compiler compared to the DDR version of CG.

***Index Terms*—LLVM; HBW; KNL; MCDRAM; OpenMP;**

I. Introduction

High-Bandwidth Memory (HBM) systems, such as MCDRAM in KNL and HBM2 in the NVIDIA Pascal GPU, use a new 3D memory technology that vertically stacks memory chips on processor dies. This new technique enables the stacking-up of up to eight memory dies together vertically and makes them behave as a single device; this type of memory achieves higher performance, in particular bandwidth, at reduced power than traditional 2-dimensional DRAM. If DDR still provides lower latency compared to HBM, its limited bandwidth is a significant bottleneck when looking at performance and power consumption issues. Thus, the efficient use of HBM constitutes a vital step to exploiting modern memory systems, explaining why HBM is appearing everywhere on emergent and planned future architectures.

However, HBM systems suffer from a limited capacity. The user needs to select which parts of his data have to reside on the HBM, using vendor-provided libraries and directives to manage HBM data placement and movement. However, this data shuffling is a burden because this scheme relies on the programmer to provide high-level hints on which level of memory a particular data section should be allocated. Also, adding these low-level constructs at the user-code level decreases performance portability among different systems. Indeed, NVIDIA, AMD and Intel, for instance, have their own respective API to manage their HBM. Several efforts are being developed to add extensions to better support HBM at the API level. To the best of our knowledge, this paper is the first effort to look at the problem from a compiler point of view.

LLVM is a widespread SSA-based compilation infrastructure for sequential and parallel languages. In this paper, we use LLVM to decide when it is beneficial to allocate data in the HBM for sequential and OpenMP [1] codes. As a case study, we will use the HBM of Knights Landing (KNL), the second generation of Intel Xeon Phi processors; the HBM in KNL is called MCDRAM. LLVM will be used to detect the temporal locality of data (frequently-used data) in order to decide on which memory level allocations should be performed: HBM or DDR. Frequently-used data is involved in a high number of memory operations. However, in the presence of computation-heavy kernels, the latency of memory load operations will be hidden, via pipelining with computations. Therefore, a ratio of memory over computation operations should be computed in lieu of the sole number of memory operations. Moreover, since the latency of irregular accesses is higher than the one for contiguous ones, this should be also added in the metric used in the compiler analysis.

We introduce a priority function to estimate, for each dynamically-created variable in a program, whether it corresponds to high-bandwidth-critical data or not. The resulting new pass is called Bandwidth-Critical Data Analysis (BCDA). The compiler will associate a priority to each variable; the priority function depends on the ratio of memory uses over computations, the latency of operations (irregular vs. regular) and the presence of simultaneous vs. individual accesses. Admittedly, our design is the result of many trade-offs between the bandwidth, latency and size of data, and is experimentally

978-1-5090-3878-7/16 $31.00 © 2016 IEEE
DOI 10.1109/LLVM-HPC.2016.7

validated by our use case study. Note also that this cost model can be extended with other metric parameters such as the size of data.

Once the priority information is generated, the next step is to allocate higher-priority data in the HBM. In this work, the actual allocation on the MCDRAM is performed using the LLVM compiler via a static code transformation that transforms `malloc` calls to calls to a new function, namely `memkind_alloc`. This function, at run time, allocates non-zero-priority variables on the HBM if HBM is present and a sufficient size of HBM is available.

As a use case for validating our approach, we show how the Conjugate Gradient (CG) benchmark from the NAS Parallel suite can be optimized for the use of MCDRAM of Knights Landing. We implement our Bandwidth-Critical Data analysis in the LLVM compiler and apply it on CG to detect when it is beneficial to allocate data in the HBM. Then, we allocate the data in the MCDRAM using *hbwmalloc* library calls. Note that our analysis pass can also be used by analysis and profiling tools to guide the user when trying to take advantage of HBM in the system. Using the priority generated by our analysis, we achieved a $\times 2.29$ performance improvement using the LLVM compiler and $\times 2.33$ using Intel's compiler compared to a `malloc`-based version of CG.

The three main contributions of this paper are:

- the design and LLVM implementation of Bandwidth-Critical Data Analysis, which required extending the use-def chains analysis of LLVM to the interprocedural context and analyzing patterns of memory accesses;
- a transformation pass in LLVM that, based on the results generated by BCDA, transforms `malloc`s into calls to a new function, `memkind_alloc`, which calls `hbw_malloc`, at run time, to allocate Bandwidth-Critical Data that can fit within the MCDRAM in the MCDRAM memory – otherwise, the original `malloc` is kept;
- the application and validation of this analysis using NAS-NPB Conjugate Gradient Benchmark on the KNL's MCDRAM.

After this introduction, we describe the LLVM compilation framework and provide an overview of KNL and its HBM, MCDRAM, in Section II. The methodology for automatic HBM allocations, which we motivated in Section III, including our Bandwidth-Critical Data analysis design and implementation, is introduced in Section IV. Section V illustrates and discusses performance results of the application of Bandwidth-Critical Data analysis on NAS-NPB Conjugate Gradient Benchmark. We survey related work in Section VI. We discuss future work and conclude in Section VII.

