TDDD56 Multicore Computing /
TDDC78 Programming of Parallel Computers

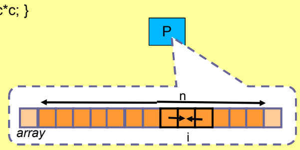
Prof. Dr. Christoph Kessler, IDA, Linköping university

Parallelization Patterns and Algorithmic Skeletons

An Introduction

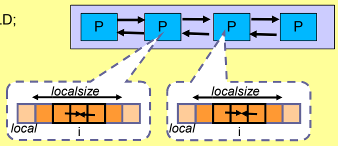
Example 1: 1D smoothening filter in C

```
…
float filter (float a, b, c) { return wa*a + wb*b + wc*c; }
…
void main ( int argc, char *argv[] )
{
  float *array = new_FloatArray( n+2 );
  float *tmp = new_FloatArray( n+2 );
  …
  while ( globalerr > 0.1 ) {
     for (i=1; i<=n; i++)
        tmp[i] = filter( array[i-1], array[i], array[i+1] );
     globalerr = 0.0;
     for (i=1; i<=n; i++)
        globalerr = fmax (globalerr, fabs(array[i] – tmp[i]) );
     for (i=1; i<=n; i++)
        array[i] = tmp[i];
  }
}
```

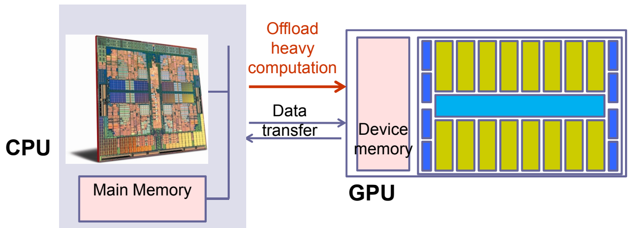


Example 1: 1D smoothening filter in C + MPI

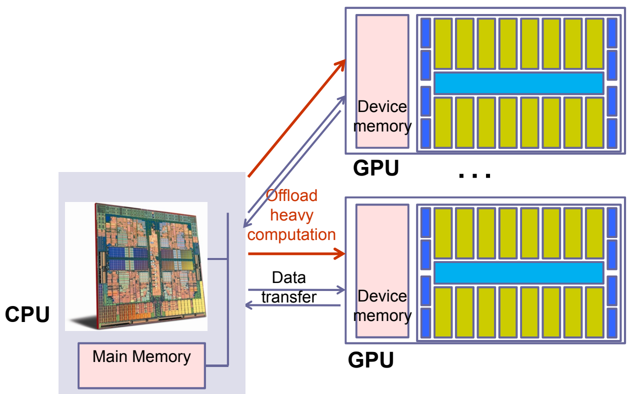
```
…
void main ( int argc, char *argv[] ) {
   MPI_Comm com = MPI_COMM_WORLD;
   MPI_Init ( &argc, &argv );
   MPI_Comm_size ( com, &np );
   MPI_Comm_rank ( com, &me );
   …
   localsize = (int) ceil ( (float) n / np );
   local = new_FloatArray( localsize + 2);
   …
   while ( globalerr > 0.1 ) {
      if (me>0)       MPI_Send ( local+1,  1, MPI_FLOAT,  left_neighbor, 10,  com );
      if (me<np-1)  MPI_Send ( last, 1, MPI_FLOAT,  right_neighbor,  20,  com );
      for (i=1; i<=localsize; i++)
         tmp[i] = filter( local[i-1], local[i], local[i+1] );
      if (me<np-1)  MPI_Recv ( tmp,  1,  MPI_FLOAT,  right_neighbor,  10,  com,  … );
      if (me>0)       MPI_Recv ( tmp+localsize+1, 1, MPI_FLOAT,  left_neighbor,  20,  com,  … );
      tmp[1] = filter( local[0], local[1], local[2] );
      tmp[localsize] = filter( local[localsize-1], local[localsize], local[localsize+1] );
      localerr = 0.0;
      for (i=1; i<=localsize; i++)  localerr = fmax( localerr, fabs ( local[i]-tmp[i] ));
      MPI_Allreduce ( &localerr, &globalerr,  1,  MPI_FLOAT,  MPI_MAX,  com );
      for (i=1; i<localsize; i++)
         local[i] = tmp[i];
   }   …
```



