HETEROGENEOUS **CPU+GPU** COMPUTING

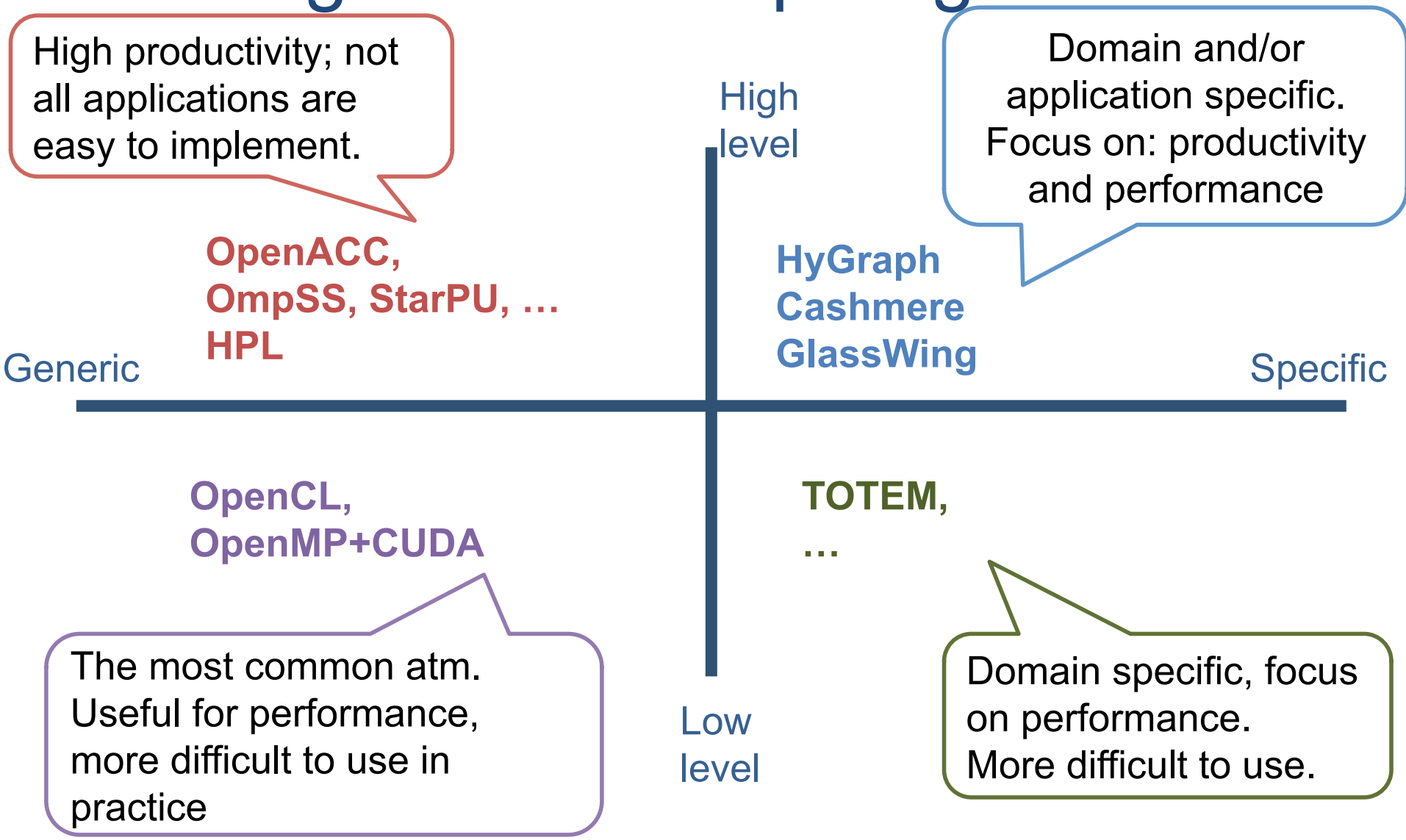
Ana Lucia Varbanescu – University of Amsterdam
a.l.varbanescu@uva.nl
Stijn Heldens – Twente University
mail@stijnh.nl

Significant contributions by: **Pieter Hijma** (UvA, NL), **Jie Shen** (TUDelft, NL), **Basilio Fraguela** (A Coruna University, ESP)

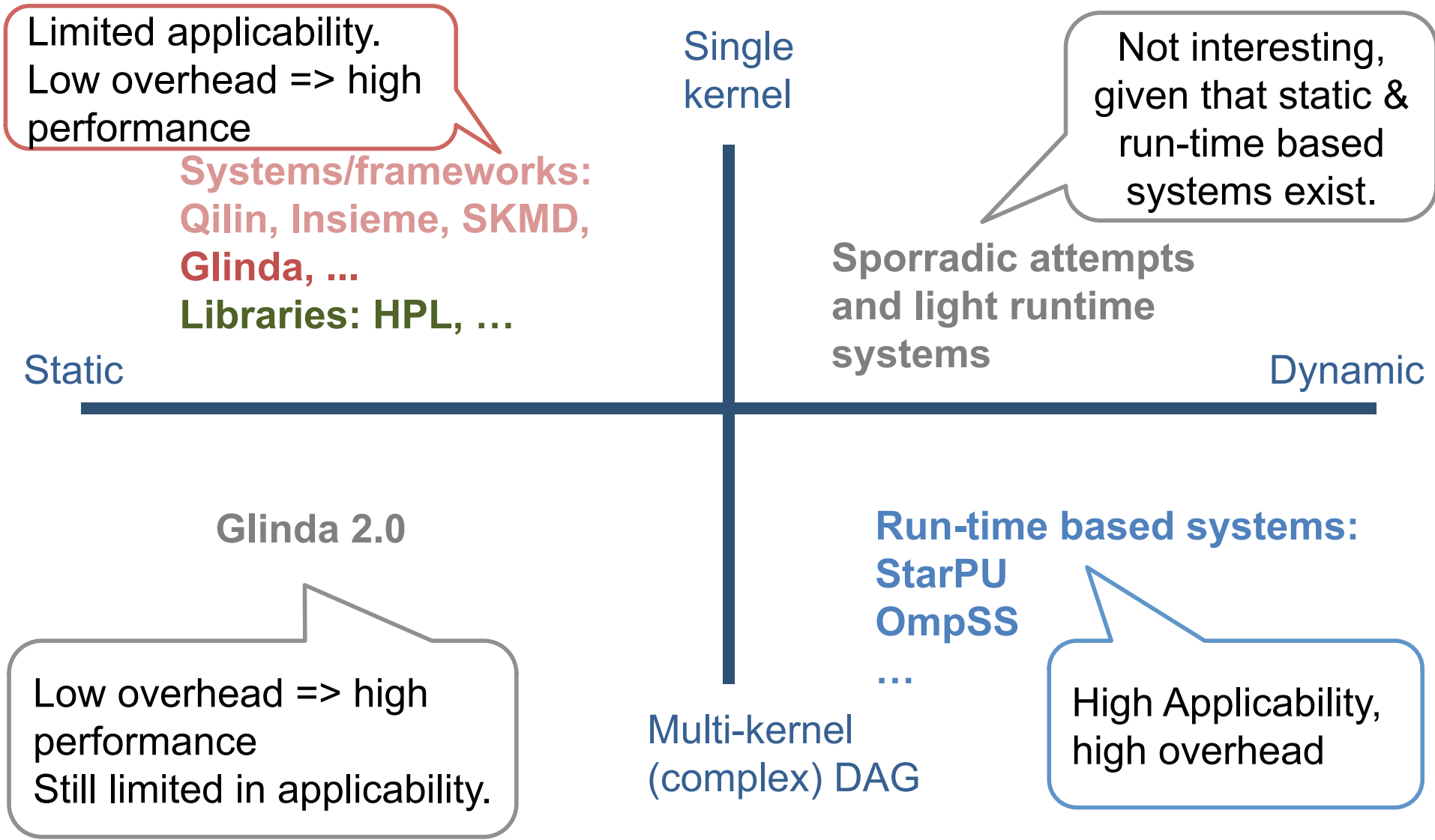
PART IV

Tools for heterogeneous processing

Heterogeneous Computing PMs



Heterogeneous Computing today



GLINDA

Computing static partitioning

Summary: Glinda

- Computes (close-to-)optimal partitioning
- Based on static partitioning
  - Single-kernel
  - Multi-kernel (with restrictions)

- No programming models attached
  - CUDA + OpenMP/TBB
  - OpenCL
  - … others => we propose HPL

HPL (HETEROGENEOUS PARALLEL LIBRARY)

Implementing static partitioning

Purpose

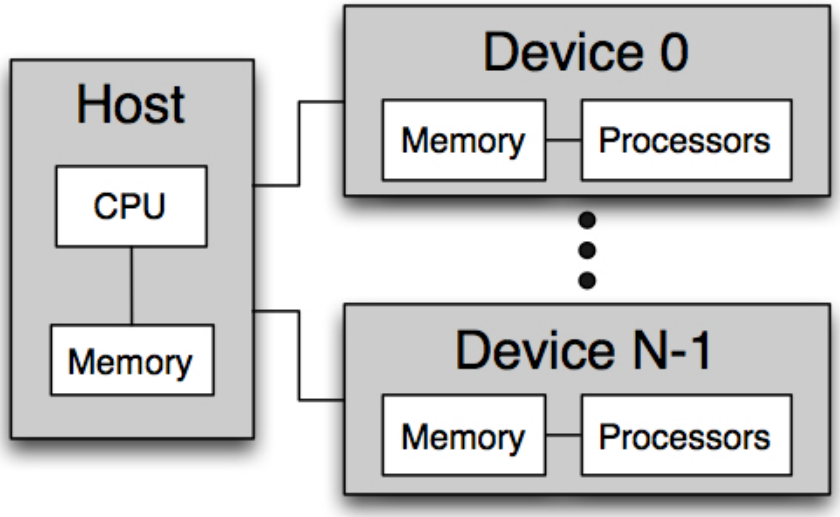
- Library to program heterogeneous systems
  - Expressive
  - Easy to use
  - Portable (uses OpenCL as backend)
  - No need to learn new languages
  - Good performance
  - Facilitate code space exploration

HPL Basics

- Key concepts
  - Kernels: functions that are evaluated in parallel by multiple threads on any device
    - Can be written either in standard OpenCL or in a language embedded in C++
  - Data types to express arrays and scalars that can be used in kernels and serial code
- Kernel code can be generated at runtime
  - Eases specialization, code space search

HPL hardware model

- Serial code runs in the host
- Parallel kernels can be run everywhere
- Processors can only access their device memory

