Building Blocks for Sparse Linear Algebra on Heterogeneous Hardware

Moritz Kreutzer, Prof. Gerhard Wellein, Dr. Georg Hager

Workshop on Programming of Heterogeneous Systems in Physics (PHSP14)
Jena, 07/15/2014
Selective Challenges

We are facing a list of challenges which we have to deal with:

1. Increasingly heterogeneous hardware
   - Well-known x86 CPUs are working together with accelerators/co-processors
   - Inherently different programming paradigms
   - Few transparently heterogeneous libraries
Selective Challenges

We are facing a list of challenges which we have to deal with:

2. Increasing level of hardware parallelism
   - Higher hardware performance only due to more parallelism
   - Application may have limited scalability with standard approaches (e.g., data parallelism)
   - Novel levels of parallelism (e.g., task parallelism) may be cumbersome to implement by application developers in an efficient way
Selective Challenges

We are facing a list of challenges which we have to deal with:

3. Vulnerability for hardware faults
   - Mean time between failures is predicted to decrease to a critical level on exascale systems
   - No stressable numbers on this topic (naturally...) but it is good to be prepared
Selective Challenges

We are facing a list of challenges which we have to deal with:

4. Library performance is often limited due to generality
   - Application knowledge is a key to high library performance
     - E.g., we can fuse kernels instead of calling them sequentially
   - Established libraries may not perform well in specific cases
     - Prominent example: Calling GEMM with tall skinny matrices may deliver poor performance even for highly-optimized BLAS libs
Ideas

- Heterogeneity in hardware architectures
  - Concurrent use of CPUs and accelerators for efficient execution

- Limited scalability with standard approaches
  - Reveal new levels of parallelism beyond data parallelism

- Future large scale systems may be prone to hardware faults
  - Utilize low-overhead fault tolerance mechanisms
  - Asynchronous checkpointing comes within “new levels of parallelism”

- Limited library performance due to generality
  - Tailor performance-sensitive parts towards the application
Contribution

A library which delivers highly efficient building blocks for sparse linear algebra ("General, Hybrid and Optimized Sparse Toolkit")

- Several levels of parallelism: MPI, OpenMP, CUDA, SIMD
- Transparent use of heterogeneous hardware
- Generic interface for hardware-affine task-level parallelism
- Highly-optimized low-level kernels (partly generated at compilation)
- Liberal open source release (beta) planned for Q4/2014

Work supported by DFG through Priority Programme 1648 "Software for Exascale Computing" (SPPEXA) under project ESSEX ("Equipping Sparse Solvers for Exascale")
Ideas

- Heterogeneity in hardware architectures
  - Concurrent use of CPUs and accelerators for efficient execution

- Limited scalability with standard approaches
  - Reveal new levels of parallelism beyond work-sharing

- Future large scale systems may be prone to hardware faults
  - Utilize low-overhead fault tolerance mechanisms

- Limited library performance due to generality
  - Tailor performance-sensitive parts towards the application
HETEROGENEOUS SPARSE MATRIX-VECTOR MULTIPLICATION

Gaining performance and interface simplicity with a unified storage format
Sparse Matrix-Vector Multiplication (SpMVM)

- Key ingredient in many matrix diagonalization algorithms and iterative solvers
  - Lanczos, Davidson, Jacobi-Davidson, CG, ...

- Inevitably memory-bound for large problems

- Easily parallelizable in shared and distributed memory

- Data storage format is crucial for performance properties
  - Default general format on CPUs: Compressed Row Storage (CRS)
  - Depending on compute architecture
Sparse Matrix Format Jungle
SpMVM in the Heterogeneous Era

- Compute clusters are getting more and more heterogeneous

- A special format per compute architecture
  1. hampers runtime exchange of matrix data
  2. complicates library interfaces

- CRS (CPU standard format) may be problematic (cf. next slides)
  - Vectorization along matrix rows
  - Bad utilization for short rows and wide SIMD units (Intel MIC: 512 bit)

→ We want to have a unified, SIMD-friendly, and high-performance sparse matrix storage format.
Compressed Row Storage (CRS)

- Standard format for CPUs
- Entries and column indices stored row-wise
CRS Vectorization

```c
unsigned int i, j;
double tmp;

#pragma omp parallel for schedule(runtime) private (tmp1, tmp2, j)
for (i=0; i<nrows; i++){
    tmp1 = 0.0;
    tmp2 = 0.0;
    for (j=rpt[i]; j<rpt[i+1]; j=j+2){
        tmp1 += val[j] * rhs[col[j]];
        tmp2 += val[j+1] * rhs[col[j+1]];
    }
    lhs[i] += tmp1+tmp2;
}
```