Example 2: GPU-Based Systems

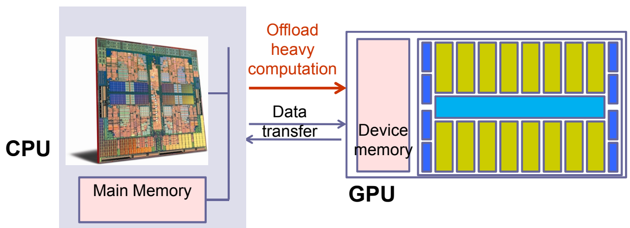


Example 2: GPU-Based Systems



Example 2 Programming of GPU-Based Systems …

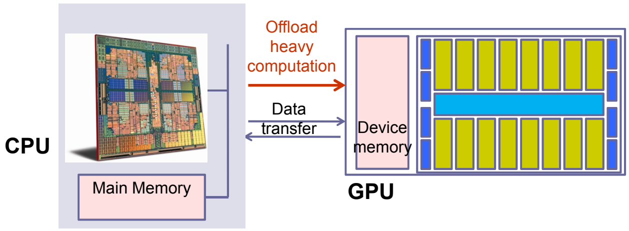
- Portability ?
- Programmability ?
- Performance portability ?



Example 2: Programming of GPU-based Systems … with OpenCL™

OpenCL

- **Portability** ☺
- **Programmability** ☹ (low level)
- **Performance portability** ☹ (requires reoptimization)



Complexity of *Parallel* Algorithms and Programs

- Many different parallel programming models
  - Identify parallelism ("tasks")
  - Synchronization and communication?
  - Memory structure, -consistency model?
  - Resource allocation, mapping, scheduling?
  - Using accelerators, e.g. GPU? Or several GPUs?
- Error prone, hard to debug
- Code portability?
- Performance portability??

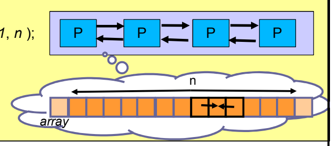
Can we make parallel programming as easy as sequential programming?

Observation

- Same characteristic form of parallelism, communication, synchronization re-applicable for all occurrences of the same specific structure of computation
  (*(parallel) algorithmic paradigm, building block, pattern, …*)
  - Elementwise operations on arrays
  - Reductions
  - Scan (Prefix-op)
  - Divide-and-Conquer
  - Farming independent tasks
  - Pipelining
  - …
- Most of these have both sequential and parallel implementations

Example 1, cont.: 1D smoothening in C + Skeletons

```
…
float filter (float a, b, c) { return wa*a + wb*b + wc*c; }

float elemError ( float a, b ) { return fabs ( a – b ); }
…
void main ( int argc, char *argv[] ) {
   …
   DistrFloatArray *array = new_DistrFloatArray ( n + 2 );
   DistrFloatArray *tmp = new_DistrFloatArray ( n + 2 );
   DistrFloatArray *err = new_DistrFloatArray ( n + 2 );
   …
   while ( globalerr > 0.1 ) {
      map_with_overlap( filter, 1, tmp, array+1, n );
      map( elemError, err, array+1, tmp, n );
      reduce( fmax, &globalerr, err, n );
      map( copy, array+1, tmp, n );
   }
   …
}
```



Data parallelism

Given:

- One or several data containers x with n elements, e.g. array(s) $x = (x_1,...x_n)$, $z = (z_1,…,z_n)$, …
- An operation f on individual elements of x, z, … (e.g. *incr, sqrt, mult*, ...)

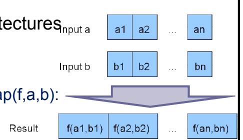
Compute: $y = f(x) = (f(x_1), ..., f(x_n))$

Parallelizability: Each data element defines a task

- Fine grained parallelism
- Portionable, fits very well on all parallel architectures

Notation with higher-order function*:*

- y = ***map*** (f, x)



Variant: map with overlap: $y_i = f(x_{i-k}, …, x_{i+k})$, $i = 0,…,n-1$ for some small constant k

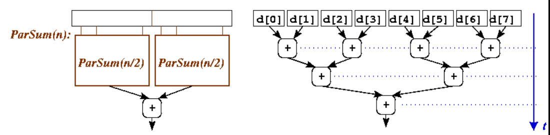
Data-parallel Reduction

Given:

- A data container x with n elements, e.g. array $x = (x_1,...x_n)$
- A binary, associative operation *op* on individual elements of x (e.g. *add, max, bitwise-or*, ...)

Compute: $y = OP_{i=1…n}\ x = x_1\ op\ x_2\ op\ ...\ op\ x_n$

Parallelizability: Exploit associativity of op



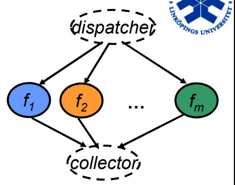
Notation with higher-order function*:*

- y = ***reduce*** (op, x)

Task farming

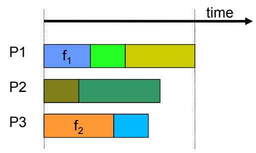
Independent computations $f_1, f_2, \ldots, f_m$ could be done in parallel and/or in arbitrary order, e.g.

