PEPPHER

Programmability and Performance Portability for Heterogeneous Many-Core Systems

Siegfried Benkner

(on behalf of PEPPHER Consortium)

Research Group Scientific Computing

Faculty of Computer Science

University of Vienna, Austria

http://www.par.univie.ac.at/

Research Group Scientific Computing

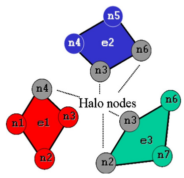
One of nine research groups at the Faculty of Computer Science.

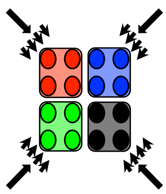
Staff: 20 people (5 Faculty, 12 Research, 3 Admin/Support)

Parallel Computing / HPC

- Programming Models and Languages
- Compiler and Runtime Technologies
- Programming Environments and Tools

Vienna Fortran, HPF+, Hybrid Programming, Many-Core…

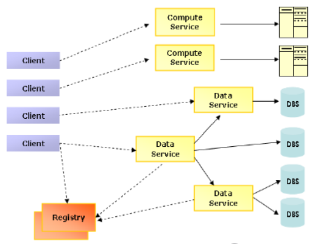




Grid/SOA/Cloud Computing

- HPC Application Services
- On-demand supercomputing, QoS
- Data Virtualization & Integration & Mining

Grid Miner, Vienna Cloud Environment, …



Research Group Scientific Computing

Selected European Research Projects

- ☐ EU Project **HPF+**, 1996-1998
- ☐ EU Project **GEMSS**, 2002-2005
- ☐ EU Project **@neurIST**, 2005-2010
- ☐ EU Project **ADMIRE**, 2008-2011
- ☐ EU Project **PEPPHER**, 2010-2012
- ☐ EU Project **VPH-SHARE**, 2011-2015
- ☐ EU Project **AutoTune**, 2011-2014

Talk Outline

PEPPHER

- ☐ Heterogeneous Many-Core Systems
- ☐ The PEPPHER Approach
- ☐ Basic Coordination Language & Pipeline Patterns
- ☐ Transformation System & Coordination Layer
- ☐ Experimental Results
- ☐ Conclusions & Future Work

PEPPHER

Heterogeneous MC Architectures

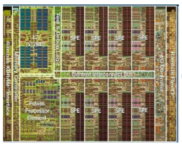
Move towards heterogeneous many-core architectures

- Better performance/power ratio
- Different types of cores; same cores but different clock frequencies, ...
- Specialized cores for specific tasks/application domains

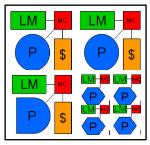
→ Parallelization & Specialization (mitigate Amdahl's law)

Examples

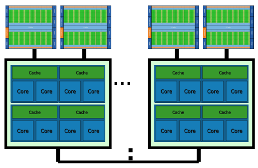
- Cell Processor: PPU + 8 SPUs
- SARC Research Processor
- CPU + GPU/Accelerators
- Tianhe-1A, Roadrunner, TSUBAME
- Nvidia Tegra, AMD Fusion, IvyBridge, ...



Cell BE



SARC



GPU Cluster

PEPPHER

Programming Heterogeneous Many-Cores

Much harder than for homogeneous systems

- Need of allocating and managing resources
- Explicit memory management (DMA transfers, double buffering, local stores …)
- Partitioning of code for different cores
- Different memory models, ISAs, compilers, APIs, programming models

Current Solutions – (Mainly) Static Code Offloading

- **Low-Level**

  IBM CellSDK, NVIDIA CUDA, ATI Stream SDK, OpenCL, …

- **Higher-Level**

  PGI Accelerator, HMPP, Codeplay Offload++; …

Challenges/Requirements

PEPPHER

Increasing architectural complexity/diversity

- Compilers can't keep pace with shorter innovation cycles
- Code-rewrite by hand not feasible
- Simple, static offloading too inefficient
- Applications must automatically adapat to new architectures

New programming models ?

- No „one-fits-all" model
- Need to integrate different models

Programmability/Productivity

- Raise level of abstraction
- Hide/Automate low-level optimization tasks

(Performance) Portability of Major Concern

- Consider different aspects not just FLOPs
- Energy/Power as important as performance

Compositional Approach

Adaptation Auto-Tuning

Resource-Awareness Performance & Platform Models

EU Project PEPPHER

PEPPHER

Performance Portability & Programmability for Heterogeneous Many-Core Architectures

- EU ICT Call 4, Computing Systems; 3 years from 1.1.2010
- 9 Partners, Coordinated by University of Vienna
- **http://www.peppher.eu**

Goal: Enable **portable**, **productive** and **efficient** programming of heterogeneous many-core systems.

