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

Dounia Khaldi and Barbara Chapman
Institute for Advanced Computational Science
Stony Brook University
Stony Brook, NY

The Third Workshop on the LLVM Compiler Infrastructure in HPC
Salt Lake City, Utah, November 14, 2016
Outline

• Introduction and Motivation
• Methodology:
  – Bandwidth-Critical Data Analysis (BCDA)
  – HBM Allocation Transformation
• Experimental Results using CG benchmark
• Conclusion and Future Work
Introduction: Exploring Memory Hierarchy

- New kinds of memory in new architectures
- Which data elements have to reside on these memories?
- High performance using HBM, with lower power requirements compared to DDR
- 3D Xpoint offers 1,000 times the performance of today’s SSDs
KNL Architecture as a Case Study

490 GB/s

90 GB/s
MCDRAM Configuration Modes

**Cache Mode Case**
- Tile L2 cache
- LLC miss
- HBM as cache
- Addressable Memory

**Flat Mode Case**
- Tile L2 Cache
- HBM
- Addressable Memory
Programming KNL MCDRAM: Flat Mode

• hbwmalloc library

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

• Intel memkind library
  – C, C++: memkind_malloc()
  – Fortran:
    • !DIR$ ATTRIBUTES FASTMEM :: object
    • Since Intel Fortran 16.0 compiler

• AutoHBW library
  – Threshold size: AUTO_HBW_SIZE

• numactl command
# Related Work

<table>
<thead>
<tr>
<th>Level</th>
<th>Work</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Level</td>
<td>Legion, Sequoia, RDDs, Adios</td>
<td>persistent</td>
</tr>
<tr>
<td></td>
<td>OpenMP 5.0?</td>
<td>Current proposal: #pragma omp allocate With memory spaces, allocators and traits</td>
</tr>
<tr>
<td></td>
<td>Vendors Low Level Libraries from Intel and Cray</td>
<td>Cray: #pragma memory(bandwidth)</td>
</tr>
<tr>
<td>Compiler level</td>
<td>Compiler transformations</td>
<td>Loop nests</td>
</tr>
<tr>
<td>Tools</td>
<td>VTune</td>
<td>Collect bandwidth profiles • Dynamic • bandwidth information and then what?</td>
</tr>
<tr>
<td>Tools</td>
<td>Compiler level</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>----------------</td>
<td></td>
</tr>
<tr>
<td>VTune</td>
<td>Compiler transformations</td>
<td></td>
</tr>
<tr>
<td>Vendor libraries from Intel and Cray</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We use LLVM, a widespread SSA-based compilation infrastructure for sequential and parallel languages.

### Decision Making

**Case study:** the HBM, called MCDRAM, of OpenMP code data in the HBM for sequential and parallel languages.

1. **Dynamic bandwidth information and then what?**
2. **Collect bandwidth profiles**
3. **Loop nests**
   - Cray: #pragma memory(bandwidth)
   - Memory spaces, allocators and traits

### Tools
- VTune
- Loop nests
- Dynamic bandwidth information and then what?
- Collect bandwidth profiles
- Memory spaces, allocators and traits
Motivation: Impact of MCDRAM on OpenMP 3D 7-point Stencil

• Setup: 1-node machine with one Intel(R) Xeon Phi(TM) CPU 7210 processor @ 1.30GHz configured with the Sub NUMA clustering mode
• ICC 16.0.3 and LLVM 3.8.1, with –O3
• DDR vs. HBM execution time of OpenMP version of 3D 7-point Stencil
• `hbw_set_policy(HBM_POLICY_BIND);`
What to allocate into HBM?

- Snippet code (NAS-NPB CG benchmark)
- Different types of memory accesses
- Several matrix and vector multiplications and additions

```c
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];
    }
    ...
}
```
Bandwidth-Critical Data (1)

• Many wires into MCDRAM $\rightarrow$ simultaneous access is needed
Bandwidth-Critical Data (2)

- Predictable memory access patterns → application is bandwidth-bound
- Many wires into MCDRAM → simultaneous access is needed
- Library solutions → not portable
- API level: might be a burden
  → Compiler + runtime solution
Methodology: Bandwidth-Critical Data Analysis (BCDA)

\[ R = R(v) \]

\[ P(R) = \text{bandwidth}(R) \sum_{r \in R} \text{cost}(r) \text{workshare}(r) \]

