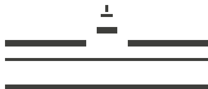


Data Parallel Algorithmic Skeletons with Accelerator Support

living.knowledge
WWU Münster

Steffen Ernsting
and Herbert Kuchen

July 2, 2015

Agenda

- Hardware accelerators
  - GPU and Xeon Phi
- Skeleton implementation (C++ vs Java)
  - Parallelization
  - Providing the user function
  - Adding additional arguments
- Performance comparison
  - 4 benchmark applications: Matrix mult., N-Body, Shortest paths, Ray tracing

living.knowledge
WWU Münster

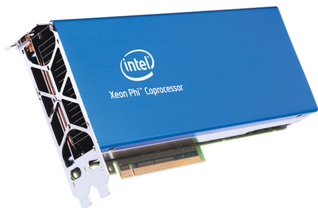
Hardware Accelerators

Graphics Processing Units

- K20x: 2688 CUDA cores
- C++ + CUDA/OpenCL
- Offload compute-intensive kernels

Intel Xeon Phi

- ~60 x86 cores (240 threads)
- C++ + pragmas (+ SIMD intrinsics)
- Offload and native programming models
- Intel: "Recompile and run"



Accelerator-Support in Java

- Various projects aim to add GPU support to Java
- Bindings vs. JIT (byte)code translation
  - Bindings: e.g. JCUDA, JOCL, JavaCL
  - Code translation: e.g. Aparapi, Rootbeer, Java-GPU
  - Java 9 ...?
- Why choose accelerated Java over accelerated C++?
  - Write and compile once, run everywhere
  - Huge JDK: lots of little helpers
  - Automated memory management

living.knowledge
WWU Münster

Aparapi

- Created by Gary Frost (AMD), released to open source
- JIT-compilation: Java bytecode → OpenCL
- Execution modes: GPU, JTP (Java Thread Pool), SEQ, PHI (experimental)
- Offers good programmability
- Restriction: primitive types only

Java code:

```
1 int[] a = {1, 2, 3, 4};
2 int[] b = {5, 6, 7, 8};
3 int[] c = new int[a.length];
4
5 for (int i=0; i<c.length; i++) {
6   c[i]=a[i]+b[i];
7 }
```

Aparapi code:

```
1 Kernel k = new Kernel() {
2   public void run(){
3     int i=getGlobalId();
4     c[i]=a[i]+b[i];
5   }
6 };
7 k.execute(Range.create(c.length));
```

living.knowledge WWU Münster

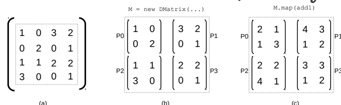
The **Mue**nster **S**keleton **Li**brary

- C++ skeleton library
- Target architectures: multi-core clusters possibly equipped with accelerators such as GPU and Xeon Phi
- Parallelization: MPI, OpenMP, and CUDA
- Data parallel skeletons (with accelerator support)
  - map, zip, fold + variants
- Task parallel skeletons
  - pipe, farm, D&C, B&B

living.knowledge
WWU Münster

Parallelization

- Inter-node parallelization (MPI):
  - Distributed data structures (DArray<T>, DMatrix<T>)



- Intra-node parallelization:
  - C++: OpenMP, CUDA
  - Java: Aparapi

```
1  T fold(FoldFunction f) {
2      // OpenMP, CUDA, Aparapi
3      T localResult = localFold(f, localPartition);
4
5      // MPI
6      T[] localResults = new T[numProcs];
7      allgather(localResult, localResults);
8
9      return localFold(f, localResults);
10 }
```

living.knowledge
WWU Münster

The User Function (C++)

- Pass function as skeleton argument
- **But:** (host) function pointers cannot be used in device code