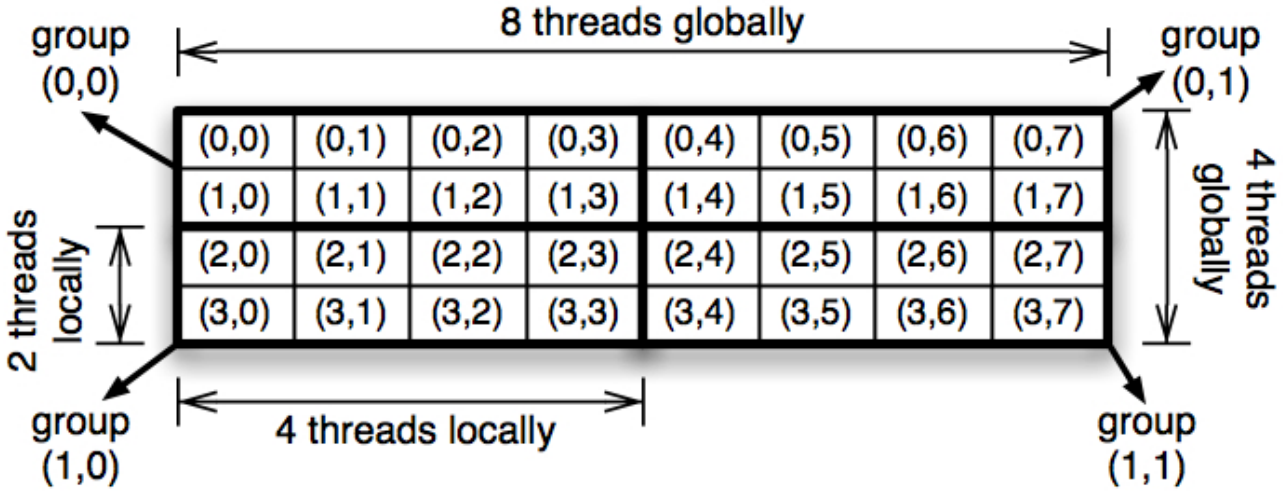


HPL memory model

- Four kinds of memory in devices:
  - Global: accessible for reading and writing by all the processors in a device
  - Local: fast scratchpad that can be shared by a group of threads
  - Constant: writeable by the host, but only readable for the device processors
  - Private: owned by each thread

Kernel evaluation index space

- Global domain required
  - Provides unique ID for each parallel thread
- Optional local domain
  - Threads in the same local domain can share scratchpad and synchronize with barriers



Arrays

- Array<type, ndims [,memFlag]> : ndims-dimensional array of elements of type type that can be used both in host code and kernels
  - Example: Array<float, 2> mx(100, 100);
  - memoryFlag can be Global, Local, Constant or Private. Appropriate default values.
- Scalars: expressed either with specialized types (Int, Float, Double, …) or with ndims=0

Array indexing

- In kernels
  - Only scalar indexing, using []: mx[i][j]
- In host code:
  - Scalar indexing, using (): mx(i,j)
  - Subarray selection using () and Ranges: mx(Range(0,9), Range(100,109))
    - Range(a,b) is inclusive (means[a,b])
    - Subarrays can be used as kernel arguments and in assignments: x(Range(a,b)) = y(Range(c,d))

Arrays (cont)

- Arrays are logical units, not physical buffers
  - Each Array is associated to buffers in different memories under the hood
- The runtime automatically keeps coherent these hidden copies
- Users just access each Array in the host and the kernels as a single entity, relying on sequential consistency
- No specification of buffers and data transfers!

HPL Kernels

- Can be written using a language embedded in C++ provided by HPL
  - The kernel code is generated at runtime
  - Allows to adapt it to the device, inputs, etc.
- Can be written using standard OpenCL C
  - Can reuse existing codes
- In both cases the kernel is associated to a C++ function whose parameters are those of the kernel

HPL language

- Control flow structs with underscore
  - if ⇨ if_; else ⇨ else_; for ⇨ for_ (with commas separating the arguments); …
- Predefined variables
  - idx, idy, idz ⇨ id for 1st, 2nd and 3rd dimension within global domain
  - lidx, lidy, lidz ⇨ idem for the local id
  - Similar ones for the group id, the sizes of the domains, etc.
- Predefined functions
  - E.g.: barrier: barrier between threads in a group

How to execute a kernel

- eval(f)(args) parallel evaluation of kernel on the arguments specified
  - Global domain defaults to the size of the first argument
  - eval(f).global(x,y,z).local(a,b,c).device(d) allows to specify the domain sizes and the device to use
- HPL has API to find devices and properties
- Several devices can be used in parallel
- If the CPU supports OpenCL, it is also a device

Example 1: SAXPY (Y=*a**X+Y)

```
#include "HPL.h"
using namespace HPL;

float v[1000];

Array<float, 1> x(1000); //host memory managed by HPL
Array<float, 1> y(1000, v); //v used as host memory for y

void saxpy(Array<float,1> y, Array<float,1> x, Float a) {
   y[idx] = a * x[idx] + y[idx];
}

int main() {
   float a; //C scalar types are allowed as eval arguments
   //the vectors and a are filled in with data (not shown)
   eval(saxpy)(y, x, a);
}
```

Example 2: A dot product

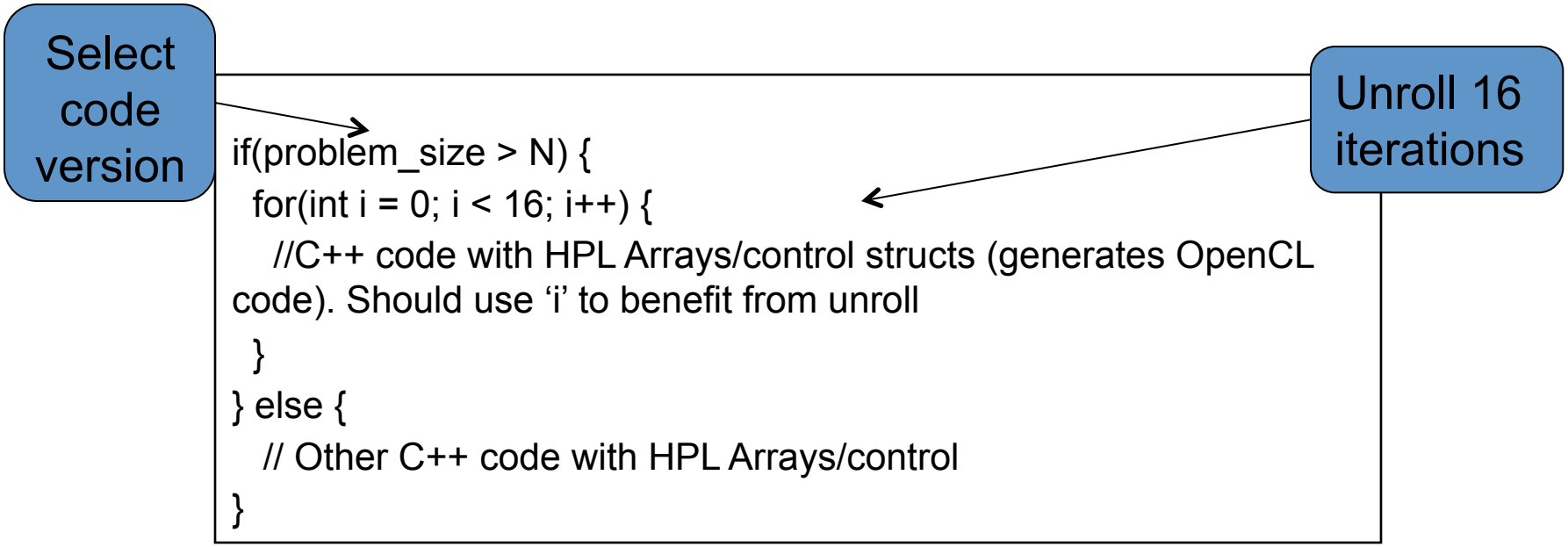
```
void dotp(Array<float,1> v1, Array<float,1> v2,
                                  Array<float, 1> pSums) {
  Array<float, 1, Local> sharedM(M);
  Int i;

  sharedM[lidx] =  v1[idx] * v2[idx];
  barrier(LOCAL);

  if_( lidx == 0 ) {
     for_( i = 0, i < M, i++ ) {
        pSums[gidx] += sharedM[i];
     }
  }
}
 . . .

eval(dotp).global(N).local(M)(v1, v2, pSums);
//reduces pSums in the host
result = pSums.reduce(std::plus<float>());
```

Kernel code generation

- The code is executed as regular C++
- HPL elements capture the code of the kernels, generating an AST
- Simple analyses are performed
  - e.g.: which arrays are read, written or both
    - Enables automated management of array transfers, minimizing them

Meta-programming

- Regular C++ can be interleaved in the kernels
  - It is not captured ⇨ it does not generate code
  - But it can control the code generated
    - Conditional/repetitive generation of code



Using OpenCL kernels

```
const char *opencl_kernel = TOSTRING(
  __kernel void saxpy(__global float *y, __global float *x, float a) {
    const size_t id = get_global_id(0);
    y[id] = a * x[id] + y[id];
  }
);

void kernel(InOut< Array<float, 1 >> y, In< Array<float, 1 >> x, Float a){}

. . .
Array<float, 1> y(1000), x(1000);
float a;
. . .
nativeHandle(kernel, “saxpy”, opencl_kernel);
eval(kernel)(y, x, a);
```

1. String with kernel

2. Handle with labels to indicate whether arguments are in, out or both

3. Associate handle, kernel name and string with its code

4. Enjoy

Dividing work among devices

- Three possibilities in HPL
  - By hand: choose subarrays to process in each device
  - Annotations: marking which dimension of the arguments to partition among the devices
  - Using an ExecutionPlan
    - Provide devices to use
    - Provide % of the problem to be run in each device or ask the ExecutionPlan to search for the best partitioning

HPL: Summary

- HPL facilitates programming heterogeneous systems using C++
- Average programmability improvement of 30-44% over OpenCL
- Typical performance overhead << 5%
- Available with manual under GPL license at http://hpl.des.udc.es

Ongoing work

- Enhancements to provide fault-tolerance to heterogeneous applications
  - To be published soon in a prestigious journal
- Extension to easily program heterogeneous clusters
  - Works great. Ready for submission
- Just-in-time compiler for adaptive codes
  - Polishing

Most relevant publications

- Basics: M. Viñas, Z. Bozkus, B.B. Fraguela. ‘Exploiting heterogeneous parallelism with the Heterogeneous Programming Library’. J. Parallel and Distributed Computing, 73(12):1627-1638. 2013