- independent loop iterations
- independent function calls



Scheduling problem

- n tasks onto p processors
- static or dynamic
- Load balancing



Notation with higher-order function:

- $(y_1,\ldots,y_m)$ = ***farm*** $(f_1, \ldots, f_m)$ $(x_1,\ldots,x_n)$

Parallel Divide-and-Conquer

(Sequential) Divide-and-conquer:

- *Divide*: Decompose problem instance P in one or several <u>smaller</u> <u>independent</u> instances of the same problem, $P_1, \ldots, P_k$
- For all i: If P_i *trivial*, solve it *directly*.
- Else, solve P_i by recursion.
- *Combine* the solutions of the P_i into an overall solution for P

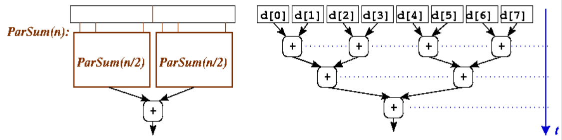
Parallel Divide-and-Conquer:

- Recursive calls can be done in parallel.
- Parallelize, if possible, also the divide and combine phase.
- Switch to sequential divide-and-conquer when enough parallel tasks have been created.

Notation with higher-order function:

- *solution* = ***DC*** (*divide, combine, istrivial, solvedirectly, n, P*)

Example: Parallel Divide-and-Conquer



Example: **Parallel Sum** over integer-array x

Exploit associativity:

$Sum(x_1,\ldots,x_n) = Sum(x_1,\ldots x_{n/2}) + Sum(x_{n/2+1},\ldots,x_n)$

Divide: trivial, split array x in place

Combine is just an addition.

y = ***DC*** (*split, add, nIsSmall, addFewInSeq, n, x*)

Data parallel reductions are an important special case of DC.

Example: Parallel Divide-and-Conquer (2)

Example: **Parallel QuickSort** over a float-array x

Divide: Partition the array (elements <= pivot, elements > pivot)

Combine: trivial, concatenate sorted sub-arrays

sorted = ***DC*** (*partition, concatenate, nIsSmall, qsort, n, x*)

Pipelining

applies a sequence of <u>dependent</u> computations $(f_1, f_2, \ldots, f_k)$ elementwise to data sequence $x = (x_1,\ldots,x_n)$

- For fixed x_j, compute $f_i(x_j)$ before $f_{i+1}(x_j)$
- Computations of f_i on <u>different</u> x_j are independent.

Parallelizability: Overlap execution of all f_i for k subsequent x_j

- *time=1: compute* $f_1(x_1)$
- *time=2: compute* $f_1(x_2)$ *and* $f_2(x_1)$
- *time=3: compute* $f_1(x_3)$ *and* $f_2(x_2)$ *and* $f_3(x_1)$
- ...
- Total time: O ($(n+k)$ $\max_i(time(f_i))$) with k processors

Notation with higher-order function:

- $(y_1,\ldots,y_n)$ = ***pipe*** ($(f_1, \ldots, f_k)$, $(x_1,\ldots,x_n)$)

Skeletons

Skeletons are reusable, parameterizable components with well defined semantics for which efficient parallel implementations may be available.

Inspired by <u>higher-order functions</u> in functional programming

One or very few skeletons per parallel algorithmic paradigm

- map, farm, DC, reduce, pipe, scan ...

<u>Parameterised</u> in user code

- Customization e.g. by instantiating a skeleton template in a user function

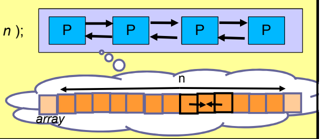
Composition of skeleton instances in program code normally by <u>sequencing+data flow</u>

- e.g. *squaresum(x)* can be defined by

```
{
    tmp = map( sqr, x );
    return reduce( add, tmp );
}
```

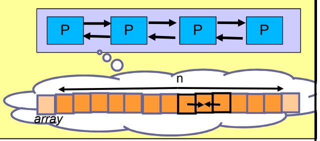
Example 1 revisited: 1D smoothening in C + Skeletons

```
…
float filter (float a, b, c) { return wa*a + wb*b + wc*c; }

float elemError ( float a, b ) { return fabs ( a – b ); }
…
void main ( int argc, char *argv[] ) {
   …
   DistrFloatArray *array = new_DistrFloatArray ( n + 2 );
   DistrFloatArray *tmp = new_DistrFloatArray ( n + 2 );
   DistrFloatArray *err = new_DistrFloatArray ( n + 2 );
   …
   while ( globalerr > 0.1 ) {
      map_with_overlap( filter, 1, tmp, array+1, n );
      map( elemError, err, array+1, tmp, n );
      reduce( fmax, &globalerr, err, n );
      map( copy, array+1, tmp, n );
   }
   …
}
```