- Focus on **single-node systems** (e.g. CPU/GPU/APU/MIC)
- **Holistic approach** considering all layers of SW stack + HW issues.

EU Project PEPPHER

PEPPHER

Holistic Approach

- Higher-Level Support for Parallel Program Development
- Auto-tuned Algorithms & Data Structures
- Compilation Strategies
- Runtime Systems
- Hardware Mechanisms

Crosscutting Application Domains

- Embedded – General Purpose – HPC

Project Consortium

PEPPHER

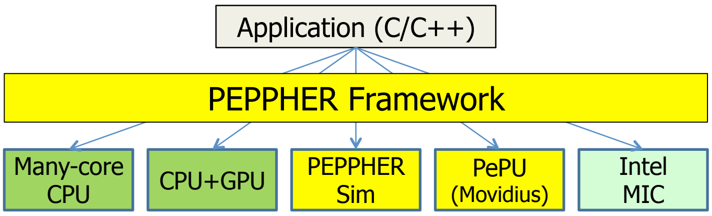
- ☐ University of Vienna (Coordinator), Austria
  Siegfried Benkner, Sabri Pllana
- ☐ Vienna University of Technology, Austria
  Jesper Larsson Träff
- ☐ Linköping University, Sweden
  Christoph Kessler
- ☐ Codeplay Software Ltd., UK
  Andrew Richards
- ☐ Karlsruhe Institute of Technology, Germany
  Peter Sanders
- ☐ Chalmers University, Sweden
  Philippas Tsigas
- ☐ INRIA, France
  Raymond Namyst
- ☐ Intel GmbH, Germany
  Herbert Cornelius
- ☐ Movidius Ltd., Ireland
  David Moloney

Performance.Portability.Programmability

PEPPHER

Methodology & framework for development of performance portable code.

- Execute same application efficiently on different heterogeneous architectures.
- Support multiple parallel APIs: **OpenMP, OpenCL, CUDA,** pThreads, ...



Focus: Single-node/chip heterogeneous architectures

Approach

- **Multi-architectural, performance-aware components**
  multiple implementation variants of functions; each with a performance model
- **Task-based execution model & intelligent runtime system**
  runtime selection of best task implementation variant for given platform

Motivating Example

PEPPHER

Cholesky factorization

```
FOR k = 0..TILES-1
   POTRF(A[k][k])
   FOR m = k+1..TILES-1
         TRSM(A[k][k], A[m][k])
   FOR n = k+1..TILES-1
         SYRK(A[n][k], A[n][n])
         FOR m = n+1..TILES-1
               GEMM(A[m][k], A[n][k], A[m][n])
```

Utilize expert written components:
BLAS kernels from MAGMA and PLASMA

Implementation variants:
- multi-core **CPU** (PLASMA)
- **GPU** (MAGMA)

Make into PEPPHER component:

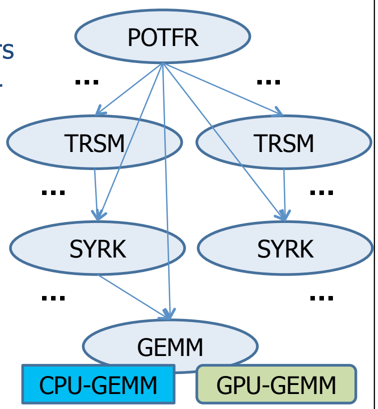
Interface, implementation variants + meta-data

PEPPHER Approach

PEPPHER

Task variant selection & scheduling

- Data/topology-aware: minimize data transfers
- Performance-aware: minimize make-span, or other objective (power, …)



Component implementation variants

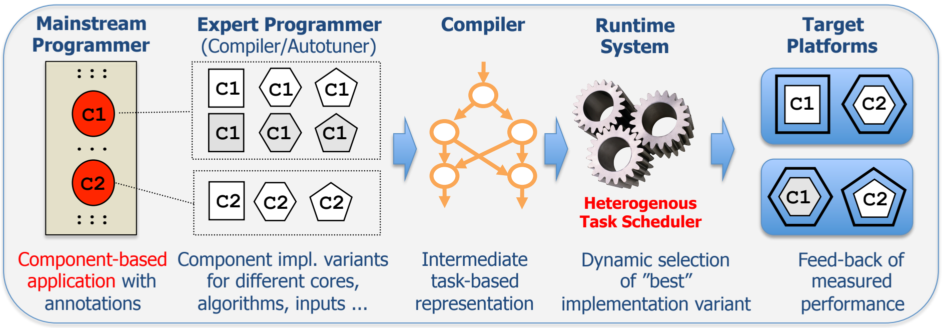
- for different architectures/platforms, …
- Generic platform model for selection
- Performance-aware components

