The MEGA-STREAM benchmark on Intel® Xeon Phi™ processors (Knights Landing)

Tom Deakin, University of Bristol, UK
John Pennycook, Intel Corporation
Andrew Mallinson, Intel Corporation
Wayne Gaudin, UK Atomic Weapons Establishment
Simon McIntosh-Smith, University of Bristol, UK
The University of Bristol is an Intel® Parallel Computing Center

A (very) brief history

- SNAP mini-app (LANL) isn't getting close to peak MCDRAM memory bandwidth on Knights Landing.
  - No progress with SNAP code directly.
  - Yet, GPU version of SNAP does exploit available memory bandwidth [1], [2].
- Not sure where or what the problem is in the sweep kernel.
  - dim3_sweep.f90
  - Data access is stride 1.
  - Looks similar to STREAM (which *does* achieve good bandwidth to MCDRAM).
- Create a mini-mini-app!
  - Start simple, and add complexity from SNAP.
  - Keep going until representative, solving each problem as we go, applying solutions to SNAP.
- Use OpenMP for data parallelism.
- Open source, GPL-3.0, available at GitHub: https://github.com/UK-MAC/mega-stream

Estimating bandwidth

- Throughout we use estimated bandwidth rather than measured bandwidth.
  - The STREAM benchmark takes a similar approach.
- Look at source and count up read and writes by hand to create a model.
- Model is generally oblivious to the cache effects:
  - E.g. Once a byte is read any future reads are “free”.
- We do not assume “read for ownership” (RFO).
  - A write is counted once, as if it was a streaming store.
  - RFO is a hardware detail; it’s not a “useful” movement of memory in the context of the model.
- Assume reads/writes recounted each timestep.
- Measured bandwidth would be that reported by Intel® VTune™ Amplifier XE.
  - Comparison between these numbers can be useful (see Conclusions for rule of thumb).

Experimental setup

- Platform:
  - Intel® Xeon Phi™ 7210 Processor
  - 1.30 GHz
  - 16 GB MCDRAM configured in Quad/Flat, 96 GB DDR (unused)
  - 1.6 GHz mesh, 6.4 GT/s
  - CentOS 7.2, XPPSL 1.5.1
- Compiler and Flags:
  - Intel® C++ Compiler 17.0.2
  - Transparent huge pages enabled
  - `-O3 -xMIC-AVX512 -qopt-report=5 -g -debug inline-debug-info`
- Launch Command:
  - `OMP_NUM_THREADS=64 OMP_PROC_BIND=true numactl -m 1 ./mega-stream ${OPTIONS}`

Version 0.1

- Original hypothesis was streaming many arrays (with different sizes) causes memory bandwidth limits not to be reached; resulted in latency becoming a dominant factor.
- Start with the Triad kernel from STREAM, and add more arrays with different sizes:

```
#pragma omp parallel for
for int i = 0; i < L_size; i++)
{
    r[i] = q[i] +
      a[i&S_mask]*x[i&M_mask] + b[i&S_mask]*y[i&M_mask]
      + c[i&S_mask]*z[i&M_mask];
}
```

- q and r are large; x, y and z are medium; and a, b and c are small in size.
- “& mask” is equivalent to “% size” as we assume arrays are powers of 2 in length.

Initial performance analysis

- None of the results are close to the 490 GB/s from STREAM.
- Code is being vectorised, but, gathers are generated even though most loads are contiguous.
  - Modular arithmetic (%) means indices might wrap around.
- Optimised via alignment and strip mining loop, and streaming stores.
- Really helps “small“, which was instruction (gather) bound.
- Only one write stream, so “large” is dominated by read bandwidth.
- Little change with default (mixed) sizes.
- With “medium”, arrays fall out of cache.
  - Measured bandwidth from Intel® VTune™ Advisor XE is much higher.
  - We should probably have picked up on this…
- Little improvement when these optimisations are applied to SNAP. ☹

Problem size	Array sizes	Original GB/s	Optimized GB/s
Default	r,q: 2^27, x,y,z: 2^23, a,b,c: 128	104.0	101.3
Small	r,q: 2^27, x,y,z,a,b,c: 128	197.0	407.0
Medium	r,q: 2^27, x,y,z,a,b,c: 2^23	69.6	70.3
Large	r,q,x,y,z,a,b,c: 2^27	333.3	340.1