Example 1 revisited: 1D smoothening in C++ + Skeletons

```
…
float filter (float a, b, c) { return wa*a + wb*b + wc*c; }

float elemError ( float a, b ) { return fabs ( a – b ); }
…
void main ( int argc, char *argv[] ) {
   …
   Vector<float> array = new Vector<float> ( n + 2 );
   Vector<float> tmp = new Vector<float> ( n + 2 );
   Vector<float> err = new Vector<float> ( n + 2 );
   …
   while ( globalerr > 0.1 ) {
      map_with_overlap( filter, 1, tmp, array );
      map( elemError, err, array, tmp );
      reduce( fmax, &globalerr, err );
      map( copy, array, tmp );
   }
   …
}
```



Generic containers (e.g., Vector<>)
provide a cleaner interface and encapsulate metadata (e.g., size) and internal state of operand data (e.g., storage format and location, distribution, modified, …)

Skeletons (cont.)

Skeletons encapsulate completely all coordination of parallelism and platform-specific issues

Threads/Process creation/termination, communication, synchronization

→ **Code portability**

→ **Reuse of the coordination code across multiple skeleton instances**

Skeletons may also have a sequential implementation

Uniform treatment of sequential and parallel programming

Skeleton Programming Systems

4 basic approaches for realizing skeletons (esp., parameterisation mechanism):

- Library of higher-order functions (functional or imperative)
- OO class library *(subclass and define abstract parameter method(s))*
- New language constructs (*intrinsics / compiler-known functions*)
- Generative programming, Static metaprogramming (Macros / *templates*)

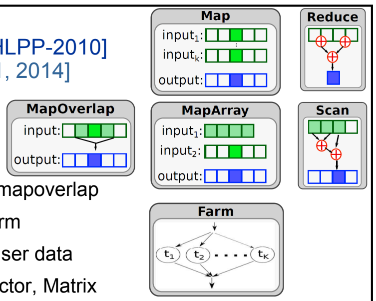
Many research prototypes, e.g.:

- *P3L* - C + skeletons
- *SCL, Eden, HDC* - functional
- *eSkel* - C + MPI
- *Lithium* - Java + RMI
- *BlockLib* - C + macros (generative) + DMA for Cell BE
- *muskel, ASSIST* - C++, grid computing
- *MueSLi, QUAFF* - C++ based, MPI
- *SkePU*, SkelCL - C++ based, for GPU based systems

Domain-specific Skeleton Systems, e.g.

- *MallBa* (combinatorial optimization: BB, DP, GA, …)
- *MapReduce* (distributed data mining, Google)

SkePU [Enmyren, K. HLPP-2010] [Dastgeer 2011, 2014]



- C++ template library
- 6 dataparallel skeletons
  - Map, reduce, scan, mapreduce, maparray, mapoverlap
- 1 task-parallel skeleton: farm
- STL-based containers for user data
  - Smart containers for Vector, Matrix
- Generation of platform-specific variants for user functions
- Multiple back-ends: C, OpenMP, OpenCL, CUDA, StarPU
  - Hybrid CPU-GPU execution (with StarPU backend)
- Multi-GPU support
- Low overhead
- Tunable

The SkePU Multi-Backend Skeleton Library

Example: Dot Product

```
#include <iostream>

#include "skepu/vector.h"
#include "skepu/mapreduce.h"

BINARY_FUNC(plus, double, a, b,
   return a+b;
)

BINARY_FUNC(mult, double, a, b,
   return a*b;
)

….
```

Macro expands to platform-specific code versions for user functions

```
….

int main()
{
   skepu::MapReduce<mult, plus>
         dotProduct(new mult, new plus);

   skepu::Vector<double> v0(1000,2);
   skepu::Vector<double> v1(1000,2);

   double r = dotProduct(v0,v1);

   std::cout<<"Result: " <<r <<"\n";

   return 0;
}
```

Generate a skeleton instance (function dotProduct)

Generic container holds operand data

Invocation at run time

SkePU: Generating platform-specific variants of user functions

```
BINARY_FUNC(plus, double, a, b,
    return a+b;
)
```

expands to

```
struct plus
{
    skepu::FuncType funcType;
    std::string func_CL;
    std::string funcName_CL;
    std::string datatype_CL;
    plus()
    {
        funcType = skepu::BINARY;
        funcName_CL.append("plus");
        datatype_CL.append("double");
        func_CL.append(
        "double plus(double a, double b)\n"
        "{\n"
        "    return a+b;\n"
        "}\n");
    }
    double CPU(double a, double b)
    {
        return a+b;
    }
    __device__ double CU(double a, double b)
    {
        return a+b;
    }
};
```