Multi-level parallelism

- Coarse-grained inter-component parallelism
- Fine(r) grained intra-component parallelism
- Exploit ALL execution units

PEPPHER Approach

PEPPHER



Programmer

- Identify performance critical parts
- Transform into performance-aware components
- Provide implementation variants for different core architectures or utilize expert components

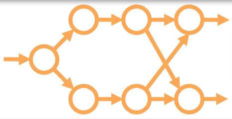
PEPPHER framework

- Management of components and implementation variants
- Compilation/code generation
- Component variant selection
- Dynamic, performance-aware task scheduling (StarPU runtime)

PEPPHER Framework

PEPPHER

Framework layer	Description
Applications Embedded · General Purpose · HPC	C/C++ source code with annotated component calls
High-Level Coordination/Patterns/Skeletons	Asynchronous calls, Data distribution Patterns, SkePU Skeletons
Components C/C++, OpenMP, CUDA, OpenCL, TBB, Offload	Component implementation variants for different core architectures ... algorithms, ...
Autotuned Algorithms · **Data Structures**	Autotuned Data Structures & Algorithms
Transformation & Composition	Component glue code Static variant selection (if any)
PEPPHER Taskgraph · **Performance Models**	Component task graph with explicit data dependecies
PEPPHER Run-time (StarPU) Scheduling Strategy · Scheduling Strategy	Performance-aware, data-aware dynamic scheduling of „best" component variants onto free execution units
Drivers (CUDA, OpenCL, OpenMP)	
CPU · **GPU** · **MIC** · **SIM** · **PePU**	Single-node heterogeneous manycore SIM = PEPPHER simulator PePU = Peppher proc. unit (Movidius)



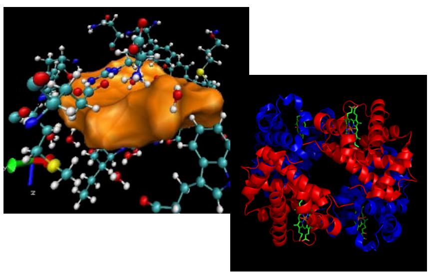
Applications

PEPPHER

Embedded, General Purpose, HPC

- **Applications**
  - KIT: Suffix array construction
  - UNIVIE: Data compression, OpenCV
  - Codeplay: Bullet (games physics sim.)
  - Movidius: Computational photography
  - Intel: GROMACS
- **Kernels**
  - INRIA: FFT
  - INRIA: MAGMA/PLASMA (QR)
  - INRIA: RODINIA (CFD solver)
  - KIT: STL (sort, find, random_shuffle)

Software optics



Molecular dynamics simulation

PEPPHER Component Model

PEPPHER

Main Ideas:

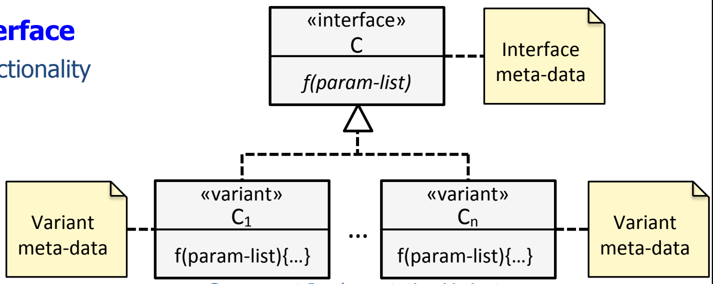
- **Separation of concerns**
  - Specification vs. implementation
  - Mainstream vs. expert programmer
  - Hide different implementation variants behind interface
- **Resource- & performance-aware components**
  - Rich component meta-data (external, XML)
  - Input/output; Platform/Resource requirements; Performance aspects
  - Component performance models
- **Dynamic, task-based execution model**
  - Runtime component variant selection and scheduling
  - Support different levels of parallelism

