
Publisher's PDF, also known as Version of record

Link to publication record in Explore Bristol Research
PDF-document

This is the final published version of the article (version of record). It first appeared online via IXPUG at https://www.ixpug.org/events/spring-2017-emea. Please refer to any applicable terms of use of the publisher.

University of Bristol - Explore Bristol Research
General rights

This document is made available in accordance with publisher policies. Please cite only the published version using the reference above. Full terms of use are available: http://www.bristol.ac.uk/pure/about/ebr-terms
Improving achieved memory bandwidth from C++ codes on Intel® Xeon Phi™ Processor (Knights Landing)

Karthik Raman, Intel Corporation (karthik.raman@intel.com)
Tom Deakin, University of Bristol (tom.deakin@bristol.ac.uk)
James Price, University of Bristol
Simon McIntosh-Smith, University of Bristol

The University of Bristol is an Intel® Parallel Computing Center

Acknowledgements:
John Pennycook, Intel Corporation
Rakesh Krishnaiyer, Intel Corporation

IXPUG Annual Spring Conference 2017
• Simple memory bandwidth benchmark, based on the McCalpin STREAM benchmark.
  • STREAM is the gold-standard baseline for memory bandwidth bound kernels.

• 5 computational kernels:
  • Copy: c[i] = a[i]
  • Multiply: b[i] = \alpha c[i]
  • Add: c[i] = a[i] + b[i]
  • Triad: a[i] = b[i] + \alpha c[i]
  • Dot: sum += a[i] * b[i]

• Aims to measure achievable memory bandwidth:
  • From a variety of programming models.
  • Across a variety of multi- and many-core devices.

• Motivation:
  • Evaluate out of box performance of portable programming modes/libraries
  • Understand limitations on each & enable necessary optimizations
  • Apply learnings to other applications using similar programming models
  • If we can’t get STREAM to perform, how can we get a real-world code to perform?

• Open Source, available at GitHub:
  http://uob-hpc.github.io/GPU-STREAM/
Programming models

• OpenMP
  • Directive based threading model.
  • #pragma omp parallel for

• Kokkos
  • C++ abstraction and portability layer.
  • Lambda based compute.
  • Execution model: parallel loops.
  • Data structures: memory space and policy/access patterns.
  • parallel_for(array_size, KOKKOS_LAMBDA (const int index) {...});
  • Uses OpenMP as a backend for threading support.

• RAJA
  • C++ abstraction layer.
  • Lambda based compute.
  • Parallel loops, with IndexSets (partition loop with different execution policies).
  • forall<policy>(index_set, [=] RAJA_DEVICE (int index) {...});
  • Uses OpenMP as a backend for threading support.
Experimental setup

- Platforms:
  - Intel® Xeon Phi™ 7210 Processor
    - 64 core, 1.30 GHz
    - 16 GB MCDRAM configured in Quad/Flat, 96 GB DDR (unused)
    - 1.6 GHz mesh, 6.4 GT/s
  - Intel® Xeon® E5-2697v4 (Broadwell-EP) processor
    - 18 core/socket, 2 sockets, 2.3 GHz
    - 128 GB DDR4

- Compiler and Flags:
  - Intel® C++ Compiler 17.0
    - -O3 -xMIC-AVX512 / -xCORE-AVX2

- Problem size: 33,554,432 doubles

- Bandwidth analysis identical to STREAM.
  For Triad, 3*array size in bytes / minimum runtime.

- Launch Command:
  - OMP_NUM_THREADS=64 OMP_PROC_BIND=true
    numactl -m 1 ./gpu-stream
Performance gap for the C++ approaches.

Why don’t they match McCalpin STREAM?

Array size: $2^{25}$ doubles

Why does STREAM do well?

- STREAM is an OpenMP benchmark written in C, so why does GPU-STREAM OpenMP struggle?
  - The only difference is GPU-STREAM is a C++ code, right?

- STREAM allocates memory on the stack, with the array sizes known at compile time.

