Part I: Introduction
ARM ECOSYSTEM IN HPC

- Arm-based processors are gaining importance in HPC:
  - ASTRA at Sandia National Labs (US) → first large scale system in Top500 in November 2018 (ThunderX2)
  - FUGAKU at RIKEN (Japan) → #1 in Top500, June 2020 (A64FX)
  - European Processor Initiative → roadmap for Arm-based processors for an EU exascale system

- Recently introduced Scalable Vector Extension (SVE) is key for designing high-performance cores.

- Our contribution:
  - Performance evaluation of WaLBerla, MiniFE, NEST and Kallisto on Arm-based server platforms
  - Porting of WaLBerla and MiniFE to including support for SVE
  - Analysis of performance with models or reference figures obtained with x86 architecture
HARDWARE PLATFORMS AND PROGRAMMING ENVIRONMENT

- Four different server architectures:
  - Servers with two Hi1616 processors, part of Huawei cluster at JSC
  - Servers with two ThunderX2 processors, part of SAGE prototype cluster at JSC
  - Server with one A64FX processor, part of Fujitsu’s early access program
  - Servers with two Intel Xeon E5-2660 v3 processors of the Haswell generation

- Binaries were generated with Arm compiler for Linux 20.1, GCC 10.1.0, Clang 10.0.0 and FCC 4.1.0

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Hi1616</th>
<th>ThunderX2</th>
<th>A64FX</th>
<th>Xeon E5-2660 v3</th>
</tr>
</thead>
<tbody>
<tr>
<td>#sockets</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>#cores/socket</td>
<td>32</td>
<td>32</td>
<td>48+4</td>
<td>10</td>
</tr>
<tr>
<td>#threads/core</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>#Flops/cycles core</td>
<td>4</td>
<td>8</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>Clock frequency [GHz]</td>
<td>2.4</td>
<td>2.2</td>
<td>2.2</td>
<td>2.6</td>
</tr>
<tr>
<td>#cores per group</td>
<td>4</td>
<td>1</td>
<td>12+1</td>
<td>1</td>
</tr>
<tr>
<td>L1D size/core [KiByte]</td>
<td>32</td>
<td>32</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>L2D size/group [KiByte]</td>
<td>1024</td>
<td>256</td>
<td>8192</td>
<td>256</td>
</tr>
<tr>
<td>L3D size/socket [MiByte]</td>
<td>32</td>
<td>32</td>
<td>–</td>
<td>25</td>
</tr>
<tr>
<td>Memory technology</td>
<td>DDR4-2133</td>
<td>DDR4-2666</td>
<td>HBM2</td>
<td>DDR4-1866</td>
</tr>
<tr>
<td>Memory data rate/server [GByte/s]</td>
<td>136</td>
<td>341</td>
<td>1024</td>
<td>119</td>
</tr>
</tbody>
</table>
Part II: STREAM
STREAM

Overview

- Standard benchmark to measure sustainable memory bandwidth on a system
- The results were obtained with version 5.10
- Array size set to $10^8$, total memory footprint 2288.8 MiByte
- Scaling data with OMP_NUM_THREADS in steps of 8.
- On A64FX environmental variable XOS_MMM_L_PAGING_POLICY=demand:demand:demand for large pages
STREAM

Results

- For all platforms except the Hi1616, 60 - 70% of the peak performance can be reached
- Actual data exchange is higher due to write-allocate mechanism
- On Arm-based architectures, this can be avoided with the dc0 instruction (currently only available in Fujitsu compiler: `-kzfill` or `-knozfill`)

<table>
<thead>
<tr>
<th>Platform</th>
<th>Bandwidth $b_{(Triad)}^{(mem)}$</th>
<th>Rel. to peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hi1616</td>
<td>60 GByte/s</td>
<td>44 %</td>
</tr>
<tr>
<td>ThunderX2</td>
<td>217 GByte/s</td>
<td>64 %</td>
</tr>
<tr>
<td>A64FX, nozfill</td>
<td>632 GByte/s</td>
<td>62 %</td>
</tr>
<tr>
<td>A64FX, zfill</td>
<td>820 GByte/s</td>
<td>80 %</td>
</tr>
<tr>
<td>Xeon E5-2660 v3</td>
<td>84 GByte/s</td>
<td>70 %</td>
</tr>
</tbody>
</table>
Part III: WaLBerla
Lattice Boltzmann Method follows an iterative update scheme
Each iteration two steps:
  - collision (arithmetic operations)
  - propagate (only memory transfer)
Performance depends only on memory performance
Uniform Grid Benchmark (D3Q19 model)