PEPPHER Components

PEPPHER

Component Interface

- Declaration of functionality



Component Implementation Variants

Implementation Variants

- Different architectures/platforms
- Different algorithms/data structures
- Different input characteristics
- Different performance goals
- Written by expert programmer
  (or generated, e.g. auto-tuning)

Features

- Different programming languages (C/C++, OpenCL, Cuda, OpenMP)
- Task & Data parallelism

Constraints

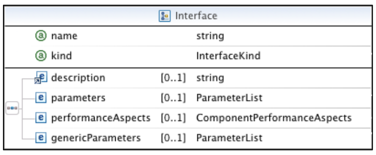
- No Side-effects; Non-preemptive
- Stateless; Composition on CPU only

Component Meta-Data

PEPPHER

Interface Meta-Data (XML)

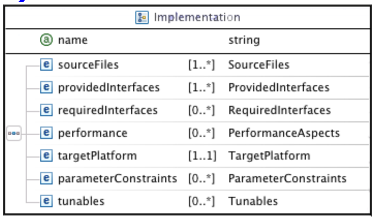
- **Parameter intent** (read/write)
- Supported **performance apsects** (execution-time, power)



XML Schema for Interface Meta-Data

Implementation Variant Meta-Data (XML)

- **Supported target platforms (PDL)**
- **Performance Model**
- Input data constraints (if any)
- Tunable parameters (if any)
- Required components (if any)



XML Schema for Variant Meta-Data

Key issues

- **Make platform specific optimizations/dependencies explicit.**
- Make components **performance-** and **resource-aware**.
- Support runtime variant selection.
- Support code transformation and auto-tuning.

Explicit Platform Descriptions

PEPPHER

Goal: Make platform specific information explicit and available in a systematic way to tools and users.

XML-based Platform Description Language (PDL)

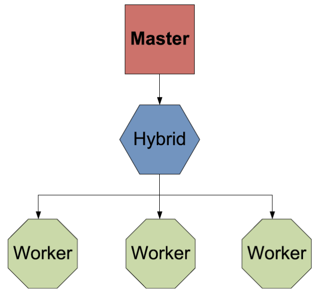
- Capture different aspects of heterogeneous platforms
  - **Control views:** delegation of computational tasks between processing units; hierarchical organization of PUs
  - **Hardware / Software properties** (e.g., core-count, memory sizes, available libraries)
- Supports expression of **platform usage patterns** (e.g. Master-Worker)
- Not a hardware description language! Programmer centric view on available resources (→ platform)

Properties
cpu-count
gpu-count
cpu-cores
opencl-available
cpu-core-arch
memory-levels
cache-size
gpu-vendors
gpu-arch
runtime-system
gpu-libraries
cuda-available
gpu-device-memory
gpu-sm-count
gpu-cores-per-sm
main-memory
cpu-cache
cpu-cacheline
memory-affinity
...

Platform Descriptors

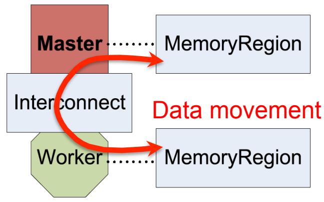
PEPPHER

Processing Units (PUs)
- **Master** (initiates program execution)
- **Worker** (executes delegated tasks)
- **Hybrid** (master & worker)



Memory Regions
- Express key characteristics of memory hierarchy
- Can be defined for all processing units

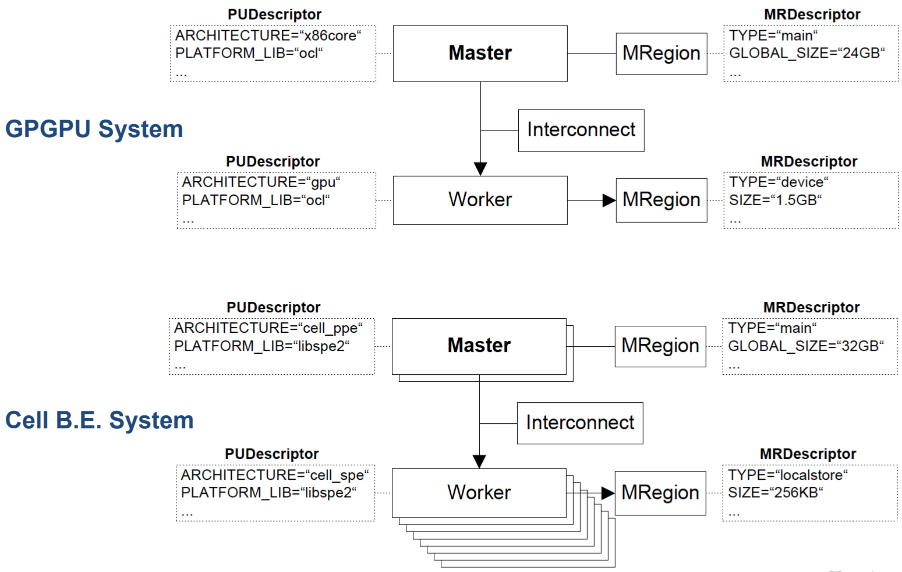
Interconnects
- describe communication facilities between PUs