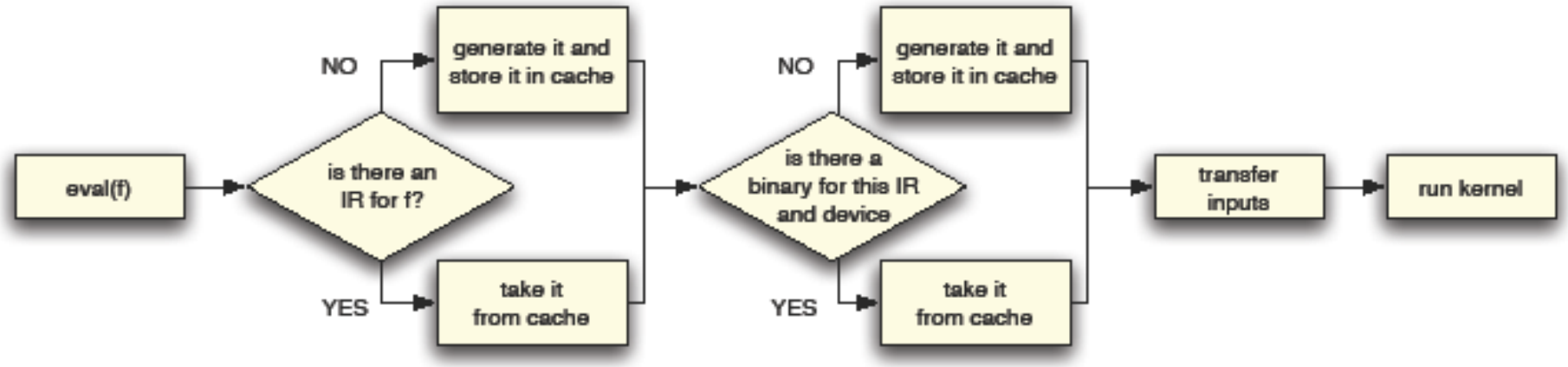
- Kernel code exploration: J.F. Fabeiro, D. Andrade, B.B. Fraguela. ‘Writing a performance-portable matrix multiplication’. Parallel Computing, 52:65-77. 2016

- Partitioning work on devices: M. Viñas, B.B. Fraguela, D. Andrade, R. Doallo. ‘High Productivity Multi-device Exploitation with the Heterogeneous Programming Library’. J. Parallel and Distributed Computing, 101:51-68. 2017

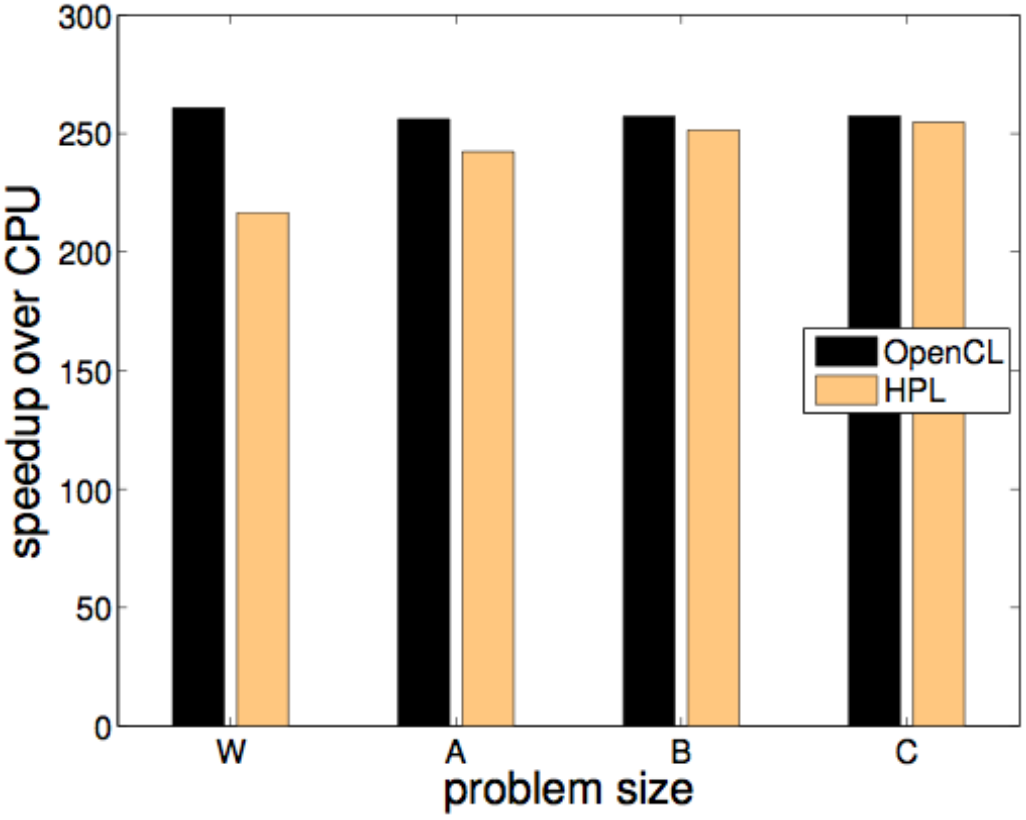
Templates

```
  template<typename T>
  void add(Array<T, 2> a, Array<T, 2> b, Array<T, 2> c) {
     a[idx][idy] =  b[idx][idy] + c[idx][idy];
  }

 . . .

  Array<float, 2> av(N,N), bv(N,N), cv(N,N);
  Array<int, 2> avi(M,M), bvi(M,M), cvi(M,M);

//We use addv to add floats
  eval(addv<float>)(cv, av, bv);

//We use addv to add ints
  eval(addv<int>)(cvi, avi, bvi);
```

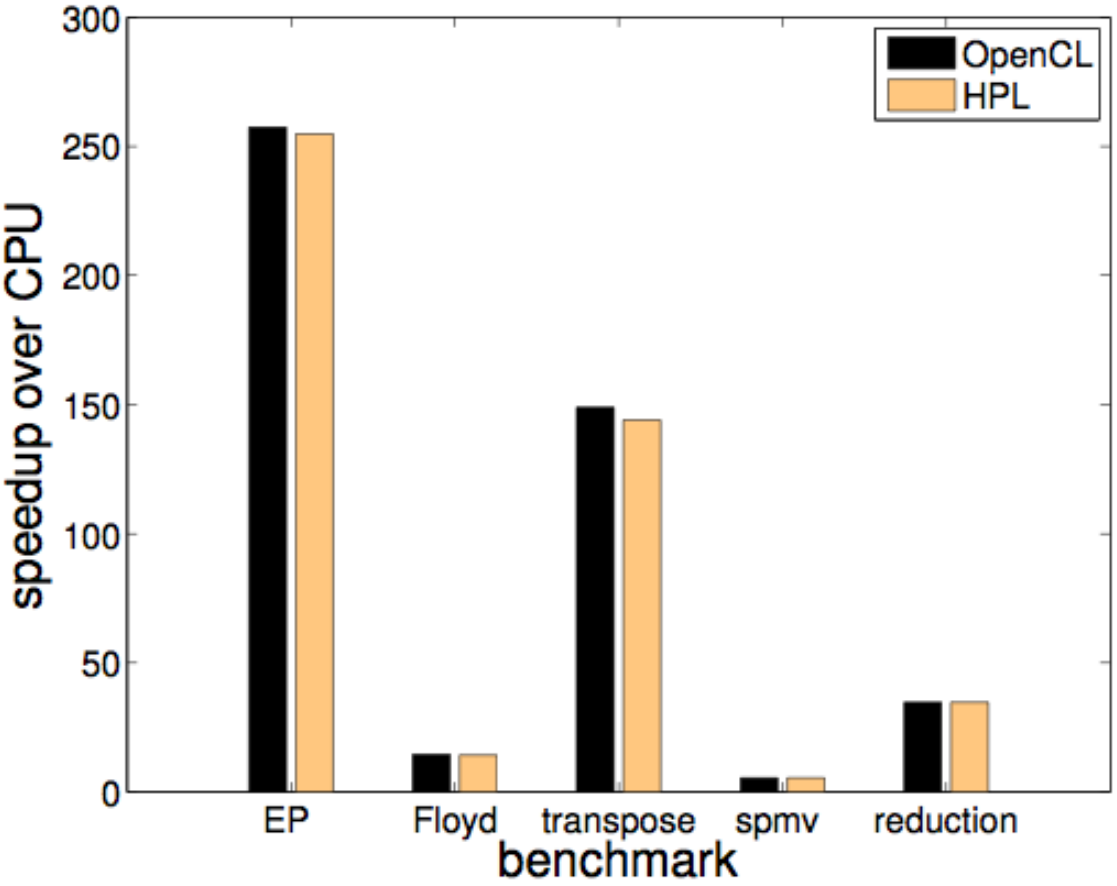
Kernel invocation process



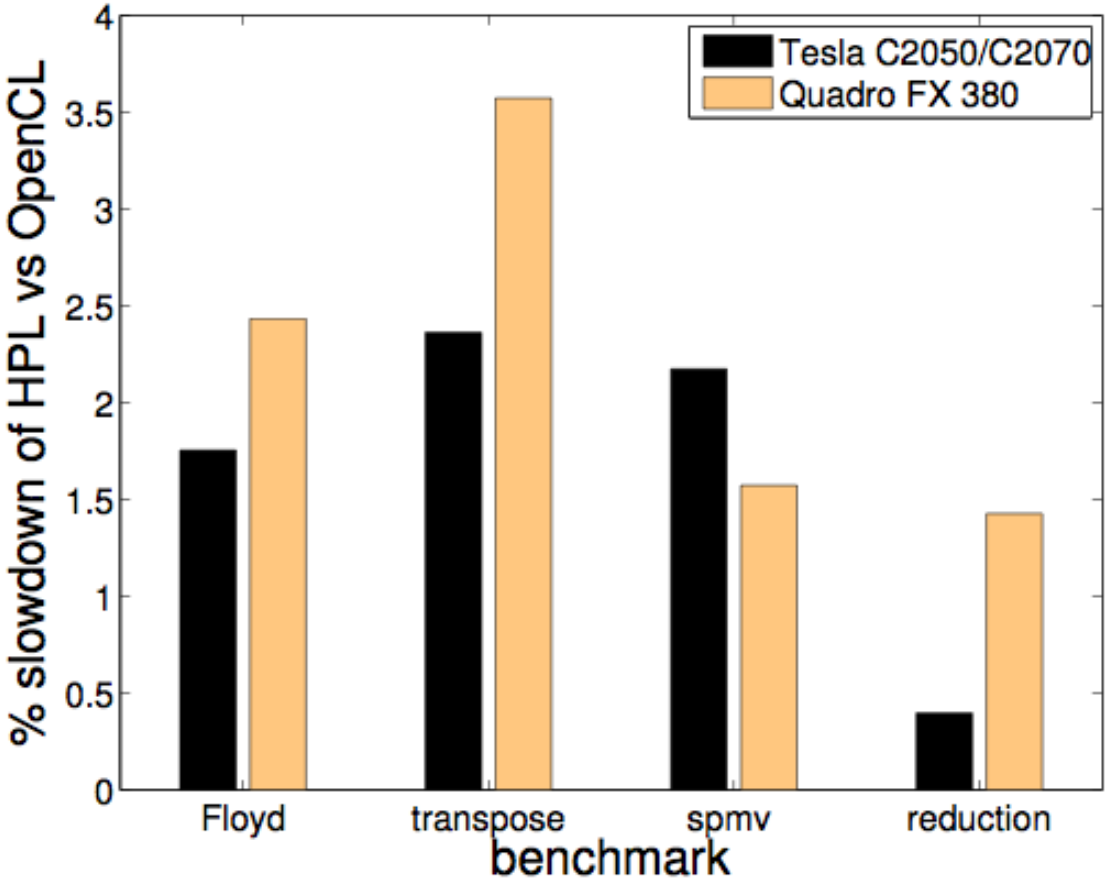
Speedup of GPU EP with respect to CPU sequential