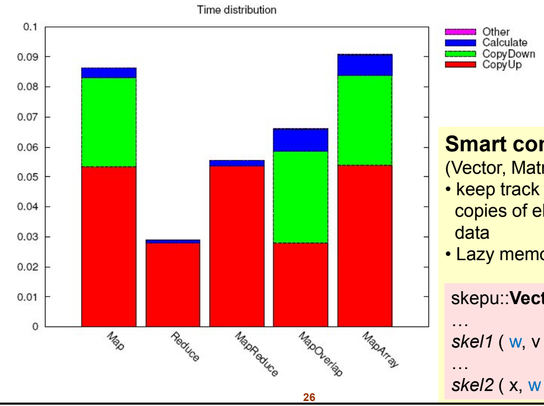
OpenCL

CPU

CUDA

Macro expands to a struct with platform-specific versions of the user function

Importance of Data Transfer Optimization



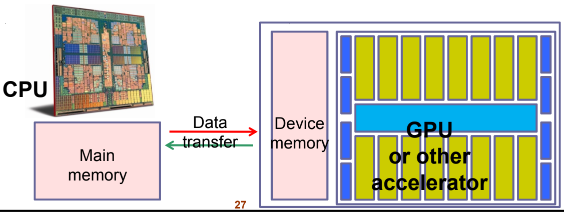
Smart containers (Vector, Matrix):
- keep track of current copies of element data
- Lazy memory copying

skepu::**Vector** v, w, x;
…
skel1 (w, v);
…
skel2 (x, w);

26

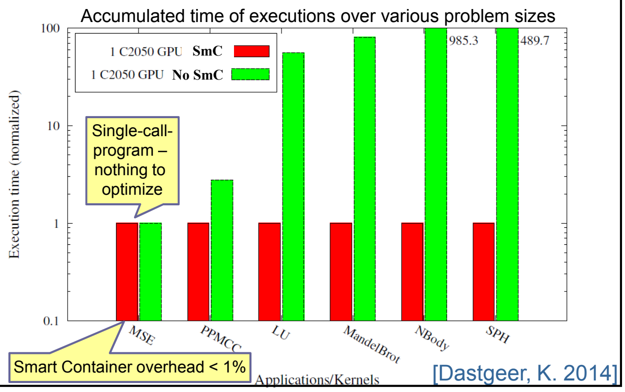
SkePU Containers

- For non-scalar operand data
  - STL-like Vector
  - Matrix
- Handle transparent data access and memory transfers between host and device (“***smart containers***”)
  - Lazy memory copying
  - Implements (coarse-grained) memory coherence in software



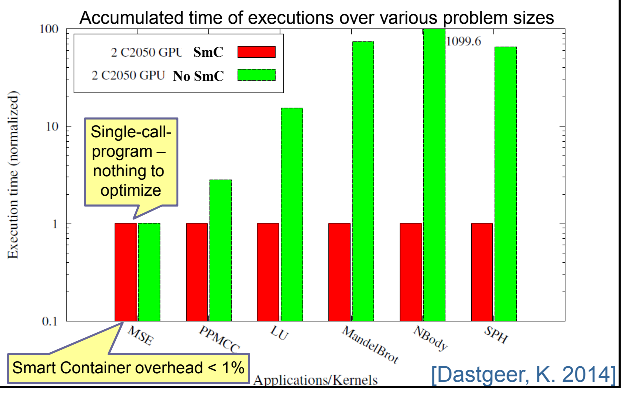
27

Evaluation
Speedup by (new) Smart Containers – 1 GPU



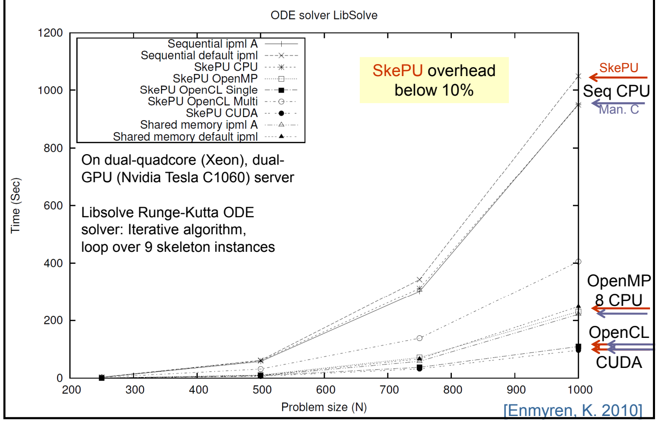
[Dastgeer, K. 2014]

Evaluation
Speedup by (new) Smart Containers – 2 GPUs



[Dastgeer, K. 2014]