Properties
- Hardware and software properties using generic key/value mechanism

PDL Examples

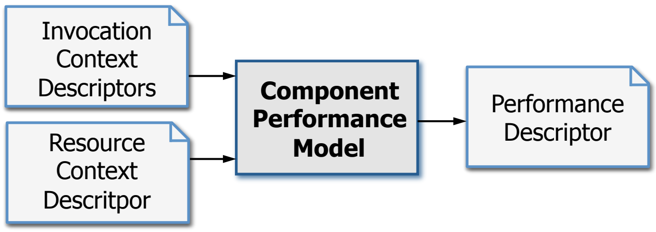
PEPPHER



Performance-Aware Components

PEPPHER

Each component is associated with an **abstract performance model.**



- ☐ **Invocation Context:** captures performance-relevant information of input data (problem size, data layout, etc.)
- ☐ **Resource Context:** specifies main HW/SW characteristics (cores, memory, …)
- ☐ **Performance Descriptor:** usually includes (relative) runtime, power estimates

Generic performance prediction function:

```
PerfDsc getPrediction(InvocationContextDsc icd, ResourceContextDsc rcd)
```


Basic Coordination Language

PEPPHER

Component calls

- asynchronous & synchronous calls

```
#pragma pph call
cf1(A, N, B, M); // A:read, B:write (XML meta-data)

#pragma pph call
cf2(B, M);
```

```
#pragma pph call sync
cf(A, N); // block until cf() returns
```

Basic Coordination Language

PEPPHER

Memory Consistency

- flush; for ensuring consistency btw. host and workers

```
#pragma pph call
cf1 (A, N);
...
#pragma pph flush(A) // block until A has become available
int first = A[0]; // explicit flush req. since A is accessed
```

Component calls

- implicit memory consistency across workers

```
#pragma pph call
cf1 (A, N); // A: read / write
... // implicit memory consistency on workers only
... // no explicit flush is needed here provided A
... // is not accessed within the master process
#pragma pph call
cf2(A, N); // A:read; actual values of A produced by cf1()
```


Basic Coordination Language

PEPPHER

Data Partitioning

- generate multiple component calls, one for each partition (cf. HPF)

```
#pragma pph call partition(A(size:BLOCK(size/2)))
cf1(A, size);
```

Access to Array Sections

- specify which array section is accessed in component call (cf. Fortran array sections)

```
#pragma pph call access(A(size:50:size-1))
cf(A+50, size-50);
```

Basic Coordination Language

PEPPHER

Parameter Assertions

- influence component variant selection

```
#pragma pph call parameter(size < 1000)
cf1(A, size);
```

Optimization Goals

- specify optimization goals to be taken into account by runtime scheduler

```
#pragma pph call optimize(TIME)
cf1(A, size);
...
#pragma pph call optimize(POWER < 100 && TIME < 10)
cf2(A, size);
```

Execution Target

- specify pre-defined target library (e.g., OPENCL) or processing unit group from PDL platform descriptor

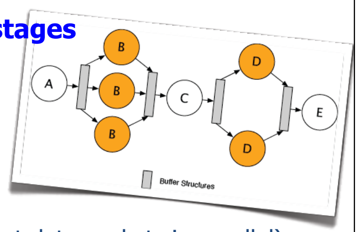
```
#pragma pph call target(OPENCL)
cf(A, size);
```


Pipeline Pattern

PEPPHER

Stream of data processed in sub-sequent stages

- Linear vs. non-linear pipelines
- Splitting, merging, replication of stages



Different types of parallelism

- Pipeline/Task Parallelism (stages process different data packets in parallel)
- Data Parallelism (within a stage)

Realization of pipelined applications on heterogeneous MC?