Memory transfer:
\[ I_{\text{mem}}^{(D3Q19)} = V \cdot 2 \cdot 38 \cdot 8 \text{ Byte} = V \cdot 608 \text{ Byte} \]

Performance model for lattice site update rate:
\[ b^{(D3Q19)} \lesssim \frac{V \cdot b_{\text{mem}}}{I_{\text{mem}}^{(D3Q19)}(V)} \]
WALBERLA
Porting strategies

- Written in C++14, no specific changes were needed for porting to Arm architecture
- Support for SIMD instructions is realised with compiler auto-vectorisation facilitated by proper data layout
- Specific compiler directives had to be used:
  - `clang loop vectorize(assume_safety)` for Clang-based Arm compiler
  - `GCC ivdep` for GCC

Table 2: Fraction of SVE instructions with (without) compiler directives.

<table>
<thead>
<tr>
<th>SVE length</th>
<th>GCC (%)</th>
<th>Arm Compiler for Linux (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>80 (10)</td>
<td>80 (40)</td>
</tr>
<tr>
<td>256</td>
<td>69 (5.4)</td>
<td>70 (25)</td>
</tr>
<tr>
<td>512</td>
<td>54 (2.8)</td>
<td>55 (15)</td>
</tr>
</tbody>
</table>
For all servers except Intel Xeon E5, the performance is close to model prediction.

- Good scaling behaviour for A64FX using Arm compiler (OMP_PROCE_BIND=close).
- Arm HPC Compiler generates slightly faster code than GCC (14%, 4% and 9% on Hi1616, ThunderX2 and A64FX respectively).

### Table 3: WaLBerla performance results.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Observed [MUPS]</th>
<th>Expected [MUPS]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hi1616</td>
<td>92</td>
<td>98</td>
</tr>
<tr>
<td>ThunderX2</td>
<td>317</td>
<td>356</td>
</tr>
<tr>
<td>A64FX</td>
<td>1044</td>
<td>1041</td>
</tr>
<tr>
<td>Xeon E5-2660 v3</td>
<td>222</td>
<td>138</td>
</tr>
</tbody>
</table>

**Figure 2:** Performance of WaLBerla on A64FX (MUPS)
Part IV: MiniFE
MiniFE is a mini-application for exploration of different parallel programming models for applications based on finite element methods.

- It implements an iterative conjugent gradient solver.
- Performance is dominated by the implementation of a sparse Matrix-Vector Multiplication (spMVM).

An information exchange analysis gives:

\[ \begin{align*}
I_{\text{mem}}^{(\text{MATVEC})} &= (N_{\text{row}} \cdot 4 + N_{\text{nz}} \cdot (4 + 8)) \cdot (N_{\text{iter}} + 1) \\
\end{align*} \]

- We set \( N_{\text{iter}} = 500 \) and \( V = 256^3 \)
Specific care is required for obtaining good performance in the presence of wide SIMD units (A64FX)

- CSR format for storing sparse Matrix is inefficient due to its row-wise data layout.
- Implementation of sliced ELLPACK format with vector-length-agnostic approach
MINIFE
Sparse Matrix-Vector multiplication

int sve_length = svcntd();
svbool_t pg = svptrue_b64();
for(int slice_id=0; slice_id < num_slices; slice_id++) {
    svfloat64_t sum = svdup_f64(0.0);
    for(int i = offsets[slice_id]; i < offsets[slice_id + 1]; i += sve_length){
        svfloat64_t acofs = svld1(pg, &Acoefs[i]);
        svuint64_t indices = svld1sw_u64(pg, &Acols[i]);
        svfloat64_t xcofs = svld1_gather_index(pg, &xcoefs[0], indices);
        sum = svmla_z(pg, sum, acofs, xcofs);
    }
    svst1(pg, &ycoefs[slice_id * sve_length], sum);
}

Listing 1: SVE-enabled part of the spMVM kernel of MiniFE
MINIFE

Results

- On all platforms we obtain 75 - 90% of peak performance
- On A64FX sliced ELLpack shows significantly better performance than CSR (only when explicitly vectorized)

Table 4: MiniFE mat-vec execution times.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Observed [s]</th>
<th>Expected [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hi1616</td>
<td>40.9</td>
<td>34.9</td>
</tr>
<tr>
<td>ThunderX2</td>
<td>14.2</td>
<td>9.7</td>
</tr>
<tr>
<td>A64FX</td>
<td>3.7</td>
<td>3.3</td>
</tr>
<tr>
<td>Xeon E5-2660 v3</td>
<td>27.1</td>
<td>25.0</td>
</tr>
</tbody>
</table>

Figure 4: Comparison of execution time for different sparse matrix formats used in MiniFE on A64FX
Part V: NEST
NEST
Overview