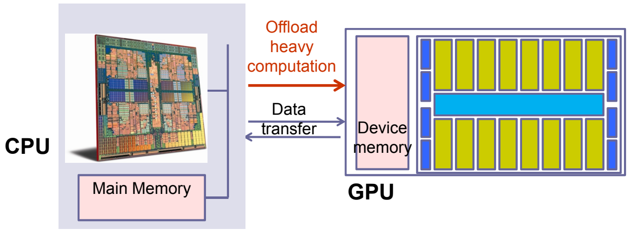
Comparison: SkePU vs hand-written code
ODE RK-Solver from libsolve



[Enmyren, K. 2010]

Programming of GPU-based Systems … with SkePU, so far…

- Portability ☺
- Programmability ☺
- Performance portability ☹

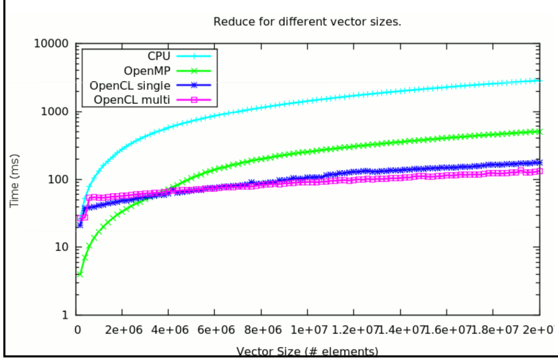


Implementation selection

- For each skeleton call, we need to decide:
  - Which skeleton implementation to use
- The decision depends upon:
  - Skeleton type
  - User function / actual computation
  - Target architecture
  - Problem size
  - etc.

SkePU Tunability [Dastgeer *et al.* IWMSE-2011]

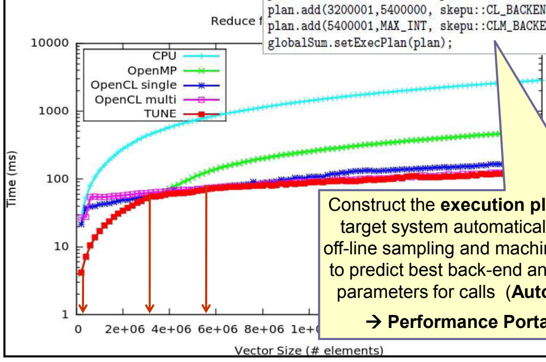
- Different Back-Ends, each with tunable parameters



SkePU Tunability [Dastgeer *et al.* IWMSE-2011]

- Execution plan

```
skepu::Reduce<plus> globalSum(new plus);
skepu::ExecPlan plan;
plan.add(1,3500, skepu::CPU_BACKEND);
plan.add(3501,3200000, skepu::OMP_BACKEND, 8);
plan.add(3200001,5400000, skepu::CL_BACKEND, 65536, 128);
plan.add(5400001,MAX_INT, skepu::CLM_BACKEND, 65536, 128);
globalSum.setExecPlan(plan);
```



Construct the **execution plan** for the target system automatically, using off-line sampling and machine learning to predict best back-end and tunable parameters for calls (**Auto-tuning**)

→ Performance Portability

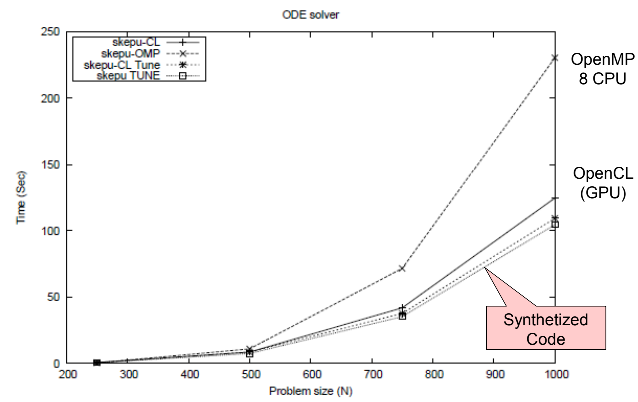
Auto-Tuning SkePU

- Use off-line, machine learning approach to predict best back-end and its tunable parameters for given problem size
- Basic tunable parameters for each skeleton:
  #threads (on CPU), thread block size + grid size (on GPU)

```
1 --- 50000 OMP_BACKEND 8
50001 --- 150000 CL_BACKEND 32 16384
150001 --- 225000 CL_BACKEND 128 2048
225001 --- 1050000 CL_BACKEND 512 2048
...
71500001 --- INFINITY CLM_BACKEND 256 2048
```

- Performance portability demonstrated across multiple GPU systems [Dastgeer, K. IWMSE'11, APPT'13]

RK-ODE-solver, synthetized from tuned SkePU skeletons



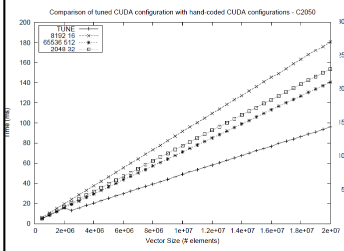
Performance portability demonstrated with autotuned SkePU skeletons

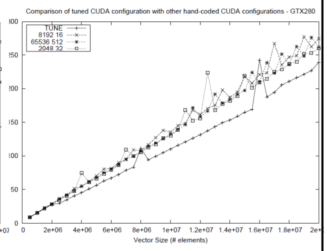
Developed on **dual-Tesla C1060 GPU server**

Re-calculate execution plan | Re-calculate execution plan

Autotuned vs. Handcoded CUDA map on **Nvidia Fermi C2050 GPU**

Autotuned vs. Handcoded CUDA map on **Nvidia GeForce GTX280 GPU**





→ Source code left untouched when migrating to new system!