Speedups in GPU with respect to CPU execution



Overhead of HPL with respect to OpenCL



STARPU

Task parallelism and smart scheduling

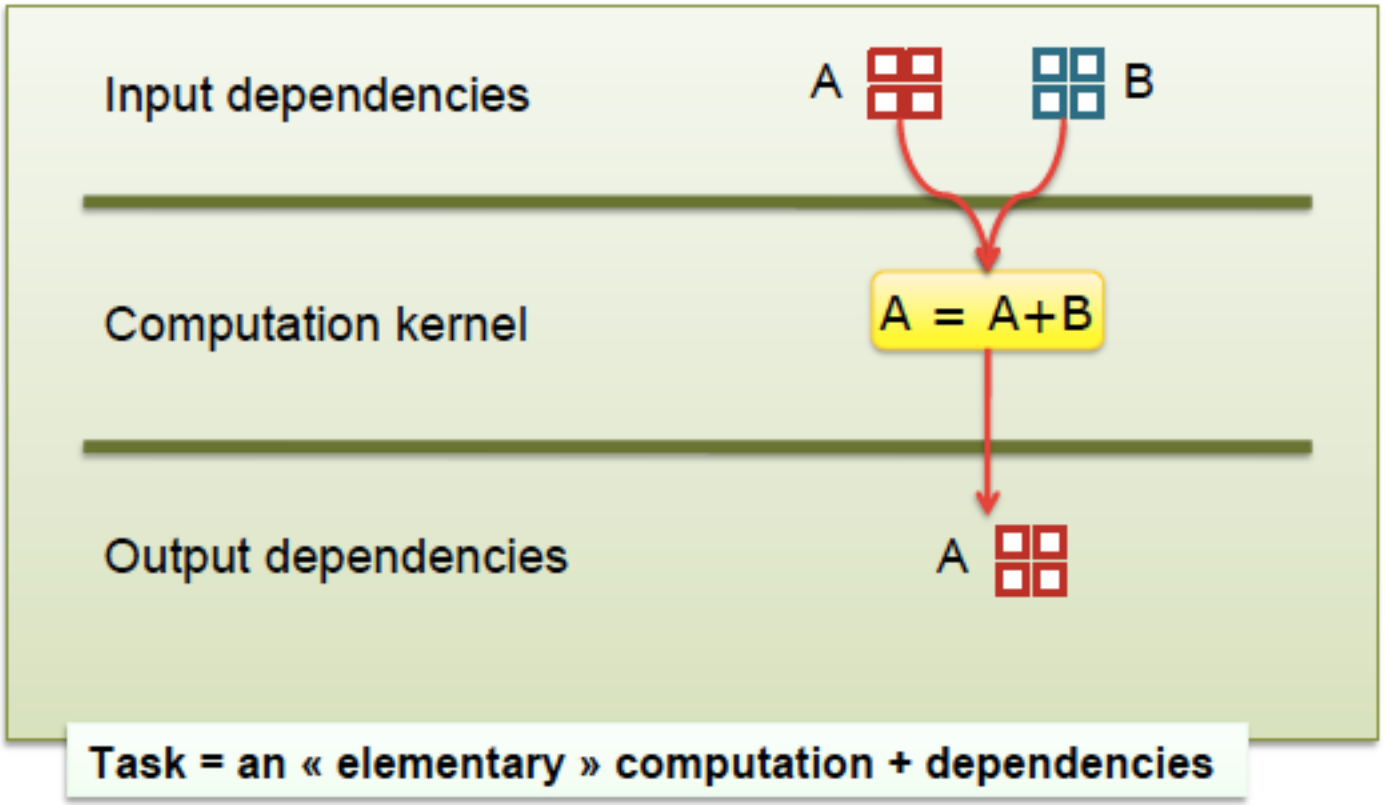
Heterogeneous Task Scheduling

Goal of StarPU: schedule a task-parallel application on a platform equipped with accelerators:

- Adapt to heterogeneity
  - Decide about tasks to offload
  - Decide about tasks to keep on CPU
- Communicate with discrete accelerator board(s)
  - Send computation requests
  - Send data to be processed
  - Fetch results back
- Adapt for performance
  - Decide about worthiness

Task parallelism

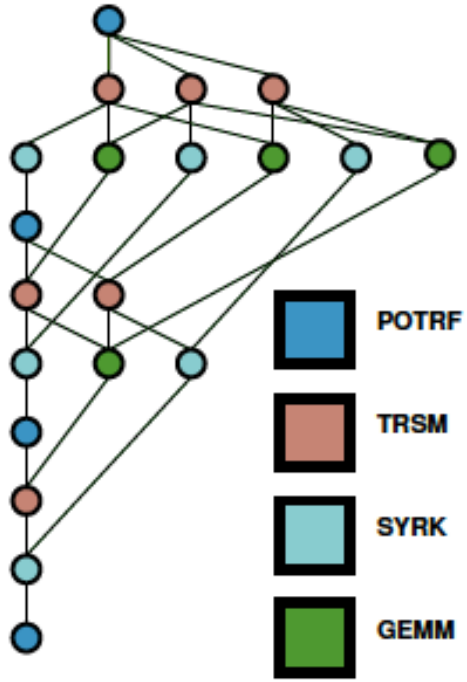
- Input dependencies
- Computation kernel
- Output dependencies



StarPU programming model

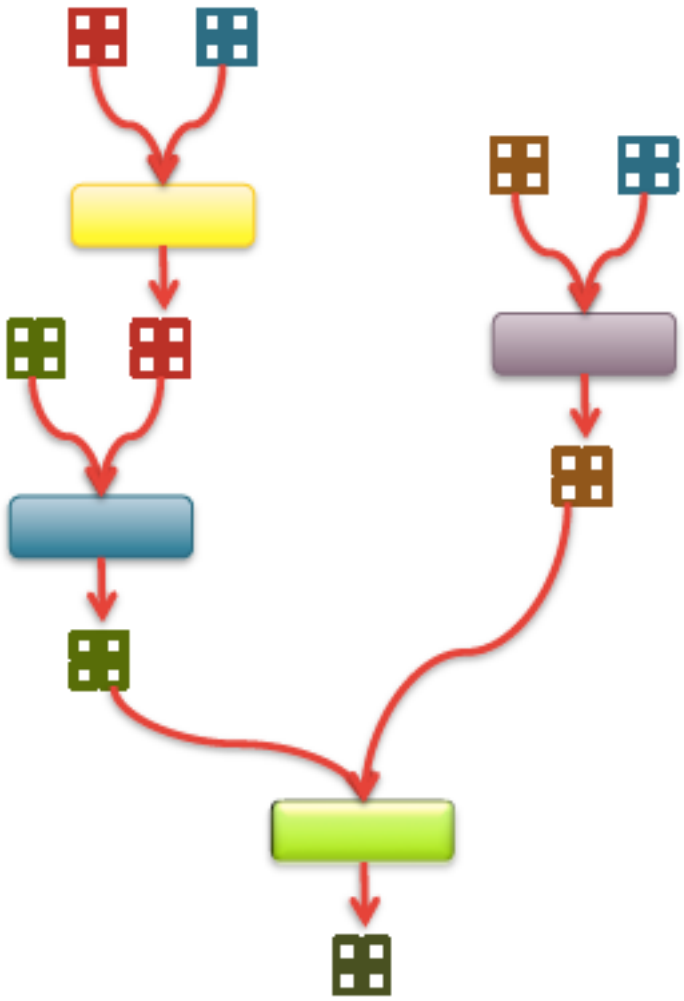
- Express parallelism. . .
- . . . using the natural program flow
- Submit tasks in the sequential flow of the program. . .
- . . . then let the runtime schedule the tasks asynchronously

```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW, A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
     for (k = j+1; k < i; k++)
       GEMM (RW,A[i][k],R,A[i][j], R,A[k][j]
  }
}
__wait__();
```



Tasks

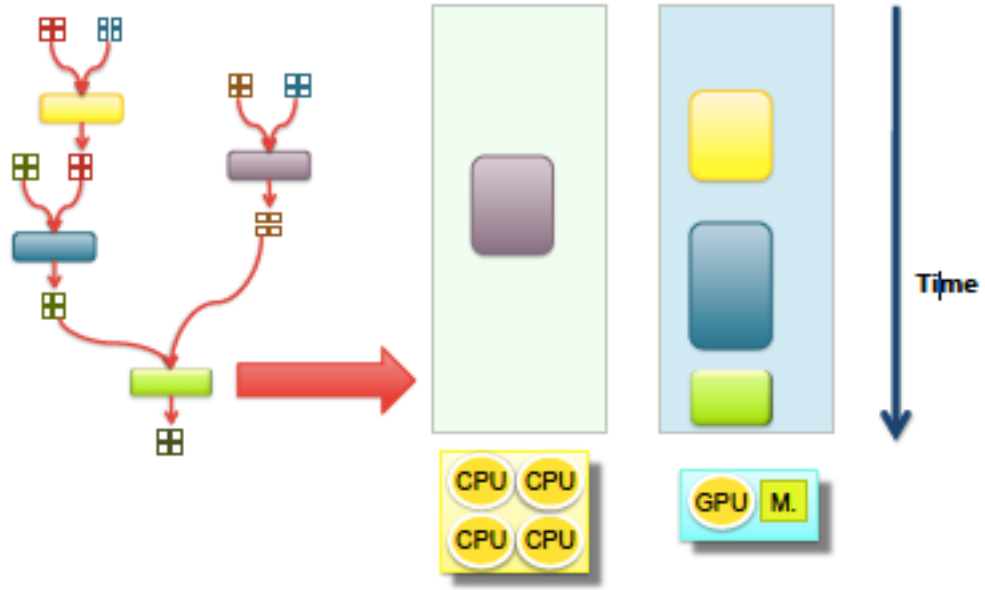
- Task Relationships
- Abstract Application Structure
- Directed Acyclic Graph (DAG)



