Raman, K., Deakin, T., Price, J., & McIntosh-Smith, S. (2017). *Improving achieved memory bandwidth from C++ codes on Intel® Xeon Phi™ Processor (Knights Landing)*. IXPUG Spring Meeting, Cambridge, United Kingdom.

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

GPU-STREAM

- 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] = α c[i]
  - Add: c[i] = a[i] + b[i]
  - Triad: a[i] = b[i] + α 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

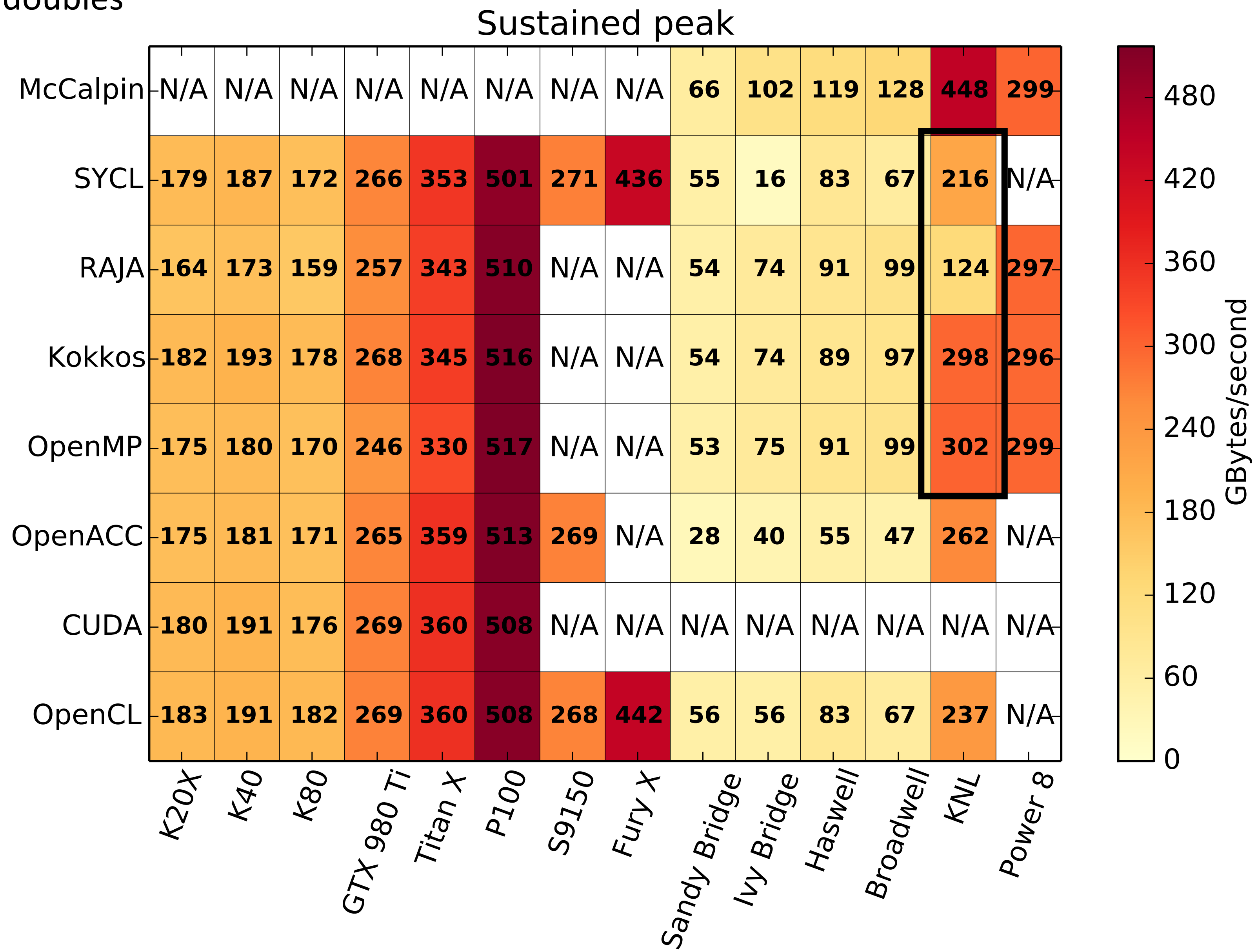
- OpenMP
  - Directive based threading model.
  - `#pragma omp parallel for`
- Kokkos
  - C++ abstraction and portability layer.
  - Lambda based compute.
  - Execution model: parallel loops.
  - Data structures: memory space and policy/access patterns.
  - `parallel_for(array_size, KOKKOS_LAMBDA (const 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
  ```

Array size: 2^25 doubles



	K20X	K40	K80	GTX 980 Ti	Titan X	P100	S9150	Fury X	Sandy Bridge	Ivy Bridge	Haswell	Broadwell	KNL	Power 8
McCalpin	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	66	102	119	128	448	299
SYCL	179	187	172	266	353	501	271	436	55	16	83	67	216	N/A
RAJA	164	173	159	257	343	510	N/A	N/A	54	74	91	99	124	297
Kokkos	182	193	178	268	345	516	N/A	N/A	54	74	89	97	298	296
OpenMP	175	180	170	246	330	517	N/A	N/A	53	75	91	99	302	299
OpenACC	175	181	171	265	359	513	269	N/A	28	40	55	47	262	N/A
CUDA	180	191	176	269	360	508	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
OpenCL	183	191	182	269	360	508	268	442	56	56	83	67	237	N/A

Performance gap for the C++ approaches.

Why don’t they match McCalpin STREAM?

Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. “GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models”, pages 489–507. Springer International Publishing, Cham, 2016.

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):

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