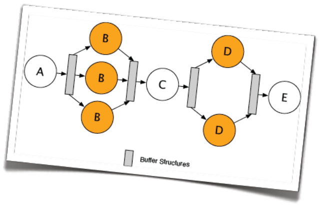
- High-level language support (annotation of while loops)
- Pipeline stage → component implementation variants (CPU, GPU, ...)
- Automatic data/buffer management
- Runtime scheduling of stage instances (tasks) to different core types

Language Support – Pipeline Pattern

PEPPHER

Annotation of while-loops

- Pipeline stages correspond to component calls
- Buffer management (size, order-type)
- Support for stage replication and stage merging




```
unsigned int N = get_max_execution_units();
...
#pragma pph pipeline with buffer(PRIORITY,N*2)
while(image.number < 32) {
        readImage(file,image);
        #pragma pph stage replicate(N) {
                resizeAndColorConvert(image);
                detectFace(image,outImage);
        }
        writeFaceDetectedImage(file,outImage);
}
```


Transformation System

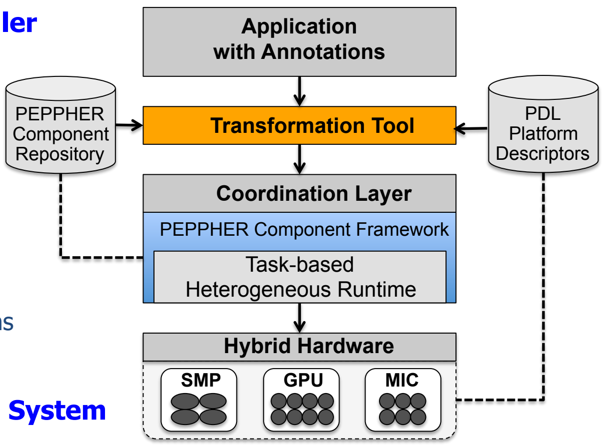
PEPPHER

Source-to-Source Compiler

- based on ROSE
- variant pre-selection if possible
- generates C++ with calls to coordination layer

Coordination Layer

- Pattern-specific optimizations on top of runtime layer

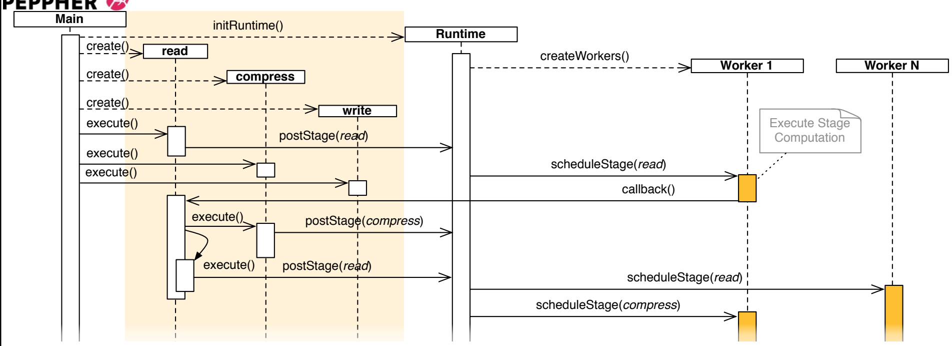


Heterogeneous Runtime System

- Based on INRIA's StarPU runtime system
- Selection of stage implementation variants based on available hardware resources
- Data-aware & performance-aware task scheduling onto heterogeneous PUs

Pipeline Coordination

PEPPHER



Local Coordination Strategy

- execute() and callback() methods of stages
- **execute()**: stage is posted to runtime system if input buffer is ready
- **callback()**: initiated by runtime after stage has finished execution
  - calls execute() of neighbor stage(s)
  - calls execute() on itself to initiate next stage instance

PEPPHER Runtime System

PEPPHER

Heterogeneous Runtime System (based on INRIA's **StarPU**)

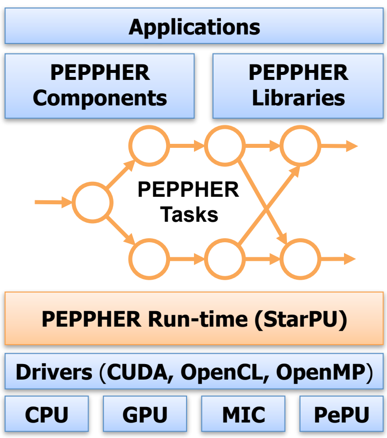
- Selection of component variants based on available hardware resources
- Data-aware & performance-aware task scheduling onto heterogeneous PUs