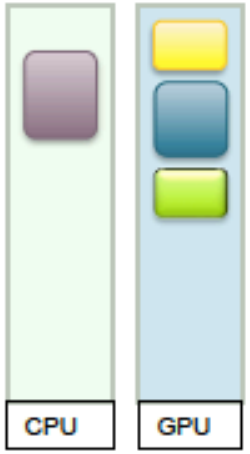
StarPU Execution model

Task Scheduling:

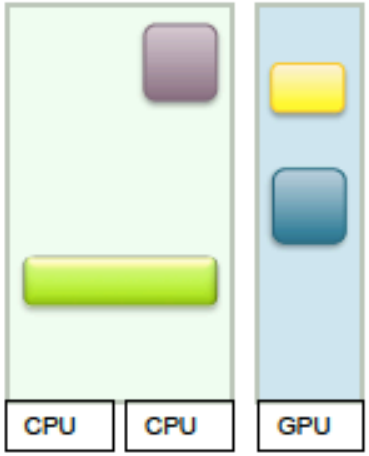
- Mapping the graph of tasks (DAG) on the hardware
- Allocating computing resources
- Enforcing dependency constraints
- Handling data transfers

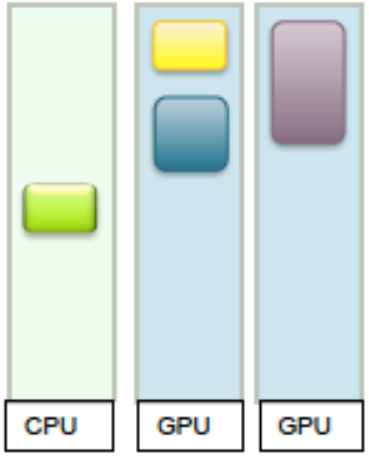


Single DAG, multiple schedules



Multicore CPU





Multi-GPUs

Terminology

- Codelet
  - . . . relates an abstract computation kernel to its implementation(s)
  - . . . can be instantiated into one or more tasks
  - . . . defines characteristics common to a set of tasks
- Task
  - . . . is an instantiation of a Codelet
  - . . . atomically executes a kernel from its beginning to its end
  - . . . receives some input
  - . . . produces some output
- Data handle
  - . . . designates a piece of data managed by StarPU
  - . . . is typed (vector, matrix, etc.)
  - . . . can be passed as input/output for a Task

API

- Initializing/Ending a StarPU session
- Declaring a codelet
- Declaring and Managing Data
- Writing a Kernel Function
- Submitting a task
- Waiting for submitted tasks
- Team

Programming

- Scaling a vector

```
1  float factor = 3.14;
2  float vector[NX];
3  starpu_data_handle_t vector_handle;
4
5  /* ... fill vector ... */
6
7  starpu_vector_data_register(&vector_handle, 0,
8                              (uintptr_t)vector, NX, sizeof(vector[0]));
9
10 starpu_task_insert(
11                  &scal_cl,
12                  STARPU_RW, vector_handle,
13                  STARPU_VALUE, &factor, sizeof(factor),
14                  0);
15
16 starpu_task_wait_for_all();
17 starpu_data_unregister(vector_handle);
18
19 /* ... display vector ... */
```

Heterogeneity at kernel level

- Heterogeneity: Device Kernels
- Extending a codelet to handle heterogeneous platforms
- Multiple kernel implementations for a CPU
- – SSE, AVX, ... optimized kernels
- Kernels implementations for accelerator devices
- – OpenCL, NVidia Cuda kernels

```
1 struct starpu_codelet scal_cl = {
2     .cpu_func   = { scal_cpu_func,
3             scal_sse_cpu_func, scal_avx_cpu_func, NULL },
4     .opencl_func   = { scal_cpu_opencl, NULL },
5     .cuda_func   = { scal_cpu_cuda, NULL },
6     .nbuffers = 1,
7     .modes = { STARPU_RW },
8 };
```

A kernel implementation

```
1  static __global__ void vector_mult_cuda(unsigned n,
2                                          float *vector, float factor){
3      unsigned i = blockIdx.x*blockDim.x + threadIdx.x;
4      if (i < n)
5          vector[i] *= factor;
6  }
7
8  extern "C" void scal_cuda_func(void *buffers[], void *cl_arg){
9      struct starpu_vector_interface *vector_handle = buffers[0];
10     unsigned n    = STARPU_VECTOR_GET_NX(vector_handle);
11     float *vector = STARPU_VECTOR_GET_PTR(vector_handle);
12     float *ptr_factor = cl_arg;
13
14     unsigned threads_per_block = 64;
15     unsigned nblocks = (n+threads_per_block-1)/threads_per_block;
16
17     vector_mult_cuda<<<nblocks,threads_per_block,0,
18         starpu_cuda_get_local_stream()>>>(n,vector,*ptr_factor);
19 }
```

StarPU scheduling

Basic policies:

- The Eager Scheduler : FCFS
- The Work Stealing Scheduler : Load Balancing

“Informed” policies

- The Prio Scheduler – based on task priorities
- The Deque Model Scheduler – based on HEFT
  - Uses codelet performance models
    - History-based
    - Statistical (regression)

To set scheduler: export STARPU_SCHED = prio/dm/…

StarPU data management

- Handles dependencies
- Handles scheduling
- Handles data consistency (MSI)

Data Transfer Cost Modelling

- Discrete accelerators
  - CPU to GPU transfers are expensive
  - Weigh data transfer cost vs kernel offload benefit
- Transfer cost modelling
  - Bus calibration
    - Can differ even for identical devices
    - Platform's topology
- Data-transfer aware scheduling
  - Deque Model Data Aware (dmda) scheduling policy variants
  - Tunable data transfer cost bias
    - Locality vs. load balancing

Data prefetching & partitioning

- Attempts to predict data to be used => prefetch
  - Manual
- Supports data partitioning
  - As close as it gets to static partitioning

Data partitioning

- Support for data parallelism
  - Data can be accessed at different granularity levels in different phases

StarPU: summary

- Implement the sequential task flow programming model
- Map computations on heterogeneous computing units
- Handles data management
  - Transfers, locality, prefetching, scheduling …

- Programming Model
  - Tasks + Data + Dependencies
    - Task to Task
    - Task to Data
  - Application Programming Interface (Library)

- Runtime System
  - Heterogeneous Task scheduling
    - User-selectable policy

OMPSS

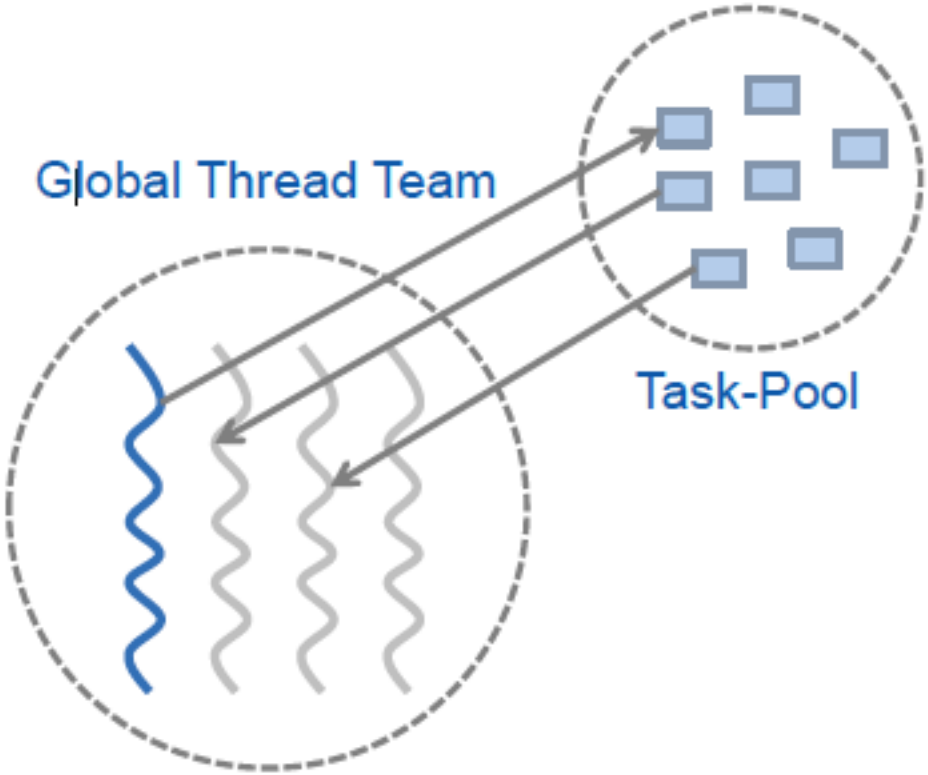
An OpenMP-like task-parallel heterogeneous model

Introduction

- Parallel Programming Model
  - Build on existing standard: OpenMP
  - Directive based to keep a serial version
  - Targeting: SMP, clusters and accelerator devices
  - Developed at Barcelona Supercomputing Center (BSC)
    - Mercurium source-to-source compiler
    - Nanos++ runtime system
- Where does it come from (a bit of history)
  - BSC had two working lines for several years
    - OpenMP Extensions: Dynamic Sections, OpenMP Tasking prototype
    - StarSs: Asynchronous Task Parallelism Ideas
  - OmpSs is folds them together

OmpSs Execution model

- Thread-pool model
- All threads created on startup
  - One of them starts executing main
- All get work from a task pool
  - And can generate new work



Memory model

- The programmer sees a single naming space
- For the runtime there are different scenarios:
  - Pure SMP
    - Single address space
  - Distributed/heterogeneous (GPUs, clusters, ...):
    - Multiple address spaces exist
    - Multiple copies of the same variable may exist
    - Data consistency ensured by the implementation

Main unit: OpenMP task

- A task is a deferrable work with some data attached

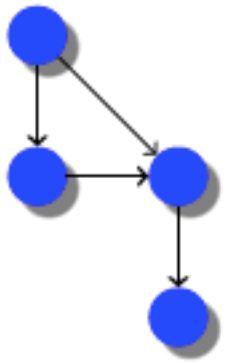
```
#pragma omp task [clauses]
    code-block
```

- A task directive can be applied to a function declaration or definition
  - Calls to the function => task spawning points

Dependence clauses

- Express data dependencies (evaluated at runtime):
  - input
  - output
  - Inout
- Used for optimization purposes, too
  - Scheduling: data reuse, critical path, ...
  - Data prefetching

```
#pragma omp task output(x)
  x = 5;
#pragma omp task input(x)
  printf ("%d\n", x);
#pragma omp task inout(x)
  x++;
#pragma omp task input(x)
  printf ("%d\n", x);
```



Extended expressions

- Dependency clauses are extended to allow:
  - Array sections: reference a range of array elements
  - Shaping expressions: convert pointers to arrays with size

```
int a [100];
int b = &a[50];

#pragma omp task input(a[10:20]) // Elements from 10 to 20
...
#pragma omp task input(b[10:20]) // Also allowed in pointers
...
#pragma omp task input(a[10;10]) // Alternative form
...
#pragma omp task input([50]b) // References an array of 50 positions
```