- Potential problem: **512 bit vector registers on Xeon Phi**
  - 8 doubles or 16 integers in a single vector
  - j-loop:16-way unrolling ➔ problem for short rows
  - horizontal add (reduction) gets more costly
Sliced ELLPACK

- Well-known sparse matrix format for GPUs

- Entries and column indices stored column-wise in chunks
- One parameter:
  1. $C$: Chunk height
Minimizing the storage overhead $\rightarrow$ SELL-C-\(\sigma\)

- Sort rows within a range \(\sigma\) to minimize the overhead
  - \(\sigma\) should not be too large in order to not worsen the RHS vector access pattern

- Two parameters:
  1. \(C\): Chunk height
  2. \(\sigma\): Sorting scope
Choosing the Sorting Scope $\sigma$

- The larger the sorting scope, the lower the storage overhead
- But what happens if the sorting scope gets too large?

\[ \beta \ldots \text{Chunk occupancy (large beta - low overhead)} \]
\[ \alpha \ldots \text{RHS vector multiple load overhead} \]
SELL-C-σ Performance

Using a unified storage format comes with little performance penalty in the worst case and up to a 3x performance gain in the best case for a wide range of test matrices.
Ideas

- Heterogeneity in hardware architectures
  - Concurrent use of CPUs and accelerators for efficient execution

- Limited scalability with standard approaches
  - Reveal new levels of parallelism beyond work-sharing

- Future large scale systems may be prone to hardware faults
  - Utilize low-overhead fault tolerance mechanisms

- Limited library performance due to generality
  - Tailor performance-sensitive parts towards the application
TRANSPARENT USE OF HETEROGENEOUS COMPUTE NODES

Ideas and implementation
Heterogeneous Architectures in a Single Node

- Different programming paradigms
  - CPU: only native mode
  - GPU: only accelerator mode
  - Xeon Phi: accelerator or native mode

- Different performance
  - Sensible work distribution

- Different architectures (obviously...)

PU = processing unit
Node Partitioning

- Distinction between architectures via MPI processes:
  - exactly one process per GPU
  - at least one process per Xeon Phi
  - at least one process per (multi-core) CPU

- Each process gets assigned a weight deciding the share of work which depends on their relative performance

- Resource management:
  - Each process running inside an exclusive CPU set (no shared cores)
  - CPU sets may span several NUMA nodes
Example Node Partitioning

- Minimal amount of MPI processes on this node: 3
- GPU is managed by a full core on the nearest socket
- CPU process spans two NUMA nodes
- Xeon Phi operated in native mode
  - one MPI process running on the coprocessor
Performance of heterogeneous SpMV

$ ./spmv_bench

8.13 Gflop/s

$ GHOST_TYPE=CUDA ./spmv_bench

21.8 Gflop/s

if (type == GHOST_TYPE_CUDA) {weight = 2.6;} else {weight = 1.0;}

$ mpirun -np 2 ./spmv_bench

27.5 Gflop/s

s/GHOST_SPARSEMAT_TRAITS_DEFAULT/GHOST_SPARSEMAT_TRAITS_SCOTCHIFY

$ mpirun -np 2 ./spmv_bench

28.8 Gflop/s
Ideas

- Heterogeneity in hardware architectures
  - Concurrent use of CPUs and accelerators for efficient execution

- Limited scalability with standard approaches
  - Reveal new levels of parallelism beyond work-sharing

- Future large scale systems may be prone to hardware faults
  - Utilize low-overhead fault tolerance mechanisms

- Limited library performance due to generality
  - Tailor performance-sensitive parts towards the application
APPLYING THE KERNEL POLYNOMIAL METHOD ON A MULTICORE CPU

A small case study
Stages of Optimization

1. Baseline implementation

\[
\text{for } r = 0 \text{ to } r < R \text{ do} \\
|v\rangle = |\text{rand()}\rangle \\
\text{Initialization steps and computation of } \mu_0, \mu_1 \\
\text{for } m = 1 \text{ to } m < M/2 \text{ do} \\
\quad \text{switch}(|w\rangle, |v\rangle); \\
\quad |u\rangle = H|v\rangle; \\
\quad |u\rangle = |u\rangle - b|v\rangle; \\
\quad |w\rangle = -|w\rangle; \\
\quad |w\rangle = |w\rangle + 2a|u\rangle; \\
\quad \mu_{2m} = \langle v|v\rangle; \\
\quad \mu_{2m+1} = \langle w|v\rangle; \\
\quad \text{end} \\
\text{end}
\]
Stages of Optimization