Alternative: Construct execution plan implicitly at runtime (StarPU back-end)

Coulombic application, 24K x 24K matrix, 3 successive executions, 1 CPU + 1 GPU.
First iteration: No data on GPU, no performance history → greedy partitioning
Data-aware scheduling (StarPU) learns at runtime
Later iterations: Some data on GPU, more performance-aware decisions: 7x speedup

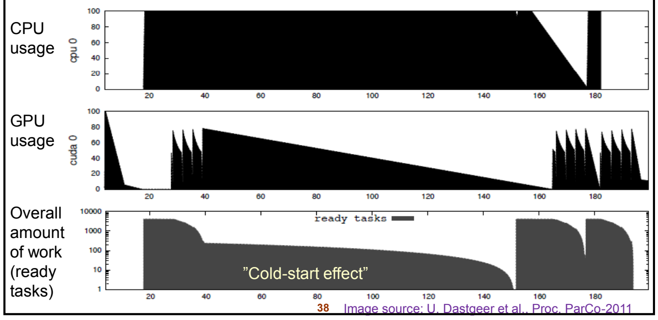
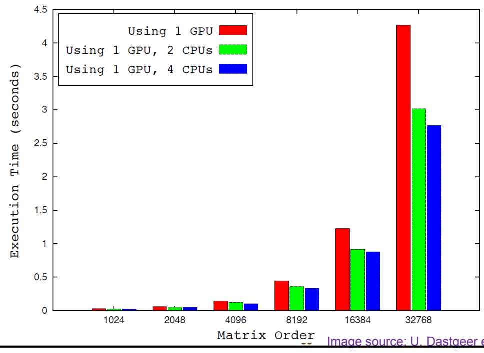


 Image source: U. Dastgeer et al., Proc. ParCo-2011

SkePU backend for StarPU runtime system

- Hybrid execution of tasks created from a single skeleton instance



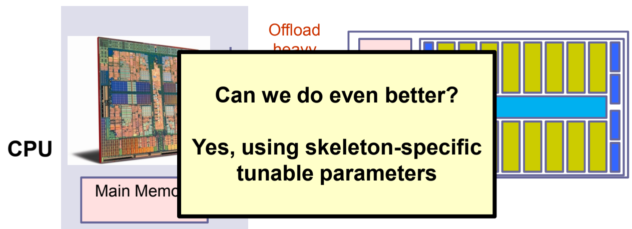
Coulombic Potential grid benchmark for different matrix sizes.

Platform: Dual-quadcore Xeon with Nvidia C2050 GPUs

Image source: U. Dastgeer et al., Proc. ParCo-2011

Programming of Heterogeneous Multi-/Manycore Systems … with SkePU

- Portability ☺
- Programmability ☺
- Performance portability ☺

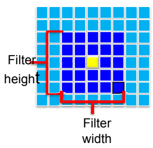


Can we do even better?

Yes, using skeleton-specific tunable parameters

Example: Autotunable 2D MapOverlap Skeleton

- Generic 2D MapOverlap skeleton
  - 2D Stencil computations
  - CUDA and OpenCL implementations for GPUs
    - Performance optimizations:
      - Use constant memory, shared memory, tiling
- Performance Portability →
  - Best tiling factor depends on architecture + user function
  - Parametric auto-tuning → predictor at runtime for best tiling factor
- Details: [Dastgeer, K., MULTIPROG-2012]



Example: Autotunable 2D MapOverlap Skeleton

GPU architecture	Memory-optimized	Adaptive Tiling, using Φ_{tiling}	Speedup
C2050	76 GFLOP/s	274 GFLOP/s	**3.6x**
GTX280	41 GFLOP/s	95 GFLOP/s	**2.3x**
8800GT	25 GFLOP/s	59 GFLOP/s	**2.4x**

Baseline: with standard optimizations (e.g. usage of shared and constant memory).