⇒ Use functors instead:

```
1 template <typename IN, typename OUT>
2 class MapFunctor : public FunctorBase
3 {
4 public:
5   // To be implemented by the user.
6   MSL_UFCT virtual OUT operator() (IN value) const = 0;
7
8   virtual ~MapFunctor();
9 };
```

living.knowledge WWU Münster

The User Function (Java)

- No function pointers in Java
- Analogous to C++: use functors

```
1  public abstract class MapKernel extends Kernel {
2      protected final int[] in, out;
3
4      // To be implemented by the user.
5      public abstract int mapFunction(int value);
6
7      public void run() {
8          int gid = getGlobalId();
9          out[gid] = mapFunction(in[gid]);
10     }
11
12     // Called by skeleton implementation.
13     public void init(DIArray in, DIArray out) {
14       this.in = in.getLocalPartition();
15       this.out = out.getLocalPartition();
16     }
17 }
```

living.knowledge
WWU Münster

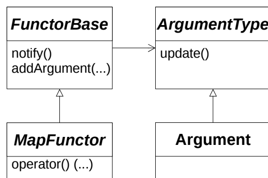
Additional Arguments (C++)

- Arguments to the user function are determined by the skeleton

⇒ Add additional arguments as data members

- But: Works for primitives and PODs, but not for classes with pointer data members
- Pointer data + multi-GPU: need 1 pointer for each GPU

⇒ Solution: Observer pattern



living.knowledge WWU Münster

Additional Arguments (Java)

- Add additional arguments as data members
- Aparapi handles memory allocation and data transfer to GPU memory

```
1  public class AddN extends MapKernel {
2      protected int n;
3
4      public AddN(int n) {
5          this.n = n;
6      }
7
8      public int mapFunction(int value) {
9          return value+n;
10     }
11 }
12 DIArray A = new DIArray(...);
13 A.map(new AddN(2));
```

- Limited to (arrays of) primitive types

living.knowledge
WWU Münster

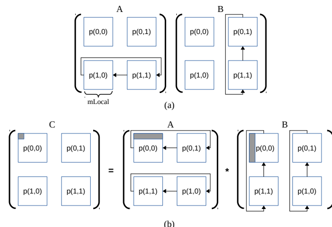
Benchmarks

- **4 benchmark applications:** Matrix mult., N-Body, Shortest paths, Ray tracing
- **2 test systems:**
  1. **GPU cluster:** 2 Xeon E5-2450 (16 cores) + 2 K20x GPUs per node
  2. **Xeon Phi system** with 8 Xeon Phi 5110p coprocessors
- **6 configurations:**
  - **2** × **CPU:** *C++ CPU*, *Java CPU*
  - **3** × **GPU:** *C++ GPU*, *C++ multi-GPU*, *Java GPU*
  - **1** × **Xeon Phi:** *C++ Xeon Phi*

living.knowledge
WWU Münster

Case Study: Matrix Multiplication

- Cannon's algorithm for square matrix multiplication
- Checkerboard block decomposition (2D Torus)



(a) Initial shifting of A and B

(b) Submatrix multiplication + stepwise shifting

living.knowledge WWU Münster

Matrix Multiplication

Main algorithm:

```
1  template <typename T>
2  DMatrix<T>& matmult(DMatrix<T>& A, DMatrix<T>& B, DMatrix<T>* C) {
3    // Initial shifting.
4    A.rotateRows(&negate);
5    B.rotateCols(&negate);
6
7    for (int i = 0; i < A.getBlocksInRow(); i++) {
8      DotProduct<T> dp(A, B);
9      // Submatrix multiplication.
10     C->mapIndexInPlace(dp);
11
12     // Stepwise shifting.
13     A.rotateRows(-1);
14     B.rotateCols(-1);
15   }
16   return *C;
17 }
```