```
LOOP BEGIN at OMPStream.cpp(160,3)
<Multiversioned v1>
  remark #25228: Loop multiversioned for Data Dependence
  remark #15389: vectorization support: reference a has unaligned access   [
OMPStream.cpp(164,5) ]
  remark #15389: vectorization support: reference b has unaligned access   [
OMPStream.cpp(164,12) ]
  remark #15389: vectorization support: reference c has unaligned access   [
OMPStream.cpp(164,28) ]
  remark #15381: vectorization support: unaligned access used inside loop body
  remark #15305: vectorization support: vector length 16
  remark #15309: vectorization support: normalized vectorization overhead 1.778
  remark #15300: LOOP WAS VECTORIZED
  remark #15442: entire loop may be executed in remainder
  remark #15450: unmasked unaligned unit stride loads: 2
  remark #15451: unmasked unaligned unit stride stores: 1
  remark #15475: --- begin vector cost summary ---
  remark #15476: scalar cost: 10
  remark #15477: vector cost: 0.560
  remark #15478: estimated potential speedup: 14.630
  remark #15488: --- end vector cost summary ---
LOOP END
```

Unaligned accesses, Regular Stores

OpenMP Triad Loop (Optimized):

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

```
LOOP BEGIN at OMPStream.cpp(155,3)
  remark #15388: vectorization support: reference a has aligned access   [
OMPStream.cpp(159,5) ]
  remark #15388: vectorization support: reference b has aligned access   [
OMPStream.cpp(159,12) ]
  remark #15388: vectorization support: reference c has aligned access   [
OMPStream.cpp(159,28) ]
  remark #15412: vectorization support: streaming store was generated for a   [
OMPStream.cpp(159,5) ]
  remark #15305: vectorization support: vector length 8
  remark #15309: vectorization support: normalized vectorization overhead 1.429
  remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
  remark #15448: unmasked aligned unit stride loads: 2
  remark #15449: unmasked aligned unit stride stores: 1
  remark #15467: unmasked aligned streaming stores: 1
  remark #15475: --- begin vector cost summary ---
  remark #15476: scalar cost: 10
  remark #15477: vector cost: 0.870
  remark #15478: estimated potential speedup: 10.340
  remark #15488: --- end vector cost summary ---
LOOP END
```

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):

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

```
LOOP BEGIN at
KOKKOS/kokkos/install/include/OpenMP/Kokkos_OpenMP_Parallel.hpp(86,7)
inlined into KOKKOSStream.cpp(117,3)
   remark #15389: vectorization support: reference this[index] has unaligned access
[ KOKKOSStream.cpp(119,6) ]
   remark #15389: vectorization support: reference this[index] has unaligned access
[ KOKKOSStream.cpp(119,17) ]
   remark #15389: vectorization support: reference this[index] has unaligned access
[ KOKKOSStream.cpp(119,35) ]
   remark #15381: vectorization support: unaligned access used inside loop body
   remark #15305: vectorization support: vector length 16
   remark #15309: vectorization support: normalized vectorization overhead 0.455
   remark #15300: LOOP WAS VECTORIZED
   remark #15450: unmasked unaligned unit stride loads: 2
   remark #15451: unmasked unaligned unit stride stores: 1
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 13
   remark #15477: vector cost: 1.370
   remark #15478: estimated potential speedup: 8.6
   remark #15488: --- end vector cost summary ---
LOOP END
```

No Peel Loop, Unaligned regular stores

Kokkos Triad Loop (Optimized):

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