- The compiler can choose to align the memory, generating aligned loads and stores.

- The compiler can choose to generate streaming stores.
What’s your problem?

- Problems sizes of application codes usually only known at runtime.

- What happens if we modify STREAM so that problem size is known at runtime?
  - Original bandwidth: 448 GB/s.
  - Now: 270-345 GB/s.

- By allocating on the heap and setting the problem size at runtime, all this information is lost and the compiler has to ensure correctness.

- The optimizations we present for OpenMP also apply to regular STREAM with the problem size known at runtime.
Improving the OpenMP performance

- **Align the heap memory to page boundary (2MB)**
  - Allocate using
    - `_mm_malloc(*a, 2097152)`
    - OR
    - `aligned_alloc(2097152, sizeof(a)*array_size)` → C11 Standard

- **Enable non-temporal stores**
  - Compile the code with: `-qopt-streaming-stores=always`
  - This option is fine for STREAM benchmark
  - In general, recommended to use streaming stores on per loop basis via
    - `#pragma vector nontemporal [ var1, var2..]`

- **Tell compiler about aligned arrays in the loops**
  - `__assume_aligned(a, 2097152)`
  - OR
  - `#pragma omp parallel for simd aligned(a : 2097152)`
  - OR
  - `#pragma vector aligned`
    - (requires start/end of loop iteration to be multiple of SIMD length)
Compiler Optimization Reports (OpenMP code)

OpenMP Triad Loop (Baseline):

```c
#pragma omp parallel for
for (int i = 0; i < array_size; i++)
{
    c[i] = a[i] + b[i];
}
```

Unaligned accesses, Regular Stores

OpenMP Triad Loop (Optimized):

```c
#pragma omp parallel for simd aligned (a, b, c: 2097152)
for (int i = 0; i < array_size; i++)
{
    c[i] = a[i] + b[i];
}
```

Aligned accesses, Non-Temporal Stores
Improving the Kokkos performance

- Ensure memory alignment.
  - Can compile the Kokkos library specifying memory alignment.
    --cxxflags=-DKOKKOS_MEMORY_ALIGNMENT=2097152

- Enable non-temporal stores.
  - x86 Intel architecture by default does allocate on stores (RFO – Read for Ownership)
  - Streaming stores were not being generated by the compiler by default.
  - These are key to getting peak bandwidth performance
    - Large arrays with no re-use, avoid cache capacity wastage for writes.
  - Compile the code with: -qopt-streaming-stores=always
    - Can also use for McCalpin STREAM benchmark
    - In general, recommended to use streaming stores on per loop basis via
      #pragma vector nontemporal [ var1, var2..]
Improving the Kokkos performance

- Change loop iterator type.
  - Simple C implementation, loop index $i$ & array-access $a[i]$ uses “int” for loop indexing and the induction-variable
    
    ```
    e.g. for (int $i = 0; i < array_size; i++) {a[i]= ...}
    ```
  - The Kokkos version was
    ```
    parallel_for(array_size, KOKKOS_LAMBDA (const int $index$) {});
    ```
  - Kokkos library internally uses `long` data type (hardcoded) for induction variable
    - Mismatch between induction variable type and subscript type in array accesses $a[index]$
    - Mixing multiple-sized induction variables reduces compiler optimizations
  - Compiler unable to perform data-dependence multiversioning & “Peel Loop” generation automatically for aligned stores in the vectorized kernel loop
  - Change loop iterator data type in user code to `long` to match Kokkos implementation.
    ```
    parallel_for(array_size, KOKKOS_LAMBDA(const long $index$) {});
    ```
Compiler Optimization Reports (Kokkos code)

Kokkos Triad Loop (Baseline):

```cpp
const T scalar = startScalar;
parallel_for(array_size, KOKKOS_LAMBDA (const int index) {
    a[index] = b[index] + scalar*c[index];
});
```

Kokkos Triad Loop (Optimized):

```cpp
const T scalar = startScalar;
parallel_for(array_size, KOKKOS_LAMBDA (const long index) {
    a[index] = b[index] + scalar*c[index];
});
```