- NEST is a simulator for spiking neural network models
- Simulation is performed in two steps:
  1. Network is built by randomly connecting chosen number of neurons
  2. Simulation if performed (time dominating phase)
- `hpc_benchmark.sli` is used, with 22500 neurons and $2.5 \times 10^8$ connections
- Hard to develop models to predict performance. (complex control flow, random memory accesses and rather few computations)
NEST

Results

- ThunderX2 and Xeon benefit from multiple threads per core
- Scaling efficiency: $\epsilon = \frac{\Delta t(2)}{\Delta t(N_{\text{core}})} \frac{2}{N_{\text{core}}}$
- Main reason for poor A64FX performance are backend stalls

Table 5: NEST build/simulation time and scaling efficiency

<table>
<thead>
<tr>
<th>Platform</th>
<th>Build [s]</th>
<th>Simulation [s] / P (best)</th>
<th>$\epsilon$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hi1616</td>
<td>2.41</td>
<td>0.93 / 64</td>
<td>1.4</td>
</tr>
<tr>
<td>ThunderX2</td>
<td>1.59</td>
<td>0.59 / 128</td>
<td>0.9</td>
</tr>
<tr>
<td>A64FX</td>
<td>4.41</td>
<td>1.53 / 48</td>
<td>1.5</td>
</tr>
<tr>
<td>Xeon E5-2660 v3</td>
<td>8.7</td>
<td>1.23 / 40</td>
<td>1.2</td>
</tr>
</tbody>
</table>

Figure 5: NEST simulation time

Figure 6: NEST: Proportion of cycles spent stalling vs. other cycles on A64FX and ThunderX2.
Part VI: Kallisto
KALLISTO

Overview

- Kallisto foresees two phases for processing Ribonucleic acid (RNA) sequencing transcripts:
  - 1. Index table is constructed by building a de Bruijn graph from all sequences of nucleotides of length $k$ that are found in the transcriptome.
  - 2. Sequencing data is read and decomposed into its k-mers. Then a corresponding path in the de Bruijn graph is found with the help of the index table.
- Execution time is typically limited by performance and efficiency of at which data is read
- Modified version of Kallisto is used that uses memory-mapped I/O for reading data in parallel
- For benchmarking purposes, all data was preloaded into memory using /dev/shm
- Used gencode hg38 v27, input file contains $34.5 \times 10^6$ RNA-seq reads
KALLISTO

Results

- Time spent in I/O is similar for all platforms
- The difference in single-thread execution time is mainly due to an increased number of stall cycles
- Increasing number of threads reduces such latency differences

Table 6: Kallisto execution times without and with I/O and with one or more number of threads $P$ using GCC.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Time [s] (P = 1)</th>
<th>Time [s] / P (best)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hi1616</td>
<td>291</td>
<td>31 / 32</td>
</tr>
<tr>
<td>ThunderX2</td>
<td>246</td>
<td>10 / 32</td>
</tr>
<tr>
<td>A64FX</td>
<td>374</td>
<td>13 / 48</td>
</tr>
<tr>
<td>Xeon E5-2660 v3</td>
<td>218</td>
<td>20 / 32</td>
</tr>
</tbody>
</table>

Table 7: Kallisto execution times with one or more number of threads $P$ using the Arm Compiler.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Time [s] (P = 1)</th>
<th>Time [s] / P (best)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hi1616</td>
<td>291</td>
<td>27 / 32</td>
</tr>
<tr>
<td>ThunderX2</td>
<td>243</td>
<td>10 / 32</td>
</tr>
<tr>
<td>A64FX</td>
<td>401</td>
<td>15 / 48</td>
</tr>
</tbody>
</table>
Part VII: Conclusion
CONCLUSION

- Observed results of applications with significantly different characteristics show increasing performance of current Arm-based server platforms.
- Most notable improvements due to increased memory bandwidth and availability of SVE which can be fully exploited for memory-bound applications (Walberla and MiniFE).
- We currently obtained less good performance on A64FX for applications with complex control flow (Nest and Kallisto).
  - Optimisation for latency hiding are necessary for improving current results.
- Overall, Arm HPC compiler performs slightly better than GCC when compiling Walberla and NEST.
We would like to thank:
- HUAWEI for making the JUAWEI cluster hardware available at JSC
- Fujitsu for allowing us to participate in the A64FX early access program
- M. Becker and J.L. Schultze (University of Bonn) for a sequencing data set

Funding for parts of this work is received from:
- the European Commission H2020 program under Grant Agreement 779877 (Mont-Blanc 2020)
- Helmholtz-Gemeinschaft e.V. under the grant ZT-I-0007 (Sparse2Big)