Version 0.3

- Needed to better capture SNAP data access patterns.
  - Make benchmark code *more representative* of SNAP.
- Add additional loops, changing accesses into multi-dimensional arrays.
  - Used an indexing macro, but could also cast to a VLA.
- Add updates += to “medium” sized arrays.
  - Creates an interesting reuse pattern.
- Add a reduction over the inner-most loop.

```c
#pragma omp parallel for
for (int m = 0; m < Nm; m++) {
  for (int l = 0; l < Nl; l++) {
    for (int k = 0; k < Nk; k++) {
      for (int j = 0; j < Nj; j++) {
        double total = 0.0;
        #pragma omp simd reduction(+:total)
        for (int i = 0; i < Ni; i++) {
          /* Set r */
          r[IDX5(i,j,k,l,m,Ni,Nj,Nk,Nl)] =
            q[IDX5(i,j,k,l,m,Ni,Nj,Nk,Nl)] +
            a[i] * x[IDX4(i,j,k,m,Ni,Nj,Nk)] +
            b[i] * y[IDX4(i,j,l,m,Ni,Nj,Nl)] +
            c[i] * z[IDX4(i,k,l,m,Ni,Nk,Nl)];

          /* Update x, y and z */
          x[IDX4(i,j,k,m,Ni,Nj,Nk)] =
            0.2*r[IDX5(i,j,k,l,m,Ni,Nj,Nk,Nl)] - x[IDX4(i,j,k,m,Ni,Nj,Nk)];

          y[IDX4(i,j,l,m,Ni,Nj,Nl)] =
            0.2*r[IDX5(i,j,k,l,m,Ni,Nj,Nk,Nl)] - y[IDX4(i,j,l,m,Ni,Nj,Nl)];

          z[IDX4(i,k,l,m,Ni,Nk,Nl)] =
            0.2*r[IDX5(i,j,k,l,m,Ni,Nj,Nk,Nl)] - z[IDX4(i,k,l,m,Ni,Nk,Nl)];

          /* Reduce over Ni */
          total += r[IDX5(i,j,k,l,m,Ni,Nj,Nk,Nl)];

        } /* Ni */

        sum[IDX4(j,k,l,m,Nj,Nk,Nl)] += total;

      } /* Nj */
    } /* Nk */
  } /* Nl */
} /* Nm */
```

Loop	Name	Default size
Nm	outer	64
Nj, Nk, Nl	middle	16
Ni	inner	128

Initial results

- The code doesn't get good useful bandwidth out of the box.
- We're already aligning to 2MB pages and it does vectorise.
- Using `-qopt-streaming-stores=always` helps.
  - But we probably don't want streaming stores for arrays with reuse…
  - Code generated peel loop for r only.
- Wide variation depending on problem size.

Version	Bandwidth GB/s
Initial	75.2
+ streaming stores	107.1

Optimisations (1/3)

- Streaming stores:
  - Only want a streaming store for r, other arrays have reuse.
  - Enable them with a compiler directive:
    `#pragma vector nontemporal(r)`
  - Prevents r data polluting the cache.
  - No “read for ownership” so array simple written to.
  - Arrays need to be aligned for streaming stores.
    - Baseline aligned to 2MB pages.
    - Could ensure generate aligned instructions via OpenMP aligned clause but no penalty on KNL for using unaligned loads on aligned data.

Optimisations (2/3)

- Cache blocking:
  - x, y and z arrays are reused in the middle and outer loops.
  - We want them to be cached for the inner i loop.
  - For sufficiently large Ni * Nj * Nk, x will fall out of cache.
    - By default, x is 128 * 16 * 16 * 8 bytes = 256 KiB.
    - L2 cache is 512 KB per core, so x, y and z will not fit in cache!
  - Restructure the loops and data to promote data-reuse:
    - Split Ni loop into groups of vectors / cache lines.
    - Add an extra loop, and an extra dimension to all arrays which index i.
  - Decreases the size of data we must keep in cache to VLEN * Nj * Nk.
    - By default: 8 * 16 * 16 * 8 = 16 KiB.