II. Background

In this section, we give an overview of the LLVM compiler and KNL architecture.

A. LLVM Optimization Framework

LLVM [2] is an open-source compilation framework that uses an intermediate representation (IR) in Static Single Assignment (SSA) [3] form. Clang is the LLVM front-end for C/C++/Objective-C compilation. LLVM includes multiple tools, especially the LLVM optimizer (opt) and Polly. opt is the modular LLVM optimizer and analyzer. It can perform different LLVM-specific optimizations or/and analyses on an input source file; the output is an optimized IR file or pretty-printed analyses. As a complement to opt, Polly [4] is a high-level loop and data-locality optimizer for LLVM. Polly uses the polyhedral model and employs an abstract mathematical representation based on integer polyhedra to analyze and optimize the memory access patterns of a program. Polly also performs classic loop transformations, especially tiling and loop fusion, to improve data locality.

LLVM is widely used, mainly thanks to its rich backend system. In fact, LLVM provides code generators for x86, x86-64, PowerPC, PowerPC-64, ARM, Thumb, SPARC, Alpha, CellSPU, MIPS, MSP430, SystemZ, and XCore. In this work, we extend LLVM by an analysis and a transformation pass to efficiently and automatically allocate data in the HBM.

B. KNL Architecture

The Knights Landing (KNL) Many Integrated Core (MIC) architecture [5] is the second generation of the Intel Xeon Phi family of products. As shown in Figure 1, KNL is composed of up to 36 physical tiles interconnected by a 2-D mesh on a single die. Each tile is composed of two wide out-of-order cores and a shared L2 cache. Each core is, furthermore, composed of two 512-bit vector processing units (handling AVX-512 instructions). KNL uses a distributed tag directory to keep the L2 caches in all tiles coherent with each other. Each tile contains a caching/home agent that holds a portion of the distributed tag directory and serves as the connection point between the mesh and the tile. The KNL has two kinds of memory: (1) a High-Bandwidth Memory of 16-GigaBytes (GB) Multi-channel DRAM (MCDRAM) with up to 490 GB/s bandwidth, and (2) Low-Latency Memory of up to 384 GB at double data rate (DDR4), with a 90 GB/s bandwidth. The MCDRAM can be configured in 3 modes:

1) cache mode, where the MCDRAM serves as a cache layer between L2 and DDR and is transparent to software;
2) flat mode, where the MCDRAM is used as an addressable high-bandwidth memory and shares the physical address space with the DDR memory;
3) hybrid mode, where part of the MCDRAM is used as cache and part as a DDR extension.

The 2-D KNL KNL mesh interconnect supports 3 types of cluster modes.

All-to-all In this mode, there is no affinity between tile, directory and memory. Hence, an L2 cache miss can result in a request being sent to the tile that owns the tag directory, yielding a request sent to either MCDRAM

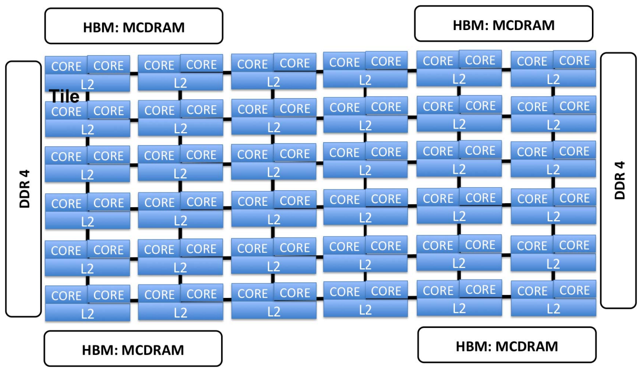

Figure 1: Knights Landing CPU and memory hierarchy

or DDR, depending on the location of the actual address; this will be followed by the request being serviced by the memory controller to the initial requesting tile over the 2-D mesh. This generally results in poor memory access performance.

Quadrant The tiles are divided into four quadrants. There is no affinity between tiles and directory, but there is one between directory and memory. Hence a tag directory will forward memory requests to a controller within the quadrant where the tag directory lies.

Sub-NUMA Clustering (SNC) Each quadrant (cluster) is considered as a separate NUMA domain by the OS. This configuration looks analogous to a 4-socket Xeon. There is affinity between tile, tag directory and memory and requests are localized to quadrants to which the tile belongs.

In this paper we are interested in the flat and Sub-NUMA clustering modes, which are the recommended ones for HPC applications striving to take full advantage of the HBM.

C. Programming KNL MCDRAM

Using MCDRAM as addressable memory requires more application development than in the simpler cache-based model. This can be performed in four ways, using (1) calls to the *hbwmalloc* library [6], which is the high-bandwidth memory interface, (2) calls to the Intel *memkind* library [7], calls to the AutoHBW library [8], which is part of memkind, or (4) via the `numactl` command.