Tasks
- Explicit dependencies with other tasks
- Multiple implementations (GPU, CPU)

Automatic data transfer
- Virtual Shared Memory (VSM) layer
- Minimize data transfers btw. PUs

Flexible scheduling strategies
- Performance-aware
- Scheduling algorithm = plug-in

Performance Feed-back



PEPPHER Runtime System (StarPU)

PEPPHER

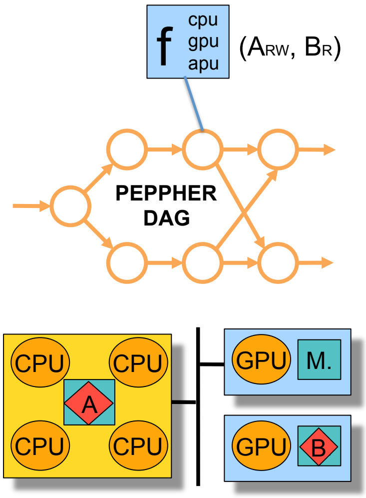
Schedule dynamic DAG of tasks onto pool of heterogeneous processing units.

Tasks

- Multiple implementations (e.g.; CPU, CUDA, OpenCL, OMP)
- Data input & output
- Dependencies with other tasks
- Scheduling hints

High-level data management layer

- Automate data transfers btw. PUs
- Support for data partitioning
- Avoid unnecessary data transfers (VSM)



PEPPHER Runtime System (StarPU)

PEPPHER

Task completion time estimation

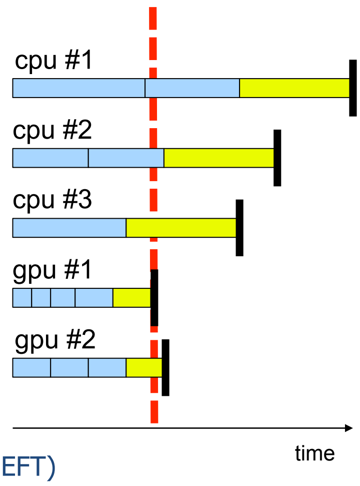
- History-based
- Component performance model

Data transfer time estimation

- Sampling based on off-line calibration

Used to improve scheduling

- e.g. Heterogeneous Earliest Finish Time (HEFT)

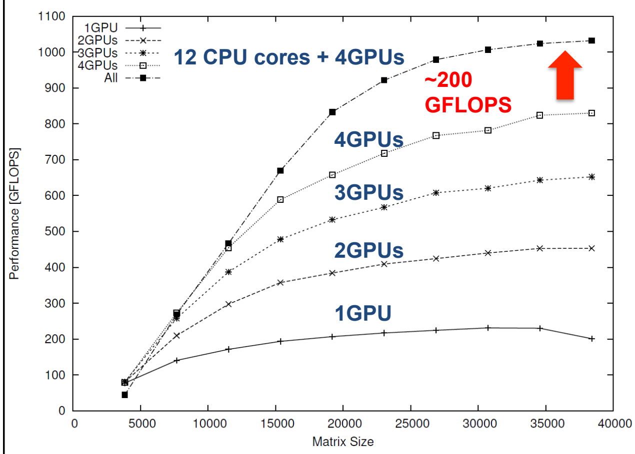


Experimental Results

PEPPHER

Tiled QR Decomposition & StarPU runtime

- Platform: 4 quad-core Opteron 8358 SE + 4 NVIDIA GPUs (C1060)



Single-Precision Performance
12 CPU cores ~150 GFLOPS

Performance increase
when we add to 4 GPUs
12 CPU cores ~200 GFLOPS

More performance
than expected!

Resolution:
Run-time schedules
best variant on best device

Experimental Results

PEPPHER

Tiled QR Decomposition ctd.

Affinity-based scheduling

- Select variants with highest expected performance
- Utilize **both** CPUs and GPUs

BLAS kernel	CPU Gflops	GPU GFlops	Speed-up ratio
SGEQRT	9	30	**3**
STSQRT	12	37	3
SORMQR	8.5	227	27
SSSMQR	10	285	**28**

- SSSMQR: 90% of tasks mapped to GPUs
- SGEQRT: 20% of tasks mapped to GPUs

Experimental Results

PEPPHER

Face detection application

- Based on OpenCV library
- Two different implementation variants for detection stage (CPU vs. GPU)
- Comparison to hand-coded Intel TBB version

```
unsigned int N = get_max_execution_units();

#pragma pph pipeline with buffer(PRIORITY,N*2)
while(image.number < 32) {
     readImage(file,image);
     #pragma pph stage replicate(N)
     {
            resizeAndColorConvert(image);
            detectFace(image,outImage);
     }
     writeFaceDetectedImage(file,outImage);
}
```