Remark: #15389: Vectorization support: reference this[index] has unaligned access

```cpp
remark #15389: vectorization support: reference this[index] has unaligned access
```
Improving the RAJA performance

- Enable non-temporal stores.
  - x86 Intel architecture by default does allocate on stores (RFO – Read for Ownership)
  - Streaming stores were not being generated by the compiler by default.
  - These are key to getting peak bandwidth performance
    - The arrays are large enough and there is no reuse so we do not want to use the cache capacity for writes.
- Compile the code with:
  `-qopt-streaming-stores=always`
  - Can also use for McCalpin STREAM benchmark
  - Recommended to use streaming stores on per loop basis via
    `#pragma vector nontemporal [ var1, var2..]`
Improving the RAJA performance

- Change loop iterator type
  - Change data type of “Index type” in RAJA library to “long”
    - Reduces mismatch between different sizes for induction variables & loop index bounds after all C++ abstraction routines inlined by the compiler.
    - Enables much better compiler loop optimizations.
    - Change the indices to be of type long in the user code to get better efficiency in vectorization
      e.g. forall<policy>(index_set, [=] RAJA_DEVICE (long index){
        a[index] = b[index] + scalar*c[index]; })

- Avoid “false dependencies”
  - Compiler not able to vectorize loops due to assumption of false dependencies
  - Enable “restrict” keyword in pointers to indicate no pointer aliasing, thus aiding optimizations
  - Compile RAJA with:
    -DRAJA_PTR="RAJA_USE_RESTRICT_ALIGNED_PTR"
  - Use “RAJA_RESTRICT” for the pointers in the user code.
Compiler Optimization Reports (RAJA code)

**RAJA Triad Loop (Baseline):**

```
T* a = d_a; T* b = d_b; T* c = d_c;
const T scalar = startScalar;
forall<policy>(index_set, [=] RAJA_DEVICE (int index)
    { a[index] = b[index] + scalar*c[index]; })
```

**RAJA Triad Loop (Optimized):**

```
T* RAJA_RESTRICT a = d_a; T* RAJA_RESTRICT b = d_b;
T* RAJA_RESTRICT c = d_c; const T scalar = startScalar;
forall<policy>(index_set, [=] RAJA_DEVICE (long index)
    { a[index] = b[index] + scalar*c[index]; })
```

Loop not vectorized

Peeled Loop, Vectorized main loop + Aligned non-temporal stores
# Triad Performance

<table>
<thead>
<tr>
<th>Model</th>
<th>Intel® Xeon Phi™ (Knights Landing)</th>
<th>Intel® Xeon® E5-2697v4 (Broadwell)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original GB/s</td>
<td>Optimized GB/s</td>
</tr>
<tr>
<td>McCalpin Stream</td>
<td>448</td>
<td>-</td>
</tr>
<tr>
<td>OpenMP</td>
<td>302</td>
<td>438</td>
</tr>
<tr>
<td>Kokkos</td>
<td>298</td>
<td>436</td>
</tr>
<tr>
<td>RAJA</td>
<td>124</td>
<td>436</td>
</tr>
</tbody>
</table>
Conclusions and Insights

▪ Out of the box, C++ and OpenMP struggle to show close to peak achievable memory bandwidth.

▪ Partially down to the knowledge the compiler has at compile time.
  ▪ Needs to know the alignment and trip counts to generate the best vector code.

▪ Can use OpenMP to give the compiler enough knowledge to do the right thing.

▪ Using an abstraction layer hides some detail away.
  ▪ Must ensure the abstraction layer holds enough information to generate the same best vector code.

▪ Key optimizations:
  ▪ Ensure memory alignment (Align and tell compiler).
  ▪ Remove abstraction layer loop iteration typecasts (Avoid datatype conversions)
  ▪ Non-temporal stores (for peak memory bandwidth, use only where applicable)
Website: http://uob-hpc.github.io/GPU-STREAM/