In the first case, the allocation of `fv` into DDR is done using:

```
float *fv;
fv = (float *)malloc(sizeof(float)*n);
```

while allocation into MCDRAM is achieved via:

```
float *fv;
fv = (float *)hbw_malloc(sizeof(float)*n);
```

The second approach uses the Intel memkind library [7], which is a memory management system that can support any number of memory classes. This heap manager enables allocations to memories having different properties. The memkind library is built on top of the existing Libnuma API; one example of memkind API call is `memkind_malloc`. While memkind and hbwmalloc are for heap allocation in C, allocatable arrays in Fortran are managed using a different mechanism. For this, Intel has a `FASTMEM` directive in their Fortran 16.0 compiler[1] which can be used as follows:

```
!DIR$ ATTRIBUTES FASTMEM :: object
```

The third technique uses the AutoHBW API [8], which automatically allocates data in the MCDRAM, with no need for user application modification or recompilation. A threshold size can be specified using an environment variable, namely `AUTO_HBW_SIZE`; only the data of size above this threshold will be HBM-allocated.

Finally, the `numactl` Linux command line can be used when a whole application can fit within the MCDRAM. In this paper, we are interested in the case when only portions of the data set can fit in the HBM.

III. Motivation

HBM achieves high performance with power requirements reduced compared to DDR. The best case scenario for the user is when all of her application data fits into the HBM: the user does not have to change her application (using `numactl`, in the case of KNL). However, in the case where the data sets are larger than the HBM size, the user has to detect the bandwidth-critical parts and adapt these portions to make them reside in the HBM. The user needs analysis tools to guide her in this selection process or to perform the selection and allocation transparently. If AutoHBM provides the size of data as a metric for helping in this decision process, there are other factors that should be used to determine what is bandwidth-critical data. VTune [9] performance analyzer has the capability to provide profiling information about the high-bandwidth parts of user code. However, with VTune, the user needs to run her application at least once, to be able to make the selections, and then introduce explicit library calls in her code to perform the allocations in the HBM in subsequent runs. In this paper, we design and implement a static analysis to detect bandwidth-critical data. The result of this analysis can be used by the user to manually change her code or by compiler/runtime to automatically transform DDR allocations to HBM allocations. This can be completely transparent to the user.

Here we use two well-known and heavily-used scientific benchmarks to motivate our analysis. The first code, Stencil, shown in Figure 2, represents a portion of the OpenMP version of a 3D 7-point stencil computation. Stencil codes present a high level of data reuse between neighboring points; this reuse grows linearly with the number of array elements. In this code snippet, the data reuse is manifested in one kernel for the array `A0`; the code is parallel (except for the outer loop).

For performance experiments, we use a one-node machine with one Intel(R) Xeon Phi(TM) CPU 7210 processor @ 1.30GHz configured with the Sub-NUMA clustering mode.

[1]See https://software.intel.com/en-us/node/580357

```
#define Index3D(nx,ny,i,j,k)                \
                 ((i)+nx*((j)+ny*(k)))
...
for (t = 0; t < timesteps; t++) {
#pragma omp parallel for
  for (k = 1; k < nz - 1; k++) {
    for (j = 1; j < ny - 1; j++) {
      for (i = 1; i < nx - 1; i++) {
        Anext[Index3D (nx, ny, i, j, k)] =
         A0[Index3D(nx, ny, i, j, k + 1)] +
         A0[Index3D(nx, ny, i, j, k - 1)] +
         A0[Index3D(nx, ny, i, j + 1, k)] +
         A0[Index3D(nx, ny, i, j - 1, k)] +
         A0[Index3D(nx, ny, i + 1, j, k)] +
         A0[Index3D(nx, ny, i - 1, j, k)] -
         6.0 * A0[Index3D(nx, ny, i, j, k)]
              /(fac*fac);
      }
    }
  }
  temp_ptr = A0;
  A0 = Anext;
  Anext = temp_ptr;
}
```

Figure 2: Snippet code from a 3D 7-point stencil derived from the heat equation benchmark (the sequential version has been extracted from [10])

This KNL processor has 32 tiles (64 cores with 4 threads per core), 16 GB of MCDRAM and 96 GB of DDR4. We used ICC 16.0.3 and LLVM 3.9, sporting Clang 3.9 for compiling OpenMP and sequential codes. We used the -O3 option for both ICC and LLVM compilations.

We show in Figure 3 the performance results of running the OpenMP version of our 3D 7-point stencil on KNL using two compilers: the Intel compiler and LLVM. We can see the advantage of putting data in the high-bandwidth memory of KNL, namely MCDRAM, especially for larger stencils. Stencil codes thus fall under the category of memory-bandwidth-bound codes.

The memkind library provides a mechanism to set a fallback policy, used when insufficient high-bandwidth memory is available. In our experiments, we used the following statement:

```
hbw_set_policy(HBM_POLICY_BIND);
```

in order to ensure that, if insufficient high-bandwidth memory is available to satisfy an allocation request, an error will be returned. Thus, all our allocation requests in the HBM will actually reside in the HBM.