Experimental Results

PEPPHER

Results achieved with PEPPHER Transformation System

Architecture A

- 2 Intel Xeon X7560 (8 cores)
- RHEL 5.0

→ **speedup > 13**

Image Size	VGA	SVGA	XGA	QXGA
Intel TBB (1 Core)	15.61	23.51	41.84	170.58
our approach (1 Core)	12.40	17.85	30.72	140.86
Intel TBB (16 Core)	1.26	1.92	3.39	13.60
our approach (16 Cores)	1.16	1.72	2.91	12.33

execution times in s

Architecture B

- 2 Intel Xeon X5550 (4 cores)
- 1 GeForce GTX 480
- 1 GeForce GTX 285
- CUDA 4.0, RHEL 5.6

→ **speedup: 7-13**

Image Size	VGA	SVGA	XGA	QXGA
Intel TBB (1 Core)	12.75	20.07	35.15	145.68
our approach (1 Core)	9.62	14.33	24.94	111.45
our approach (1 Core + 1 GPU)	3.94	5.91	10.35	46.30
our approach (1 Core + 2 GPUs)	2.95	2.72	6.53	30.81
Intel TBB (8 Cores)	1.47	2.29	4.13	17.4
our approach (8 Cores)	1.18	1.78	3.58	13.69
our approach (7 Cores + 1 GPU)	1.13	1.63	2.91	11.89
our approach (6 Cores + 2 GPUs)	0.94	1.40	2.44	10.71

execution times in s

Related Work

PEPPHER

Task Offloading

- HMPP (CAPS, France)
- OmpSs (UPC, Barcelona)
- **OpenACC**
- Offload (Codeplay, UK)
- PGI Accelerate

Algorithmic Choice

- Elastic Computing (U. Florida)
- PetaBricks (MIT)
- ...

Streaming/Pipelining Languages

- StreamIt (MIT)
- Elk (Stanford, ELM Architecture)
- ...

Current European Projects

- ADVANCE (www.project-advance.eu)
- AUTOTUNE (www.autotune-project.eu)
- CARP (www.carpproject.eu)
- ENCORE (www.encore-project.eu)
- PARAPHRASE (www.paraphrase-ict.eu)

universität wien

Future Work

PEPPHER

- Extend language support for data partitioning/management
- Extend framework with other patterns (e.g. MapReduce)
- Component Performance Models
- Auto-tuning support for patterns
- New Architectures: NVIDIA Kepler, Intel MIC, Movidius Myriad Platform
- Optimization for energy-efficiency

AutoTune Project: Automatic Online Tuning

- TU Munich (M. Gerndt, coordinator)
- Uni Wien (S. Benkner)
- CAPS (F. Bodin)
- LRZ Munich (M. Brehm)
- UA Barcelona (A. Sikora)
- ICHEC Ireland (I. Girotto)

→ **http://www.autotune-project.eu/**

S. Benkner, University of Vienna InvasIC Seminar, Erlangen, June 22, 2012

universität wien

Conclusion

PEPPHER

PEPPHER Project

Programmability and Performance Portability for Heterogeneous Manycore Systems

- **Multi-architectural, resource-/performance-aware components**
- **High-Level coordination** primitives and **Patterns**
- Source-to-source transformation system
- **Heterogeneous runtime system** for selecting and scheduling component implementation variants to different execution units
- SkePU Skeletons and Composition Tool (not covered in this talk)
- Compilation to OpenCL - Codeplay OffloadCL (not covered)
- Autotuned Algorithms and Lock-free Data Structures (not covered)
- Hardware mechanisms for performance portability (not covered)

Acknowledgments

PEPPHER

- European Commission (ec.europa.eu)
- PEPPHER Consortium (www.peppher.eu)

Some of the consortium members (from left): D. Moloney, E. Marth, S. Pllana, V. Osipov, M. Wimmer, B. Bachmayer, P. Tsigas, J.L. Träff, C. Kessler, J. Singler, S. Benkner, D. Cederman, U. Dastgeer, H. Cornelius, S. Thibault, A. Richards, M. Sandrieser, U. Dolinsky, R. Namyst, C. Augonnet, H.C. Hoppe