Heterogeneity support

- Directive for device-specific information:

```
#pragma omp target [clauses]
```

- Clauses:
  - `device` => specify a device(s) for the task (smp,cuda)
  - `copy_in, copy_out, copy_inout` => computation data
    - Extended expressions also allowed
  - `copy_deps` => copy dependencies
  - `implements` => may specify alternative implementation

Example

```
#pragma target device(smp) copy_deps
#pragma omp task_input ([N] c) output([N] b)
void scale_task(double *b, double *c, double s, int N) {
  int j;
  for (j=0; j<BSIZE; j++) b[j] = s * c [j];
}

#pragma omp target device(cuda) implements(scale_task)
void scale_task_cuda(double *b, double *c, double s, int N){
  const int threadsPerBlock = 128;
  dim3 dimBlock ( threadsPerBlock , 1 , 1 ) ;
  dim3 dimGrid ( si ze / threadsPerBlock +1) ;
  scale_kernel <<<dimGrid,dimBlock>>>(N,1,b,c,s) ;
}
```

Heterogeneity support

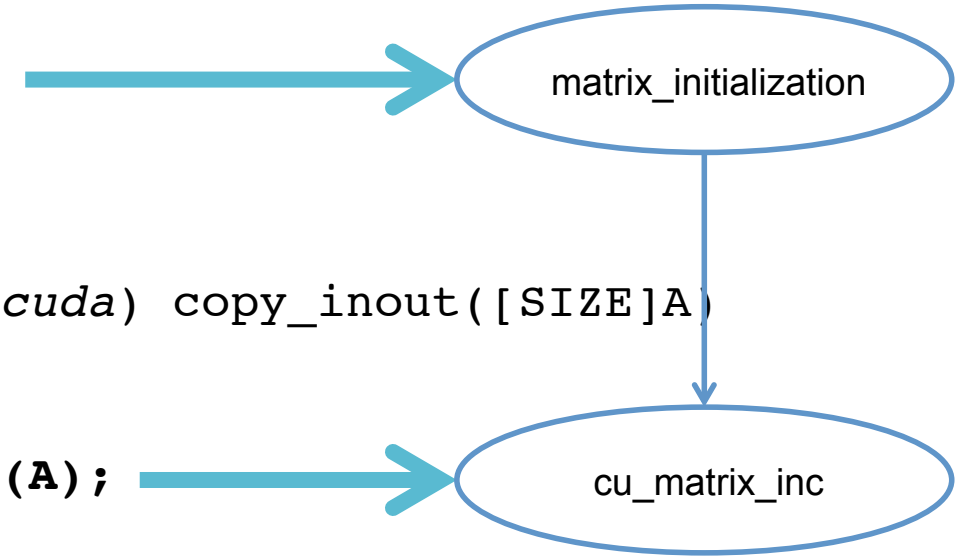
- Compiler tool-chain enables heterogeneous computing
  - Working with multiple devices architectures
  - Multiple implementations of the same function

```
int A[SIZE];

#pragma omp target device (smp) copy_out([SIZE] A)
#pragma omp task
  matrix_initialization(A);
#pragma omp taskwait

#pragma omp target device (cuda) copy_inout([SIZE]A)
#pragma omp task
{
  cu_matrix_inc<<<Size,1>>>(A);
}
```

Asynchronous data-flow execution

- Dependence clauses allow to remove synch directives
  - Runtime library computes dependences

```
int A[SIZE];

#pragma omp target device (smp) copy_out([SIZE] A)
#pragma omp task out(A)
  matrix_initialization(A);

#pragma omp taskwait

#pragma omp target device (cuda) copy_inout([SIZE]A)
#pragma omp task inout(A)
{
  cu_matrix_inc<<<Size,1>>>(A);
}
```



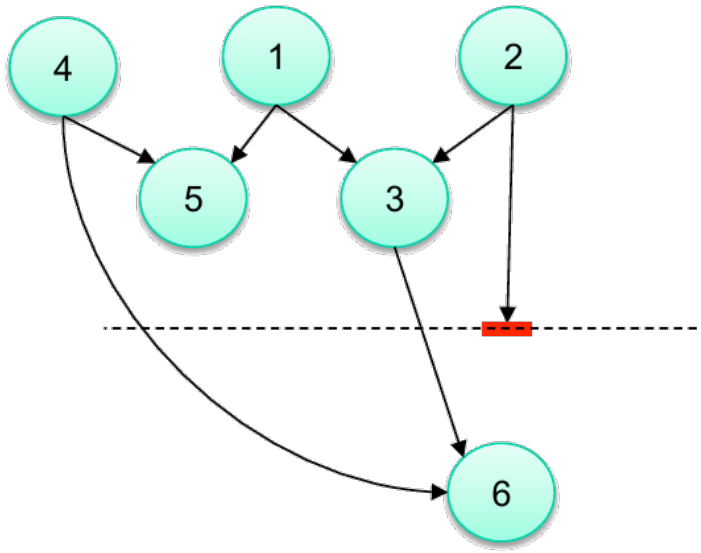
Synchronization

- Using “taskwait”:

```
#pragma omp taskwait [on (expression)]
```

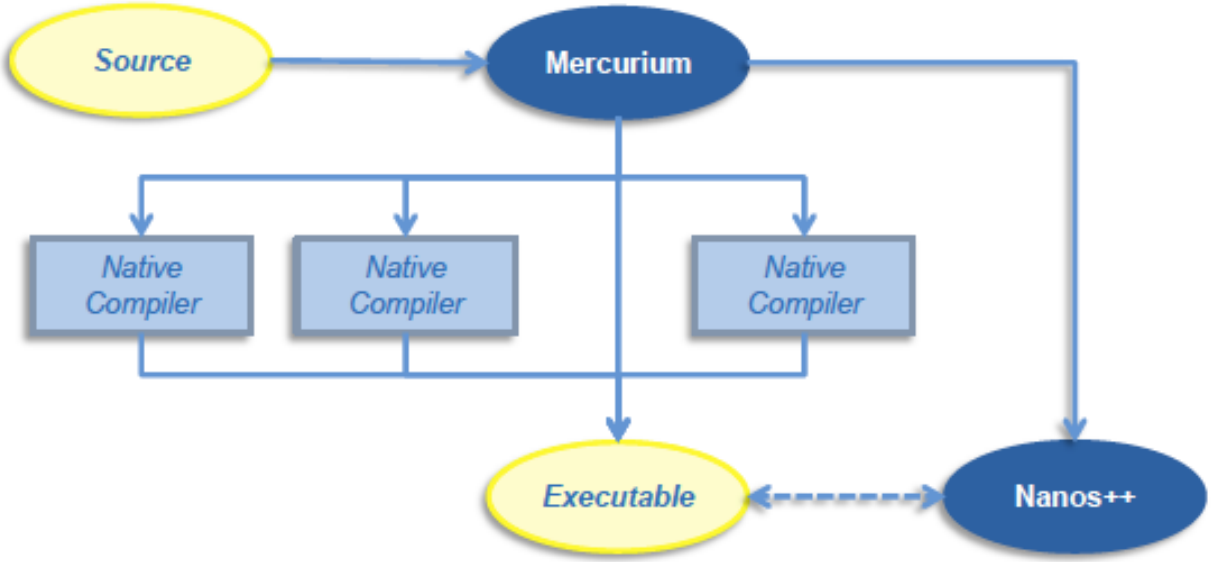
  - Suspends current task until all child tasks are completed
- The `on` clause => wait on task to produce certain data
  - Suspends the encountering task until data is available

```
dgemm(A,B,C); // 1
dgemm(D,E,F); // 2
dgemm(C,F,G); // 3
dgemm(A,D,H); // 4
dgemm(C,H,I); // 5
#pragma omp taskwait on(F)
dgemm(H,G,C); // 6
#pragma omp taskwait
print ("result C”, C) ;
```



Implementation

- Mercurium Compiler
  - Source to source compiler: from OmpSs directives to runtime calls
- Nanos++ RTL
  - Implement runtime services: create/execute tasks, synchronization, dependencies, memory consistency,…



Run-time features

- Schedulers (non-comprehensive list)
  - Breadth-first:
    - Global FCFS queue for tasks ready to execute
  - Distributed breadth-first:
    - multiple FCFS queues, one per thread
    - When local queue is empty proceed work stealing
  - Work-first scheduler:
    - Multiple FCFS queue, one per thread
    - FIFO access locally, LIFO access on steals
- Priorities
  - Supports **task priorities** to tune the scheduling and execution order
- Throttling
  - Supports policies for task creation and/or execution
    - E.g., Immediate vs. asynchronous

OmpSs with GPUs : CUDA

- C/C++ files (usually .c or .cpp) = host code
- CUDA files (.cu) = kernel code

```
/* cuda-kernels.cu */
extern "C" {  // specify extern "C" to call from C code
__global__ void init(int n, int *x) {CUDA code here}
__global__ void increment(int n, int *x) {CUDA code here}
} /* extern "C" */
```

```
#pragma omp target device(cuda) copy_deps ndrange(1, n, 1)
#pragma omp task out(x[0 : n-1])
  __global__ void init(int n, int *x);
#pragma omp target device(cuda) copy_deps ndrange(1, n, 1)
#pragma omp task inout(x[0 : n-1])
  __global__ void increment(int n, int *x);

init(10, x); increment(10, x);
#pragma omp taskwait
```

OmpSs with GPUs : OpenCL

- C/C++ files (usually .c or .cpp) = host code
- OpenCL files (.cl) = kernel code

```
/* cuda-kernels.cu */
extern "C" {  // specify extern "C" to call from C code
__kernel void init(int n, int __global *x)       {OCL code}
__kernel void increment(int n, int __blobal *x) {OCL code}
} /* extern "C" */
```

```
#pragma omp target device(opencl) copy_deps ndrange(1,n,8) \
  file(ocl_kernels.cl)
#pragma omp task out(x[0 : n-1])
  void init(int n, int *x);
#pragma omp target device(cuda) copy_deps ndrange(1, n, 1)
#pragma omp task inout(x[0 : n-1])
  void increment(int n, int *x);
…
init(10, x); increment(10, x); …
```

OmpSs: Summary

- Easy-to-use
  - OpenMP model
  - Task-based
- No embedded support for data-parallelism
  - Has to be “emulated” by tasks
- Run-time optimization is their core research
  - User kept “out-of-the-loop”