The second code, shown in Figure 4, is extracted from the function `conj_grad` of the OpenMP C version of the NAS Parallel CG benchmark [11]. Here, we show two kernels where data reuse is manifested for the arrays `p` and `q`. The CG benchmark is a more complicated case than Stencil because it contains different types of memory accesses and several matrix and vector multiplications and additions. Note that a pseudo-code algorithm to show the arrays involved in this computation can be found in [12]. In this case, there are

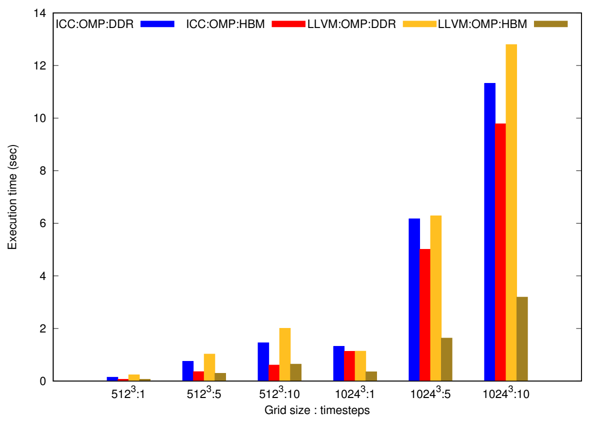

Figure 3: DDR vs. HBM execution time of OpenMP version of 3D 7-point Stencil

multiple considerations to take into account while allocating data in the MCDRAM, since an application only takes full advantage of KNL if it is highly parallelizable and memory movements are regular, i.e., with a constant stride. Arrays that are used in irregular memory accesses in CG, namely `p` and `z`, will hurt the performance if allocated in the MCDRAM. The arrays `A`, `q` and `r`, which are used with contiguous accesses, should reside in the MCDRAM. Finally, `x` should be given a low priority to be allocated in the MCDRAM because it is not used in a store operation. Store operations are more costly than load operations.

```
for (cgit = 1; cgit <= cgitmax; cgit++){
  ...
  #pragma omp for
  for (j = 0; j < lastrow - firstrow + 1; j++) {
    suml = 0.0;
    for (k = rowstr[j]; k < rowstr[j+1]; k++) {
      suml = suml + a[k]*p[colidx[k]];
    }
    q[j] = suml;
  }
  #pragma omp for reduction (+:d)
  for (j = 0; j < lastcol -firstcol + 1; j++){
     d = d + p[j] * q[j];
  }
  ...
}
```

Figure 4: Snippet code from the NAS-NPB CG benchmark (from [11])

An another consideration to be taken into account when using HBM in our allocation strategy is how sequential code performs on KNL when allocating data in the MCDRAM. One property of MCDRAM is that it is physically placed on-package, with many wires into it. Because of the many wires, many memory locations can be accessed simultaneously. However, if the code is sequential or the parallel region is executed by only one thread (indicated by the `omp single` or `omp master` constructs of OpenMP), individual accesses to the HBM take about the same time as that of the DDR memory. We show in Figures 5 and 6 results of running the sequential versions of Stencil and CG using both the LLVM and Intel compilers. These results confirm that a program with individual data accesses does not take advantage of the HBM.

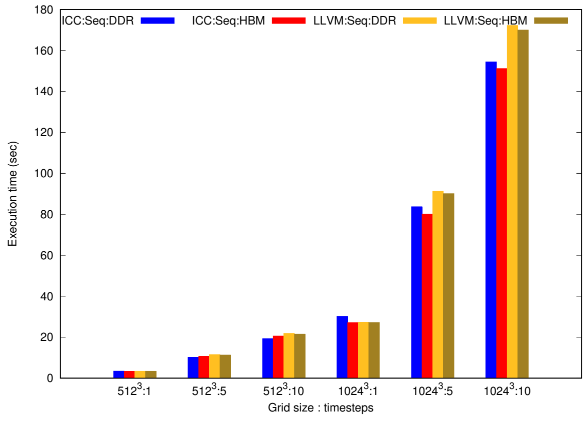

Figure 5: DDR vs. HBM execution time of the sequential version of 3D 7-point Stencil

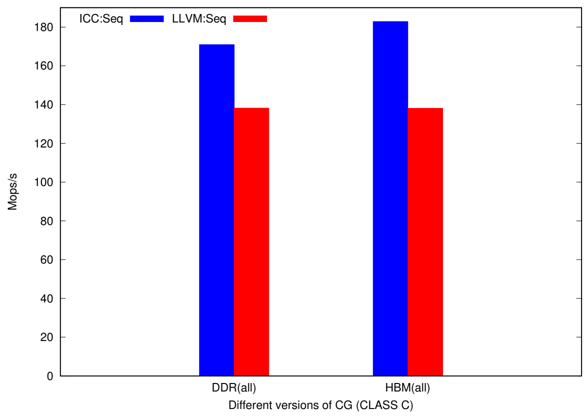

Figure 6: DDR vs. HBM execution time of the CG sequential version

IV. HBM Allocation Methodology

Data reuse can occur within the same instruction inside one loop (case of Stencil), between different loops in the same function or between loops in different functions (case of CG). In this paper, we address these three cases in a compiler analysis pass that assigns a priority value to arrays to help decide which array(s) should reside into the HBM. This priority function is fed to a compiler transformation pass that automatically allocates higher-priority data in the HBM. This section details the methodology used for HBM allocations.