living.knowledge
WWU Münster

Matrix Multiplication

Map functor (C++):

```
1  template <typename T>
2  struct DotProduct : public MapIndexFunctor<T, T> {
3    LMatrix<T> A, B;
4
5    dotproduct(DMatrix<T>& A_, DMatrix<T>& B_)
6      : A(A_), B(B_)
7    {
8    }
9
10   MSL_UFCT T operator()(int row, int col, T Cij) const
11   {
12     T sum = Cij;
13     for (int k = 0; k < this->mLocal; k++) {
14       sum += A[row][k] * B[k][col];
15     }
16     return sum;
17   }
18 };
```

living.knowledge
WWU Münster

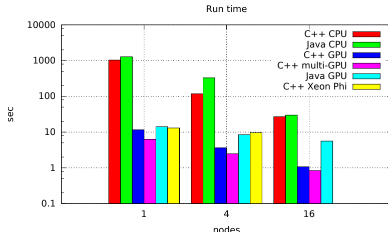
Matrix Multiplication

Map functor (Java):

```
1  class DotProduct extends MapIndexInPlaceKernel {
2      protected float[] A, B;
3
4      public Dotproduct(DFMatrix A, DFMatrix B) {
5          super();
6          this.A = A.getLocalPartition();
7          this.B = B.getLocalPartition();
8      }
9
10     public float mapIndexFunction(int row, int col, float Cij) {
11         float sum = Cij;
12         for (int k = 0; k < mLocal; k++) {
13             sum += A[row * mLocal + k] * B[k * mLocal + col];
14         }
15         return sum;
16     }
17 }
```

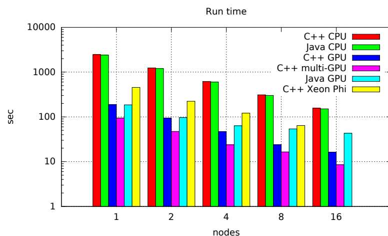
living.knowledge
WWU Münster

Results: Matrix Multiplication



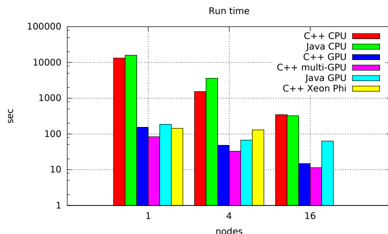
- *C++ (multi-)GPU* performs best (30×-160× speedup vs. CPU)
- *Java GPU* and *C++ Xeon Phi* on a similar level
- *C++ CPU* and *Java CPU* on a similar level
  - Superlinear speedups due to cache effects

Results: N-Body



- *C++ (multi-)GPU* performs best (10×-13× speedup vs. CPU)
- *C++ GPU* delivers better scalability than *Java GPU* on higher node counts
- CPU versions on same level
- *C++ Xeon Phi* perf. between CPU and GPU perf.

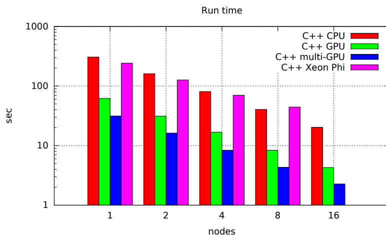
Results: Shortest Paths



- *C++ (multi-)GPU* performs best (20×-160× speedup vs. CPU)
- *Java GPU* and *C++ Xeon Phi* on a similar level
- *C++ CPU* and *Java CPU* on a similar level

living.knowledge WWU Münster

Results: Ray tracing



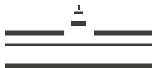
- *C++ (multi-)GPU* performs best (5×-10× speedup vs. CPU)
- Xeon Phi performance only close to CPU performance
  - No auto-vectorization

living.knowledge
WWU Münster

Conclusion & Future Work

- C++ and Java version offer comparable performance
- Still: restrictions on Java side
  - (Arrays of) primitive types
  - No multi-GPU support (yet)
  - Will be addressed in future work
- Xeon Phi can be considered in-between CPUs and GPUs
  - Performance-wise and in terms of programmability

living.knowledge
WWU Münster



Thank you for your attention!

Questions?

living.knowledge
WWU Münster