- WiP: Glinda + OmpSS

CASHMERE + MCL

A divide-and-conquer approach

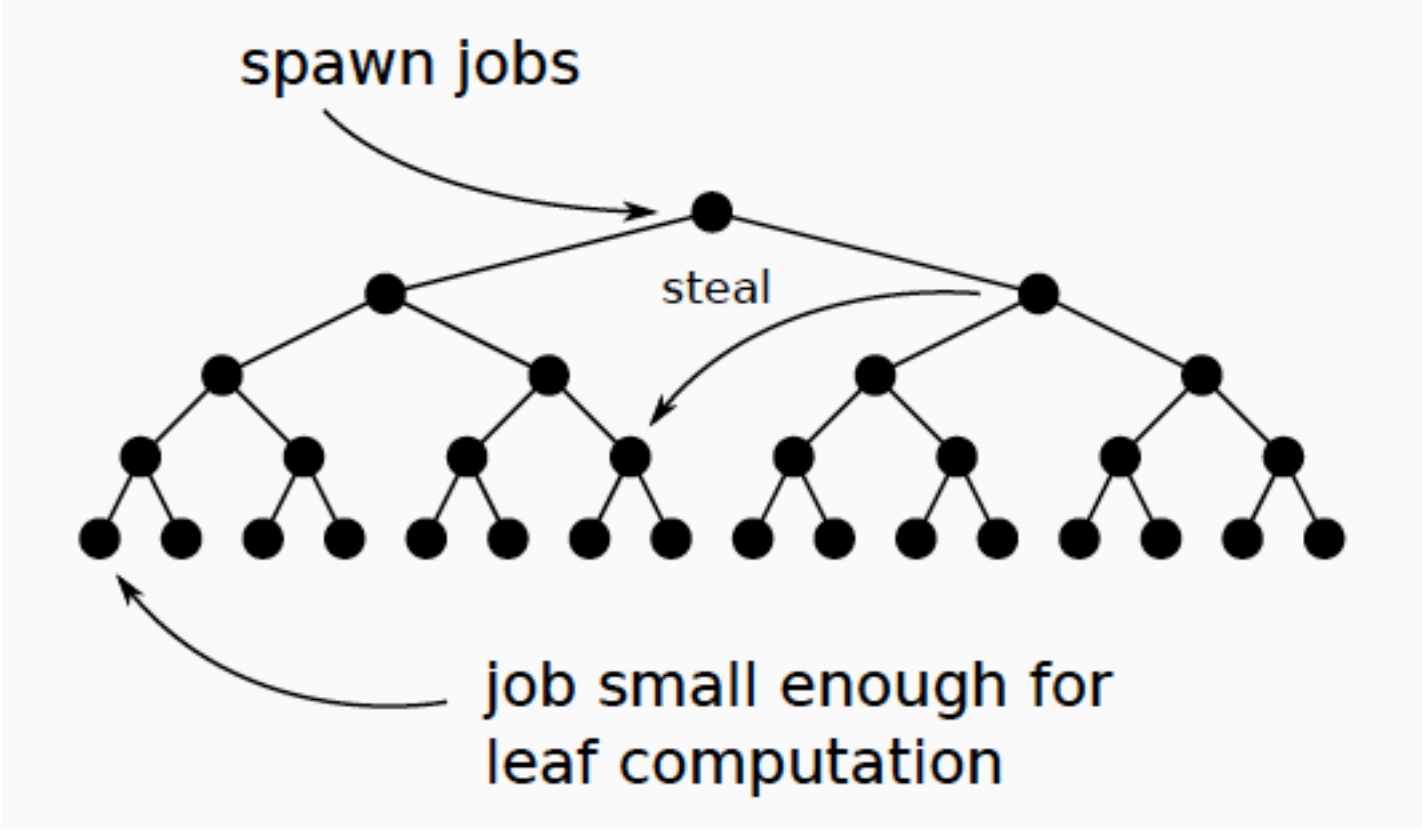
Cashmere* [C-1]

- Dynamic runtime support for distributed heterogeneous clusters
  - Specific for divide-and-conquer
- Provides scalability on heterogeneous many-core clusters:
  - scalability in performance
  - scalability in optimizing kernels
- Integrates two frameworks:
  - Satin [C-2]
  - MCL [3]

* What's in a name?
  Cilk ➔ Satin ➔ Cashmere
  Higher quality fabric with fine threads

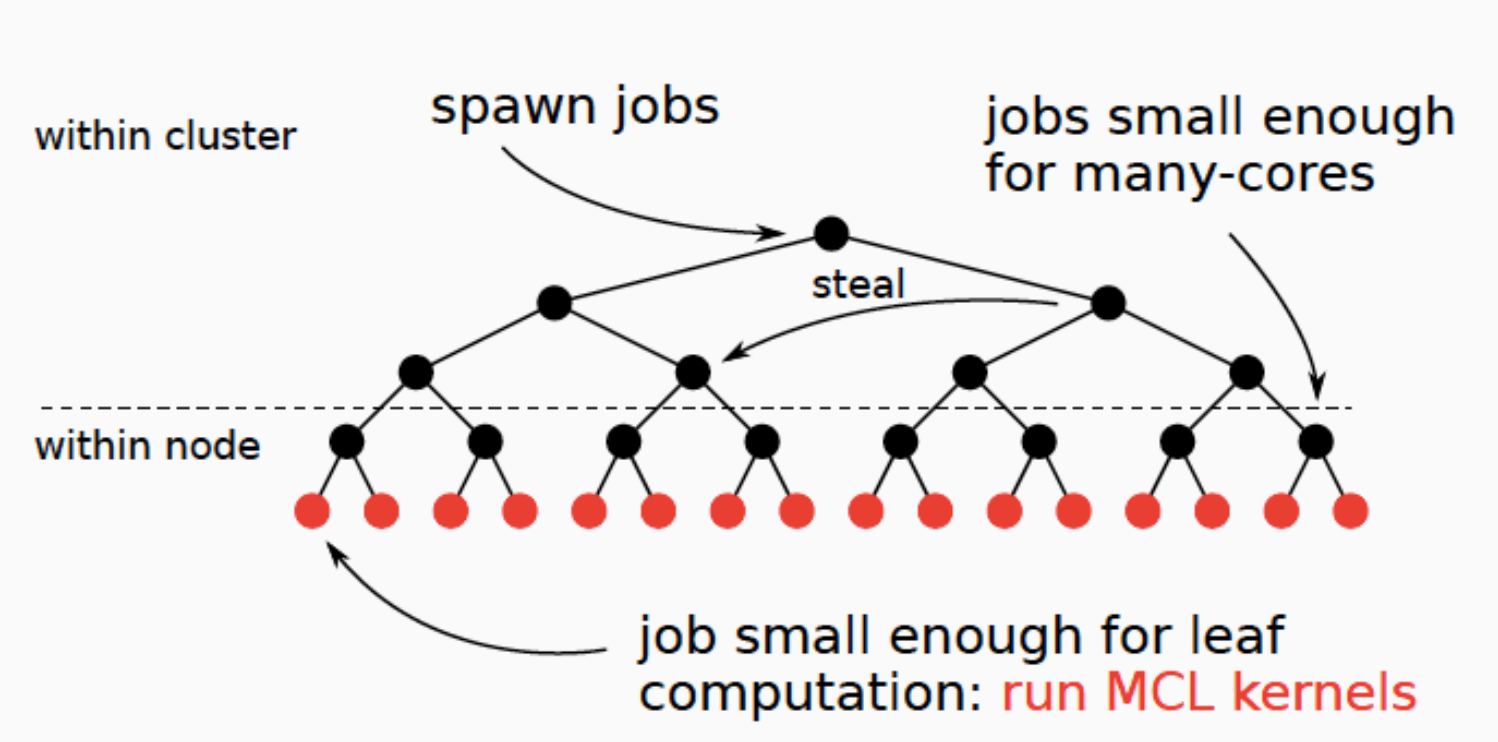
Satin

- Divide-and-conquer
  - automatic load balancing due to job stealing



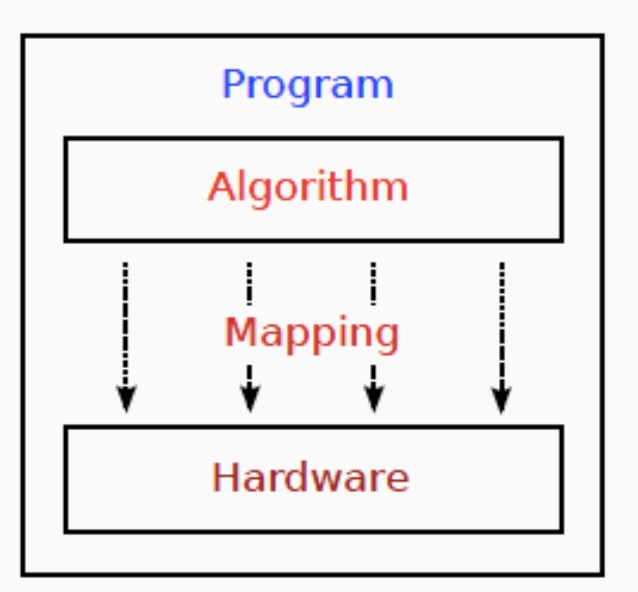
Cashmere

- Two-level divide-and-conquer
  - Cluster: load balancing with job stealing
  - Node: multiple devices per node
    - overlap data-transfers with kernel execution