A. BCDA Design and Implementation

In this paper, we create a compiler analysis, called Bandwidth-Critical Data Analysis (BCDA), that intends to detect bandwidth-bound data by calculating a priority $\mathcal{P}(\mathcal{R}(v))$ for each variable v. We assume, for this computation, the existence of a function $\mathcal{R}$ that provides, for any variable v, the set $\mathcal{R}(v)$ of its references in a given program. To estimate how much the references $R = \mathcal{R}(v)$ to the data associated to a variable v is bandwidth-critical, the priority computation $\mathcal{P}(R)$ relies on the following three weighted metrics;

1) access pattern $bandwidth(R)$, which tests if the set of (accesses to the) references R can be described as a linear function of loop iterations or if it is irregular;
2) data reuse cost $cost(r)$, which is the cost of a memory operation on a reference $r \in R$;
3) individual vs. simultaneous access $workshare(r)$, testing whether $r \in R$ is part of a sequential or parallel programming construct.

We gather these various traits into a single formula defining the priority function $\mathcal{P}$:

$$\mathcal{P}(R) = bandwidth(R) \sum_{r \in R} cost(r) workshare(r) \tag{1}$$

with

$$cost(r) = \begin{cases} 2 & \text{if } r \text{ is a store operation} \\ 1 & \text{otherwise} \end{cases} \tag{2}$$

$$workshare(r) = \begin{cases} 0 & \text{if } r \text{ is individual} \\ 1 & \text{if } r \text{ is simultaneous} \end{cases} \tag{3}$$

$$bandwidth(R) = \begin{cases} 1 & \forall r \in R,\ r \text{ is regular} \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Note that this metric could be extended with other metric parameters, such as the size of the underlying variable.

Interprocedural Memory Operations Count ($\sum$) The sum used in the priority formula function counts the number of significant memory operations in R. In practice, these memory operations refer to the set of references $R = \mathcal{R}(v)$ for a given variable v. To implement this sum, we use the existing use-def chain analysis of LLVM, which provides the set of users of each value definition in LLVM.

However, this analysis is intra-procedural, i.e., it is limited to the function where the definition occurs. In our work, the analysis pass recurses when encountering uses of a given variable as an instance of a function call argument. Thus, our analysis is able to get the inter-procedural use/def instances of values. This extension is required; for instance, to get the number of use/defs of the matrix `A` in CG, we need to visit the main function, then the `conj_grad` procedure, and also the outlined function resulting from the OpenMP parallel regions that are passed as an argument to the OpenMP library routine `__kmpc_fork_call`. Thus, we count the number of memory operations in the generated LLVM IR, namely `load`, `store` and `getelementptr` instructions used to compute the address of a given reference.

Data Reuse Cost (*cost*) Store operations are more costly than load operations. The function *cost* is used to assign a weight to each reference operation. We chose, based on an empirical study, the weight 1 for a load operation and 2 for a store operation.

Individual vs. Simultaneous Access ($workshare$) As a use-case study, we chose OpenMP because it is the defacto shared-memory programming language standard. Our analysis will detect if an access r has been performed in an OpenMP work-sharing region or not. An individual reference corresponds to a memory access that happens in sequential regions such as `single` or `master` regions in OpenMP or sequential code. A simultaneous/parallel access, on the other hand, corresponds to an access that will be performed by multiple threads at once. Since using MCDRAM gives comparable results to using DDR in the case of single OpenMP regions and sequential codes, we set $workshare$ to 0 for this particular case. Otherwise, the region is work-sharing, and $workshare$ is set to 1.

Regular vs. Irregular Access Pattern ($bandwidth$) To detect if a given R is regular or not, we analyze the subscripts present in all the references r in R, and represent this information as 1 if all accesses are regular or irregular. Irregular memory accesses make the code latency-bound and thus it should be an absorbing coefficient (0) to the priority function. For instance, `p` and `z` in the CG code are used with irregular accesses. Note that this is a very important metric parameter, because arrays with irregular accesses will hurt performance if allocated in the MCDRAM. The value of $bandwidth(R)$ has been chosen, based on empirical study, to be 0 if at least one access $r \in R$ is irregular.

In our LLVM prototype, regarding irregular accesses, we only consider the case of indirect accesses. This corresponds to analyzing the indices arguments of the `getelementptr` instruction used in computing the reference address of r. We analyze the previous uses of these indices, and mark r as irregular if at least one of these previous uses has been computed using a `getelementptr` instruction, thus indicating that r is an indirect access.