Conclusion (SkePU)

- SkePU skeletons are pre-defined generic components
  - For frequently occurring algorithmic patterns
  - map, reduce, scan, mapoverlap, farm …
- Multiple back-ends, multi-GPU support
  - Seq, OpenMP, OpenCL, CUDA
  - StarPU backend for task parallelism and hybrid parallel execution
  - MPI back-end for GPU clusters
- Smart containers to avoid unnecessary data transfers
- Auto-tunable
  - Off-line + on-line tuning of resource allocation for calls
  - Parametric autotuning for specific skeletons

Other Skeleton Programming Frameworks

Example: Intel TBB Algorithm Templates

Intel Threading Building Blocks (TBB)

- Library for programming multicore processors
- extends C++ with a task based parallel programming model including
  - tasks (also fine-grained), no threads
  - high-level parallel *algorithm templates* (functions - nestable),
    - data-parallel (e.g. parallel for, reduce, scan) and task-parallel (e.g. pipe)
    - User functions (body) more coarse-grained than individual elements, to better perform on CPU
  - concurrent containers,
  - mutexes, atomic operations, etc.
- sophisticated run-time task scheduling mechanism,
  - At runtime, the TBB run-time system creates tasks and assigns them to threads which the OS schedules to cores
  - Dynamic load balancing by *task stealing*.

TBB example

Intel: TBB tutorial, 2010, www.intel.com and threadingbuildingblocks.org

- Data-parallel loop in TBB

```
#include "tbb/tbb.h"
void ParallelSquare( float a[], size_t n )
{
   parallel_for( blocked_range<size_t>(0,n),
                 Square(a));
}
```

- Class Square defines a **functor** (= function object, instance of a class containing a member function that overloads the (.) operator → invoking "()" looks like a function call)
  - "element function", "user function":
    Works on a contiguous subarray (index subrange) at a time

Summary

- **Skeleton programming**
  - Algorithmic paradigms
  - Predefined generic parallel components, parameterized in user code
  - Hiding complexity (parallelism and low-level programming)
  - ☺ Abstraction
  - 😐 Enforces structuring
  - ☺ Parallelization for free
  - ☺ Easier to analyze and transform
  - ☹ Requires complete understanding and rewriting
  - ☹ Available skeleton set does not always fit
  - ☹ May lose some efficiency compared to manual parallelization
- Industry (beyond HPC domain) has adopted skeletons
  - map, reduce, scan in many modern parallel programming APIs
    - e.g., Intel Threading Building Blocks (TBB): par. for, par. reduce, pipe
    - NVIDIA Thrust
  - Google MapReduce (for distributed data mining applications)

Thesis projects available!

- Extension of SkePU
  - New skeletons, new containers, new platforms…
- Porting applications to SkePU
  - Medical image visualization
  - Linear equation system solvers
  - …
- Automated tuning for new optimization objectives (energy)
- Improvements of the autotuning framework
- Static selection for multiple calls to consider inter-call effects
- SkePU is an open-source project
  - Documentation + download: www.ida.liu.se/~chrke/skepu

TDDD56 Multicore Computing /
TDDC78 Programming of Parallel Computers

Prof. Dr. Christoph Kessler, IDA, Linköping university

Questions?

Some literature on skeleton programming

M. Cole: *Algorithmic Skeletons: Structured Management of Parallel Computation*, MIT Press & Pitman, 1989. http://homepages.inf.ed.ac.uk/mic/Pubs/pubs.html

S. Pelagatti: *Structured Development of Parallel Programs.* Taylor and Francis, 1998.

F. Rabhi and S. Gorlatch (eds.): *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag, 2003.

M. Ålind, M. Eriksson, C. Kessler: BlockLib: A Skeleton Library for Cell Broadband Engine. Proc. ACM Int. Worksh. on Multicore Software Engineering, Leipzig, 2008.

J. Enmyren, C. Kessler: SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems. Proc. HLPP-2010 Int. Workshop on High-Level Parallel Programming, Sep. 2010, Baltimore, USA. ACM.

U. Dastgeer: Performance-Aware Component Composition for GPU-Based Systems. PhD thesis, Linköping University, 2014. Chapter 3.

SkePU Documentation and Download: http://www.ida.liu.se/~chrke/skepu

Acknowledgment: SkePU research and development is funded by EU FP7 projects PEPPHER (2010-12), EXCESS (2013-16); and by SeRC (www.e-science.se), 2011-2014

Glossary

- **Performance Portability**

  … is the ability of a program to automatically adapt to a new execution platform to achieve an automated best-effort optimization of performance on the new target system, without manual rewriting / reoptimization.

- **[Algorithmic] Skeleton**

  … is a pre-defined, generic software construct
  for high-level programming
  that implements a specific *pattern* of control and data flow,
  that can be *parameterized* by problem-specific code
  to instantiate a problem-specific function, and
  whose implementation internally *encapsulates*
  all platform-specific details such as parallelism,
  heterogeneity, communication and synchronization.