Optimisations (3/3)

- Software prefetching:
  - Intel® VTune™ Amplifier XE shows L2 cache misses for load of the q array.
    - Used hardware counters `L2_HIT_LOADS_PS` and `L2_MISS_LOADS_PS`.
  - Add a manual prefetch into L2 with a distance of 32 vector iterations.
    ```
    _mm_prefetch((const char *) &q[m][g][l][k][j][0] + 32*VLEN, _MM_HINT_T1);
    ```
    - Tried a variety of distances and chose the one that worked best.
    - Started with what the compiler inserts with the –qopt-prefetch=3 flag.
  - Swapped to use C VLA syntax to avoid calculating the prefetch offset ourselves.
    - Compiler actually generates more efficient code with VLA syntax.

```c
#pragma omp parallel for
for (int m = 0; m < Nm; m++) {
  for (int g = 0; g < Ng; g++) {
    for (int l = 0; l < Nl; l++) {
      for (int k = 0; k < Nk; k++) {
        for (int j = 0; j < Nj; j++) {
          double total = 0.0;
          _mm_prefetch((const char*) (&q[m][g][l][k][j][0] + 32*VLEN), _MM_HINT_T1);
          #pragma vector nontemporal(r)
          #pragma omp simd reduction(+:total) aligned(a,b,c,x,y,z,r,q:64)
          for (int v = 0; v < VLEN; v++) {
            /* Set r */
            r[m][g][l][k][j][v] =
              q[m][g][l][k][j][v] +
              a[g][v] * x[m][g][k][j][v] +
              b[g][v] * y[m][g][l][j][v] +
              c[g][v] * z[m][g][l][k][v];

            /* Update x, y and z */
            x[m][g][k][j][v] = 0.2*r[m][g][l][k][j][v] - x[m][g][k][j][v];
            y[m][g][l][j][v] = 0.2*r[m][g][l][k][j][v] - y[m][g][l][j][v];
            z[m][g][l][k][v] = 0.2*r[m][g][l][k][j][v] - z[m][g][l][k][v];

            /* Reduce over Ni */
            total += r[m][g][l][k][j][v];

          } /* VLEN */

          sum[m][l][k][j] += total;

        } /* Nj */
      } /* Nk */
    } /* Nl */
  } /* Ng */
} /* Nm */
```

Results

Version	Bandwidth (GB/s)	Total time (s) ntimes = 1000	Improvement
Baseline	78.4	9.23	-
Non-temporal	236.5	2.79	3.3X
Cache blocking	318.9	2.22	4.2X (1.3X over prev)
Prefetching	345.0	2.01	4.6X (1.1X over prev)

Rows include optimisations from preceding rows.

Conclusions and Insights

- Make sure the right vector instructions are being issued.
  - Pay close attention to alignment and streaming stores.
- Examine cache behaviour.
  - See if it's possible to fit data in cache.
  - Back of the envelope calculations help!
- Compare estimated bandwidth to measured bandwidth from Intel® vTune™ Amplifier XE.
  - High estimate and low measurement - better cache behaviour than expected.
  - Low estimate and high measurement - worse cache behaviour than expected.
  - Mega-stream was the second of these.
- Hopefully these optimisations will carry forward into the SNAP mini-app and improve performance there.
- Remaining challenge is to avoid the software prefetch step.
  - Can the prefetchers be improved?
- A big surprise was the change in cache behaviour:
  - Baseline 0.3 version had good L2 cache reuse, but poor L1.
  - Optimised version highlighted lower L2 cache hit rates.

References

- Mega-stream: https://github.com/UK-MAC/mega-stream
- SNAP: https://github.com/lanl/snap
- GPU SNAP publications:

  - [1] T. Deakin, S. McIntosh-Smith, M. Martineau, and W. Gaudin, “An improved parallelism scheme for deterministic discrete ordinates transport,” *Int. J. High Perform. Comput. Appl.*, Sep. 2016.
  - [2] T. Deakin, S. McIntosh-Smith, and W. Gaudin, “Many-Core Acceleration of a Discrete Ordinates Transport Mini-App at Extreme Scale,” in *High Performance Computing: 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, M. J. Kunkel, P. Balaji, and J. Dongarra, Eds. Cham: Springer International Publishing, 2016, pp. 429–448.