B. Allocation Transformation Design and Implementation

Once the priority function is generated for each variable, the next step is to transform `malloc` operations to high-bandwidth allocation calls, via `hbw_malloc` instructions in the case of KNL. Making this transformation automatically will require to find adequate solutions to the following challenges. Since performance portability is a key goal here, we cannot statically generate a single code that will run on every system node, because of the variety of the node architectures: systems that do not contain HBM at all, systems that contain MCDRAM, systems with HBM2 of NVIDIA Pascal, etc. Moreover, the size of the data and of the different kinds of memories are often run-time information.

In this work, we added in LLVM a transformation pass that, based on the priority value generated by BCDA for every variable, replaces `malloc` calls to calls to `memkind_alloc`. This system-specific function, at run time, allocates non-zero-priority variables on the HBM if such a HBM exists and is of sufficient size. The goal of this transformation pass is to provide a preliminary prototype for our analysis in the case of MCDRAM while addressing the issue of performance portability. As a simple example, the code for the allocation of the integer array a:

```
int *a = malloc(sizeof(int) * n);
```

which is represented in the LLVM IR as follows:

```
%call3 = call i8* @malloc(i64 %mul)
%6 = bitcast i8* %call3 to i32*
store i32* %6, i32** @a, align 8
```

will be transformed using our transformation pass (*HBMTransform*)[2] into the following:

```
%call31 = call i8* @memkind_alloc(i64 %mul)
%6 = bitcast i8* %call31 to i32*
store i32* %6, i32** @a, align 8
```

In order to implement our transformation pass, we extend the existing LLVM runtime library, namely *compiler-rt*, with the code definition of the `memkind_alloc` function. This function first queries the system using `hbw_check_available()` to check if MCDRAM is available (handling the cases when KNL is booted in cache mode or of processors with no MCDRAM at all). Then, in case of success, it allocates data there using `hbw_malloc`; otherwise, the original `malloc` call is preserved. This function calls `hbw_set_policy` with argument `HBW_POLICY_PREFERRED` in order to ensure that, if sufficient memory is not available from the high bandwidth memory at allocation time, a fall back to DDR allocation is used. Note that this definition can be extended in the future to handle other kinds of memories.

V. Experimental Evaluation

We, first, extended the LLVM 3.9 compiler, sporting Clang 3.9, with our BCDA pass to analyze the LLVM intermediate representation of programs. We then applied our analysis on the C OpenMP version of the NAS-NPB Conjugate Gradient Benchmark kernel (since we use Clang as front end for LLVM, the input code that can be used for this work had to be written in C/C++). NAS-NPB CG uses the conjugate gradient method to solve an unstructured sparse linear system. We used the -O3 option for both ICC and LLVM compilations.

The largest matrix used in the code extracted from [11] is the floating-point matrix `A`. The main procedure is `conj_grad`, which contains 7 matrix-vector, vector-vector multiplication and vector-vector addition kernels. These 7 kernels are contained within a loop of 25 iterations. CG operates on 6 floating-point arrays, namely `A`, `x`, `z`, `p`, `q` and `r`.

[2]Command *opt -load /home/dkhaldi/LLVM-HBM/build/lib/LLVMMemory.so -HBMTransform example.ll*

Our analysis generates the information presented in Table I. As discussed in Section III, our analysis detects that `z` and `p` are used in irregular memory accesses. `x` is used once with no store operations. All arrays are used in work-sharing OpenMP regions. Note that we analyzed only the part included in the timing-enabled code (Timer `T_bench` in CG code).

Table I: Critical data analysis results for the CG benchmark

FP Array	*cost*	*workshare*	*bandwidth*	$\mathcal{P}$(FP Array)
r	46	all parallel	regular	46
q	21	all parallel	regular	21
A	17	all parallel	regular	17
x	16	all parallel	regular	16
p	29	all parallel	irregular	0
z	21	all parallel	irregular	0

Based on these results, `memkind_alloc` calls generated by our transformation pass implemented in the LLVM compiler allocate the non-zero-priority arrays in MCDRAM using `hbw_malloc`. Note that, when using the ICC compiler to generate binary code, we have to transform the allocation calls manually because our transformation pass has been only implemented in LLVM. However, our analysis pass can be applied within any compiler or by the user via the output prettyprinted in Table I; the user can indeed allocate manually her data based on Table I.

We show in Figure 7 the performance results of running the OpenMP version of CG on the KNL machine described in Section III. We show in this figure the results for the cases where we allocate all or less-prioritized arrays in MCDRAM. These results are confirmed using both the ICC and LLVM compilers. Note that we are not comparing here LLVM against ICC but we evaluate whether our analysis gives good performance regarding the use of HBM for both compilers.

Arrays that are used with irregular memory accesses, namely `p` and `z`, hurt the performance when allocated in the MCDRAM. This is due to the fact that the code operating on these arrays is latency-bound. Since the latency of DDR is lower than the one of MCDRAM, `p` and `z` should be allocated in DDR (using `malloc`); other arrays, namely `A`, `q` and `r`, which are used with predictable and contiguous accesses, have higher priority and can be allocated in MCDRAM. Also, `x` is given a lower priority for MCDRAM allocation, because it is not used in a store operation.