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

\[ \text{workshare}(r) = \begin{cases} 0 & \text{if } r \text{ is individual} \\ 1 & \text{if } r \text{ is simultaneous} \end{cases} \]

\[ \text{bandwidth}(R) = \begin{cases} 1 & \forall r \in R, \text{ } r \text{ is regular} \\ 0 & \text{otherwise} \end{cases} \]
BCDA: Interprocedural Memory Operations Count

\[ R = R(v) \]

\[ P(R) = \text{bandwidth}(R) \sum_{r \in R} \text{cost}(r)\text{workshare}(r) \]

- LLVM IR is in SSA form
  - One definition \(\rightarrow\) multiple uses
  - Allows for Def-Use and Use-Def chain analysis

- Interprocedural Memory Operations Count (\(\sum\))
  - \(\_\_\_\_\text{kmfc}_\text{fork}_\text{call}\)
  - Number of memory operations in the generated LLVM IR (\(\text{load, store and getelementptr}\))
BCDA: Data Reuse Cost

\[ P(R) = \text{bandwidth}(R) \sum_{r \in R} \text{cost}(r) \text{workshare}(r) \]

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

- Function \textit{cost} assigns a weight to reference operations
BCDA: Individual vs. Simultaneous Access

\[ P(R) = \text{bandwidth}(R) \sum_{r \in R} \cos t(r) \text{workshare}(r) \]

\[ \text{workshare}(r) = \begin{cases} 
0 & \text{if } r \text{ is individual} \\
1 & \text{if } r \text{ is simultaneous} 
\end{cases} \]

- OpenMP as a case study
- Function \textit{workshare} detects if an access \( r \) has been performed in an OpenMP work-sharing region or not
BCDA: Regular vs. Irregular Access Pattern

\[ P(R) = \text{bandwidth}(R) \sum_{r \in R} \text{cost}(r) \text{workshare}(r) \]

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

- Function \textit{bandwidth}: latency vs bandwidth bound
- Indirect Accesses: indices arguments of the \textit{getelementptr} instruction
int *a = malloc(sizeof(int) * n);

if defined (HAVE_HBWMALLOC_H)
# include <hbwmalloc.h>
void *memkind_alloc(size_t size) {
    int avail = hbw_check_available();
    void *a;
    hbw_set_policy(HBW_POLICY_PREFERRED);
    if (avail == 0) {
        a = hbw_malloc(size);
        assert(a != NULL);
    } else {
        a = malloc(size);
    }
    return a;
}
#else
void *memkind_alloc(size_t size) {
    void *a = malloc(size);
    return a;
#endif

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

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

compiler-rt runtime library
## Experimental Results:

Critical Data Analysis Results for the CG Benchmark

<table>
<thead>
<tr>
<th>FP Array</th>
<th>cost</th>
<th>workshare</th>
<th>bandwidth</th>
<th>P(FP Array)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>All parallel</td>
<td>regular</td>
<td>46</td>
</tr>
<tr>
<td>r</td>
<td>46</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q</td>
<td>21</td>
<td></td>
<td>regular</td>
<td>21</td>
</tr>
<tr>
<td>a</td>
<td>17</td>
<td></td>
<td>regular</td>
<td>17</td>
</tr>
<tr>
<td>a</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>16</td>
<td></td>
<td>regular</td>
<td>16</td>
</tr>
<tr>
<td>p</td>
<td>29</td>
<td></td>
<td>irregular</td>
<td>0</td>
</tr>
<tr>
<td>Z</td>
<td>21</td>
<td></td>
<td>irregular</td>
<td>0</td>
</tr>
</tbody>
</table>


Performance Results

- Setup: 1-node machine with one Intel(R) Xeon Phi(TM) CPU 7210 processor @ 1.30GHz configured with the Sub NUMA clustering mode
- LLVM 3.9, sporting Clang 3.9
- Results using:
  - Conjugate Gradient (CG) benchmark (NAS Parallel suite)
  - 2.29x performance improvement using LLVM and 2.33x using ICC
Conclusion and Future Work

• HBM management from a compiler point-of-view
  – Decide when it is beneficial to allocate data in the HBM for sequential and OpenMP code
  – Case study: HBM (MCDRAM) of Knights Landing (KNL)
  – 2.29x performance improvement using LLVM compiler and 2.33x using Intel compiler compared to the DDR version of CG

• Future Work:
  – Improve the accuracy of our priority function
  – Implement more precise analyses regarding irregular accesses and instruction counts for recursive functions and nested loops
  – Use of AutoHBW to add size as an additional metric
Towards Automatic HBM Allocation using LLVM: A Case Study with Knights Landing

Dounia Khaldi and Barbara Chapman
Institute for Advanced Computational Science
Stony Brook University
Stony Brook, NY

The Third Workshop on the LLVM Compiler Infrastructure in HPC
Salt Lake City, Utah, November 14, 2016