```
LOOP BEGIN at KOKKOS/kokkos/install/include/OpenMP/Kokkos_OpenMP_Parallel.hpp(86,7)
inlined into KOKKOSStream.cpp(117,3)
<Peeled loop for vectorization>
.....
LOOP END
LOOP BEGIN at KOKKOS/kokkos/install/include/OpenMP/Kokkos_OpenMP_Parallel.hpp(86,7)
inlined into KOKKOSStream.cpp(117,3)
  remark #15388: vectorization support: reference this[iwork] has aligned access   [
KOKKOSStream.cpp(119,6) ]
  remark #15389: vectorization support: reference this[iwork] has unaligned access   [
KOKKOSStream.cpp(119,17) ]
   remark #15389: vectorization support: reference this[iwork] has unaligned access   [
KOKKOSStream.cpp(119,35) ]
.......
remark #15412: vectorization support: streaming store was generated for this[iwork][
KOKKOSStream.cpp(119,6) ]
.....
remark #15300: LOOP WAS VECTORIZED
remark #15449: unmasked aligned unit stride stores: 1
   remark #15450: unmasked unaligned unit stride loads: 2
   remark #15467: unmasked aligned streaming stores: 1
......
```

Peeled Loop,
Aligned
non-temporal stores

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];
 });
```

```
LOOP BEGIN at RAJA/install/include/RAJA/exec-openmp/forall_openmp.hxx(155,1) inlined into RAJAStream.cpp(146,3)
  remark #15344: loop was not vectorized: vector dependence prevents vectorization. First dependence is shown below. Use level 5 report for details
  remark #15346: vector dependence: assumed FLOW dependence between loop_body.a[*(begin+i*4)] (148:7) and loop_body.b[*(begin+i*4)] (148:7)
  remark #25439: unrolled with remainder by 4
LOOP END
```

Loop not vectorized

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 BEGIN at RAJA/install.opt/include/RAJA/exec-openmp/forall_openmp.hxx(155,1) inlined into RAJAStream.cpp(149,3)
<Peeled loop for vectorization>
……
LOOP END

LOOP BEGIN at /RAJA/install/include/RAJA/exec-openmp/forall_openmp.hxx(155,1) inlined into RAJAStream.cpp(149,3)
remark #15412: vectorization support: streaming store was generated for loop_body.a[…]   [ RAJAStream.cpp(151,7) ]
…
remark #15300: LOOP WAS VECTORIZED
…
  remark #15449: unmasked aligned unit stride stores: 1
  remark #15450: unmasked unaligned unit stride loads: 2
  remark #15467: unmasked aligned streaming stores: 1
….
LOOP END
```

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

Triad Performance

	Intel® Xeon Phi™ (Knights Landing)		Intel® Xeon® E5-2697v4 (Broadwell)	
Model	**Original GB/s**	**Optimized GB/s**	**Original GB/s**	**Optimized GB/s**
McCalpin Stream	448	-	129	-
OpenMP	302	438	95	130
Kokkos	298	436	96	129
RAJA	124	436	96	129

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)

References

Website: http://uob-hpc.github.io/GPU-STREAM/

[1] T. Deakin and S. McIntosh-Smith, “GPU-STREAM: Benchmarking the achievable memory bandwidth of Graphics Processing Units (poster),” in *Supercomputing*, 2015.

[2] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, “GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models,” 2016, pp. 489–507.

[3] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, “GPU-STREAM: Now in 2D! (poster),” in *Supercomputing*, 2016.

[4] S. J. Pennycook, J. D. Sewall, and V. W. Lee, “A Metric for Performance Portability,” pp. 1–7.

[5] R. Krishnaiyer “Data Alignment to Assist Vectorization”, Intel® Developer Zone article, 2015. https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization

[6] K. Raman “Optimizing Memory Bandwidth in Knights Landing” Intel® Developer Zone article, 2016. https://software.intel.com/en-us/articles/optimizing-memory-bandwidth-in-knights-landing-on-stream-triad