1. Baseline implementation
2. Augmented SpMVM
   - The scaling, shift and computation of dot products can be integrated into the SpMVM kernel

```plaintext
for r = 0 to r < R do
    |v⟩ = |rand()⟩
    |w⟩ = a(H - b)|v⟩ & μ0 = ⟨v|v⟩ & μ1 = ⟨w|v⟩;
for m = 1 to m < M/2 do
    switch(|w⟩, |v⟩);
    |w⟩ = 2a(H - b)|v⟩ − |w⟩ & μ2m = ⟨v|v⟩ & μ2m+1 ⟨w|v⟩;
end
end
```
Stages of Optimization

1. Baseline implementation
2. Augmented SpMVM
   - The scaling, shift and computation of dot products can be integrated into the SpMVM kernel

![Bar chart showing Gflop/s on one socket of Intel Ivy Bridge, SELL-C-σ storage format]

- Stage 1: 5.5 Gflop/s
- Stage 2: 8.3 Gflop/s
Stages of Optimization

1. Baseline implementation
2. Augmented SpMVM
3. Using interleaved vector blocks
   - The matrix has to be loaded for each outer loop iteration
   ➔ Apply the augmented SpMVM to a block of random vectors at once

Gflop/s on one socket of Intel Ivy Bridge, SELL-C-σ storage format

<table>
<thead>
<tr>
<th>Stage</th>
<th>Gflop/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.5</td>
</tr>
<tr>
<td>2</td>
<td>8.3</td>
</tr>
<tr>
<td>3</td>
<td>21.6</td>
</tr>
</tbody>
</table>
Ideas

- Heterogeneity in hardware architectures
  - Concurrent use of CPUs and accelerators for efficient execution

- Limited scalability with standard approaches
  - Reveal new levels of parallelism beyond work-sharing

- Future large scale systems may be prone to hardware faults
  - Utilize low-overhead fault tolerance mechanisms

- Limited library performance due to generality
  - Tailor performance-sensitive parts towards the application
A SIMPLE INTERFACE FOR TASK PARALLELISM

Allowing easy access to new levels of parallelism
Implementation Details: Task Processing

- A task is defined by the user as
  1. A function callback along with parameters (required)
  2. The number of PUs to process the task (required)
  3. The preferred NUMA node to process the task (optional)
  4. A list of tasks on whose completion this task depends (optional)
  5. A combination of flags (optional)
     - PRIO_HIGH: Put task to beginning of queue.
     - NODE STRICT: Execute task only on the given NUMA node.
     - NOT_ALLOW CHILD: Do not allow a child task to use the task‘s resources.
     - NOT_PIN: Do not register the task in the PU map.

- All tasks line up in a single queue
Implementation Details: Task Processing

- OpenMP can be used straight-forwardly in a task
- The library cares for hardware affinity and the prevention of resource conflicts
- Task control functions (selection):
  1. `ghost_task_enqueue()`
  2. `ghost_task_wait()`
  3. `ghost_task_waitany()`
  4. `ghost_task_test()`
Implementation Details: Resource Management

- Each process stores idle/busy states and locality information of each of its PUs (e.g. for initial state of CPU process)

<table>
<thead>
<tr>
<th>PU</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMA node</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Idle</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

- One shepherd thread will be created per PU:

  - `pthread_create()`
  - `pthread_cond_wait()`

- The shepherd threads wait for tasks to be put in the task queue
- More shepherd threads will be spawned on necessity
Task Lifetime

- After a suitable task has been found by the shepherd thread:

```c
omp_set_num_threads(task->npu);
#pragma omp parallel
{
    ghost_thread_pin(...);
    ghost_pumap_setbusy(...);
}

task->ret = task->func(task->arg); // execute task
// OpenMP parallel regions are pinned correctly in the task
ghost_pumap_setidle(task->cpuset);
pthread_cond_wait(); // wait for new tasks
```

- Physical OpenMP threads (and pinning) are persistent between successive parallel regions in the shepherd thread (luckily)