One possibility to improve BCDA is to take into account the size of data as an additional metric factor when computing priority. For instance, `A` has an access frequency close to the one of `x` but, in fact, requires significantly more bandwidth than `x`, because it is a large matrix; that's why its allocation into the MCDRAM gives considerable improvement. `A` should thus receive a higher priority if the size parameter were introduced into the priority calculation. Since this information cannot always be provided by the compiler, we can envision the use of hints from the user about how big the arrays in her code are. Also, our analysis can be combined with runtime libraries such as AutoHBW to avoid allocating data of size less than a specific threshold.

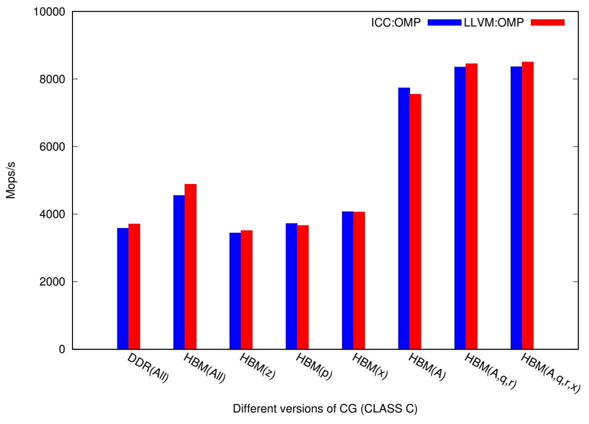

Figure 7: DDR- vs. HBM-array-allocation performance of the OpenMP version of CG

VI. Related Work

At the API level, Sequoia [13], Legion [14], [15], Resilient Distributed Datasets (RDDs) and Adios [16] added support for abstractions to allocate and move data across the memory hierarchy. Also, the Intel memkind library [7] can manage any number of memories. Recently, the memkind library has been used in High Order Seismic Simulations [17] to place data structures in a specific memory region in the KNL. This work exploited the two-level memory subsystem and 2D mesh interconnect of the KNL. Specifically, fast-running data has been placed manually in the MCDRAM whereas slow-running data has been placed manually in the DDR4. Their experiments showed that KNL outperforms the Xeon Phi Knights Corner (KNC) by more than 2.9x in time-to-solution. Similarly, Cray and Intel proposed a draft extension to OpenMP via new directives to convert standard heap allocations to high-bandwidth memory allocations. The extension from Cray introduces the `#pragma memory(bandwidth)` directive, where the bandwidth attribute maps to KNL MCDRAM. The extension from Intel adds a `memkind` directive as follows:

```
float a[n];
#pragma omp memkind(fastmem:a)
```

The goal of our approach is to automate this process in order to reduce the burden on the programmer side.

At the compiler level, memory hierarchy optimizations using loop transformations, integer linear programming and reuse buffer allocation to optimize the on-chip memory allocation and that involve scratch-pad and caches were proposed in [18], [19], [20], [21], [22], [23], [24], [25]. The main technique is to allocate the intermediate buffers that handle frequently-used data in fast memories in order to reduce the cost of data movement. The scope of these compiler approaches is per loop nest. In our work, we analyze whole programs to extract the reuse information and access patterns between loop nests, within the same function and also inter-procedurally.

At the runtime level, a memory hierarchy resource-aware online read algorithm to relieve read contention issues on SSDs is presented in [26]. In this latter paper, thresholds are set to control the read contention on SSDs depending on the number of parallel readers and total read size. The algorithm will make the system use a smaller number of processes to relieve contention if both the total read size and the number of parallel readers reach certain thresholds. We also mentioned VTune, which provides dynamic bandwidth profiling. Our analysis is static and is used by the compiler to make the HBM allocations completely transparent to the user. However, our future work will investigate using, at run time, the size of the HBM and the size of the data to make the transformation online *à la* AutoHBM. Also, we can consider using feedback analysis from profilers such as VTune to guide more accurately our transformation pass.

VII. Conclusion and Future Work

In this paper, we addressed the issue of the efficient utilization of High Bandwidth Memory from a compiler point of view. We created an analysis pass (BCDA) that detects bandwidth-critical data and allocate them in the HBM. We chose the HBM, called MCDRAM, of the Knights Landing Xeon Phi processor as a use case to assess the validity of our analysis.

We implemented our Bandwidth-Critical Data analysis and transformation in the LLVM compiler and applied it on the CG benchmarks from the NAS benchmark. Then, we allocated the data in the MCDRAM using *hbwmalloc* library calls. Using the priority generated by BCDA, we achieved a 2.29x of speedup using the LLVM compiler and a 2.33x speedup using the Intel compiler compared to the DDR version of CG. Note that our analysis pass can also be used by analysis and profiling tools to guide the user on how to best take advantage of HBM in compliant systems.