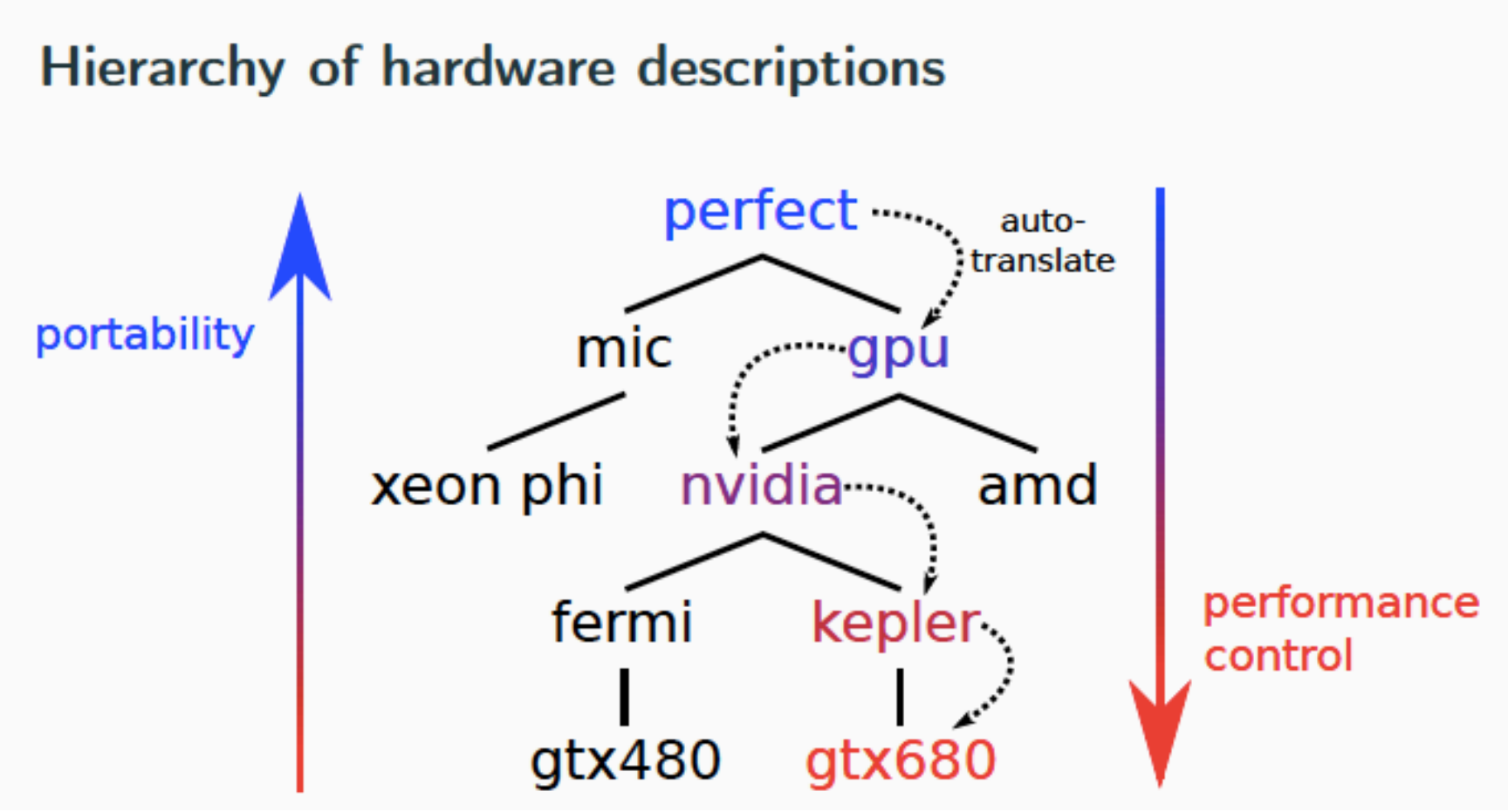
Many Core Levels (MCL)

- A program is an algorithm mapped to hardware



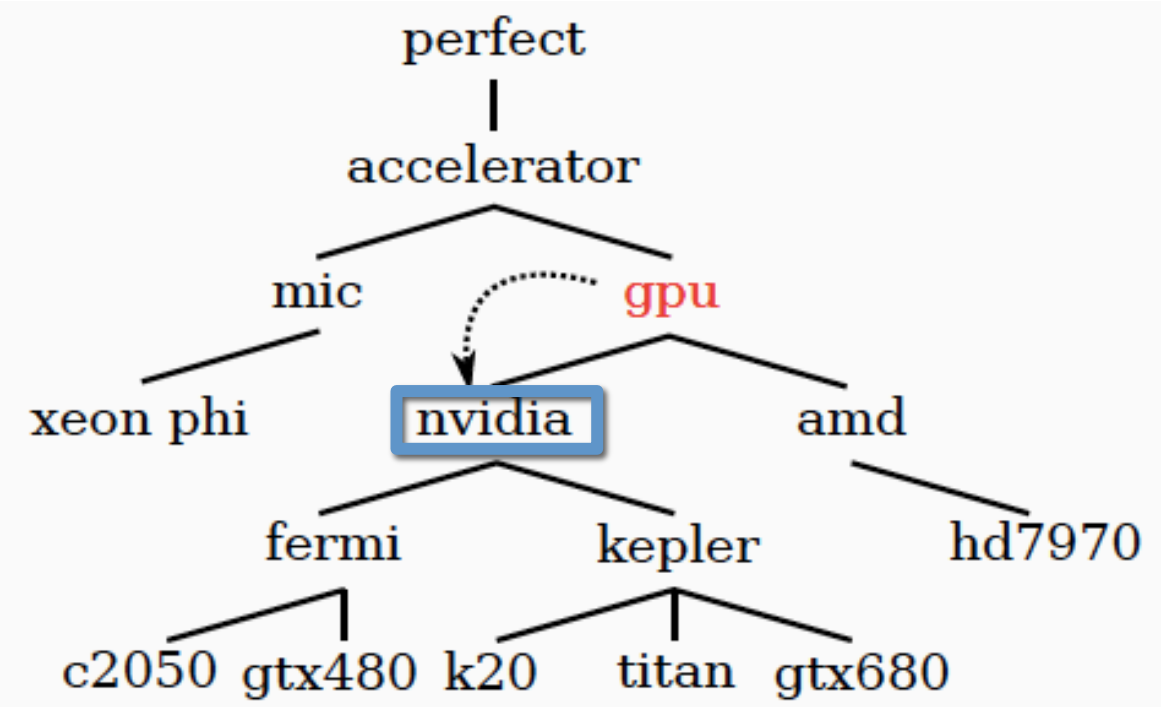
- Write kernels in MCL
- Receive performance feedback

Multiple Abstraction Layers



Performance feedback

- Based on knowledge of the hardware



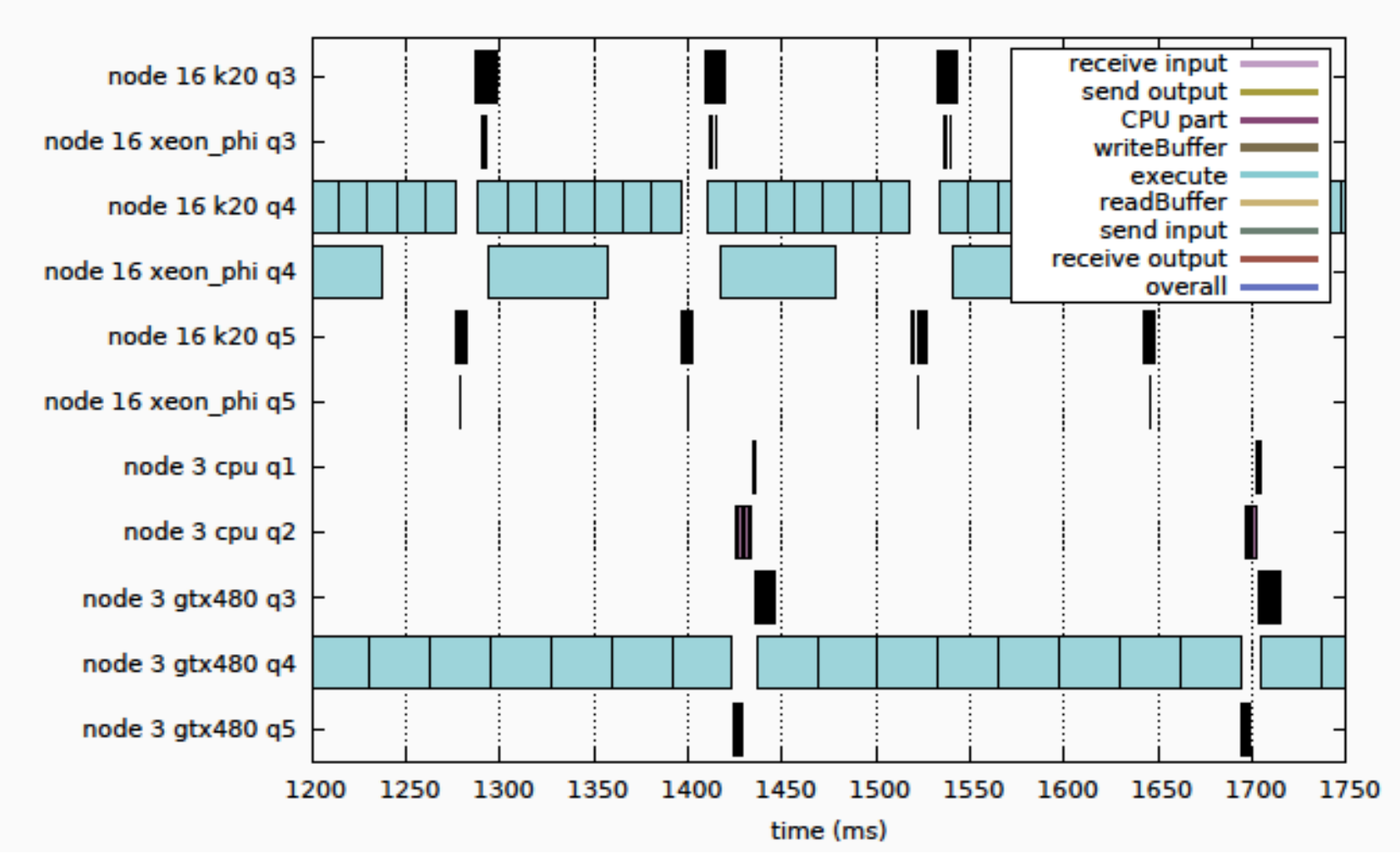
Example feedback:
Using 1/8 blocks per smp. Reduce the amount of shared memory used by storing/ loading shared memory in phases.

Programmer's interface

- The MCL compiler generates OpenCL code (node-level) and Cashmere code (cluster-level)
  - Based on divide-and-conquer => runtime system is much lighter than StarPU or OmpSs => lower overhead
- Calling a kernel:

```
1 leaf(a,b) {
2   try {
3     Kernel kernel = Cashmere.getKernel();
4     KernelLaunch kl = kernel.createLaunch ();
5     MCL.launch(kl, a, b);
6   catch ( exception ) {
7     leafCPU ( a,b )
8 }}
```

Insight in performance



Cashmere: Summary

- MCL makes optimizing kernels for many devices possible
  - seamless integration of many-core functionality
- High performance, scalability, and automatic load balancing even for widely different many-cores
- Efficiency >90% in 3/4 applications in heterogeneous execution

application	performance (GFLOPS)	configuration
raytracer	1883	10 gtx480, 2 c2050, 1 gtx680, 1 titan, 1 hd7970
matmul	3927	10 gtx480, 2 c2050, 1 gtx680, 1 titan, 1 hd7970
k-means	10644	10 gtx480, 2 c2050, 1 gtx680, 1 titan, 1 hd7970, 7 k20, 1 xeon_phi
n-body	13517	10 gtx480, 2 c2050, 1 gtx680, 1 titan, 1 hd7970, 7 k20, 2 xeon_phi

References

[C-1] Hijma et al. “Cashmere: Heterogeneous Many-Core Computing”, IPDPS, 2015

[C-2] Nieuwpoort et al. “Satin: A High-Level and Efficient Grid Programming Model,” ACM TOPLAS, 2010

[C-3] Hijma et al. “Stepwise-refinement for performance: a methodology for many-core programming,” CCPE, 2015

HYGRAPH

A graph processing solution

End of part IV

- Questions ?