In future work, we plan to improve the accuracy of our priority function. This can be done by tuning some coefficients and including other parameters such as the size of data. Moreover, we will implement more precise analyses regarding irregular accesses and instruction counts for recursive functions and nested loops. In order to make our experiments more solid, we plan to compare allocations in the flat mode of MCDRAM with the MCDRAM in cache mode.

Acknowledgments

This material is based upon work supported in part by the National Science Foundation under the grant CCF-1409946 and the United States Department of Defense. This research was supported by resources at the Los Alamos National Laboratory and the Stony Brook University's Institute for Advanced Computational Science. The authors would like to thank the anonymous reviewers for their insightful feedback. Special thanks are also due to Pierre Jouvelot and Khaled Hamidouche, whose comments improved this work.

References

[1] *OpenMP Application Programming Interface*, http://www.openmp.org/mp-documents/openmp-4.5.pdf, Nov. 2015.

[2] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar. 2004.

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct. 1991.

[4] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1250010, 2012.

[5] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Elsevier Science, 2016. [Online]. Available: https://books.google.com/books?id=DDpUCwAAQBAJ

[6] Intel Corp. HBWMalloc: The High Bandwidth Memory Interface Library, http://memkind.github.io/memkind/man_pages/hbwmalloc.html.

[7] Intel Corp. Memkind Library, http://memkind.github.io/memkind/.

[8] Intel Corp. Auto HBW Library, https://github.com/memkind/memkind/tree/dev/autohbw.

[9] Intel VTune Performance Analyzer, http://www.intel.com/cd/software/products/asmo-na/eng/vtune/239144.htm, 2008.

[10] S. Kamil, "StencilProbe: A Microbenchmark for Stencil Applications," http://people.csail.mit.edu/skamil/projects/stencilprobe/.

[11] Sangmin Seo, Jungwon Kim, Gangwon Jo, Jun Lee, Jeongho Nah, and Jaejin Lee, http://aces.snu.ac.kr/software/snu-npb/.

[12] D.Bailey, E.Barszcz, J.Barton, D.Browning, R.Carter, L.Dagum, R.Fatoohi, S.Fineberg, P.Frederickson, T.Lasinski, R.Schreiber, H.Simon, V.Venkatakrishnan, and S.Weeratunga, "The NAS Parallel Benchmarks," Tech. Rep. RNR-94-007, March 1994. [Online]. Available: http://www.nas.nasa.gov/assets/pdf/techreports/1994/rnr-94-007.pdf

[13] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188543

[14] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing Locality and Independence with Logical Regions," in *Proceedings of the international conference on high performance computing, networking, storage and analysis*. IEEE Computer Society Press, 2012, p. 66.

[15] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "Regent: A High-Productivity Programming Language for HPC with Logical Regions," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November, 2015*, 2015.

[16] N. Podhorszki, Q. Liu, J. Logan, J. Mu, H. Abbasi, J. Choi, and S. Klasky, *ADIOS 1.7 User's Manual*, OAK RIDGE NATIONAL LABORATORY, Jun. 2014.

[17] A. Heinecke, A. Breuer, M. Bader, and P. Dubey, *High Order Seismic Simulations on the Intel Xeon Phi Processor (Knights Landing)*. Cham: Springer International Publishing, 2016, pp. 343–362. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-41321-1_18

[18] J. Cong, P. Zhang, and Y. Zou, "Optimizing Memory Hierarchy Allocation with Loop Transformations for High-level Synthesis," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 1233–1238. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228586

[19] ——, "Combined Loop Transformation and Hierarchy Allocation for Data Reuse Optimization," in *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2011, pp. 185–192.

[20] I. Issenin, E. Brockmeyer, M. Miranda, and N. Dutt, "Drdu: A data reuse analysis technique for efficient scratch-pad memory management," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 2, Apr. 2007. [Online]. Available: http://doi.acm.org/10.1145/1230800.1230807

[21] A. Aribuki, "A High-Level Programming Model For Embedded Multicore Processors," Ph.D. dissertation, University of Houston, 2013.

[22] O. Ozturk, M. Kandemir, M. J. Irwin, and S. Tosun, "Multi-Level on-Chip Memory Hierarchy Design for Embedded Chip Multiprocessors," in *12th International Conference on Parallel and Distributed Systems - (ICPADS'06)*, vol. 1, 2006.

[23] M. Kandemir and A. Choudhary, "Compiler-Directed Scratch Pad Memory Hierarchy Design and Management," in *Design Automation Conference, 2002. Proceedings. 39th*, 2002, pp. 628–633.

[24] J. Mellor-crummey, D. Whalley, and K. Kennedy, "Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings," in *International Journal of Parallel Programming*, 2001, pp. 425–433.

[25] M. Thottethodi, S. Chatterjee, and A. R. Lebeck, "Tuning Strassen's Matrix Multiplication for Memory Efficiency," in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, ser. SC '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 1–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=509058.509094

[26] W. Zhang, H. Tang, X. Zou, S. Harenberg, Q. Liu, S. Klasky, and N. Samatova, "Exploring Memory Hierarchy to Improve Scientific Data Read Performance," in *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, Sept 2015.