Development of High Performance Computing Applications Across Heterogeneous Systems

Lecture ***2***

Frameworks to Aid Code Development and Performance Portability

André Pereira

LIP-Minho/University of Minho

Inverted CERN School of Computing, 23-24 February 2015

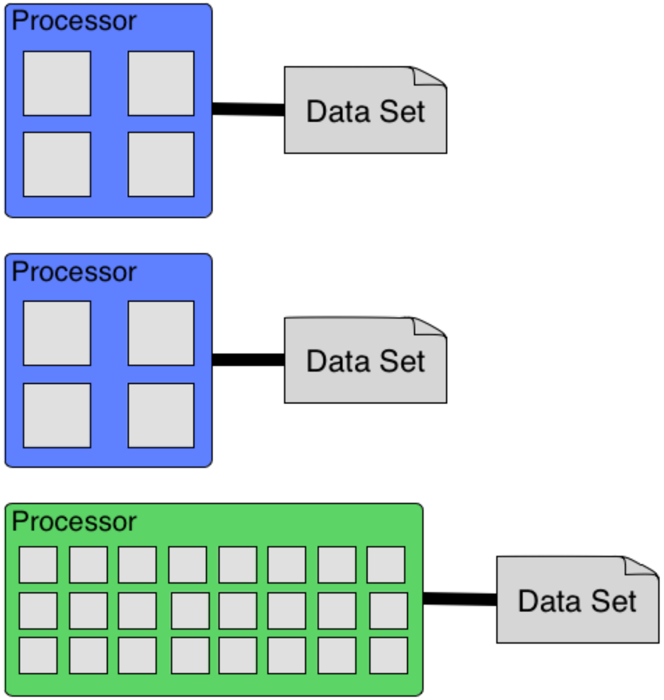
Agenda

- **Motivation**
- Frameworks for Heterogeneous Programming
- A Small Example with DICE
- Performance Analysis of Case Studies

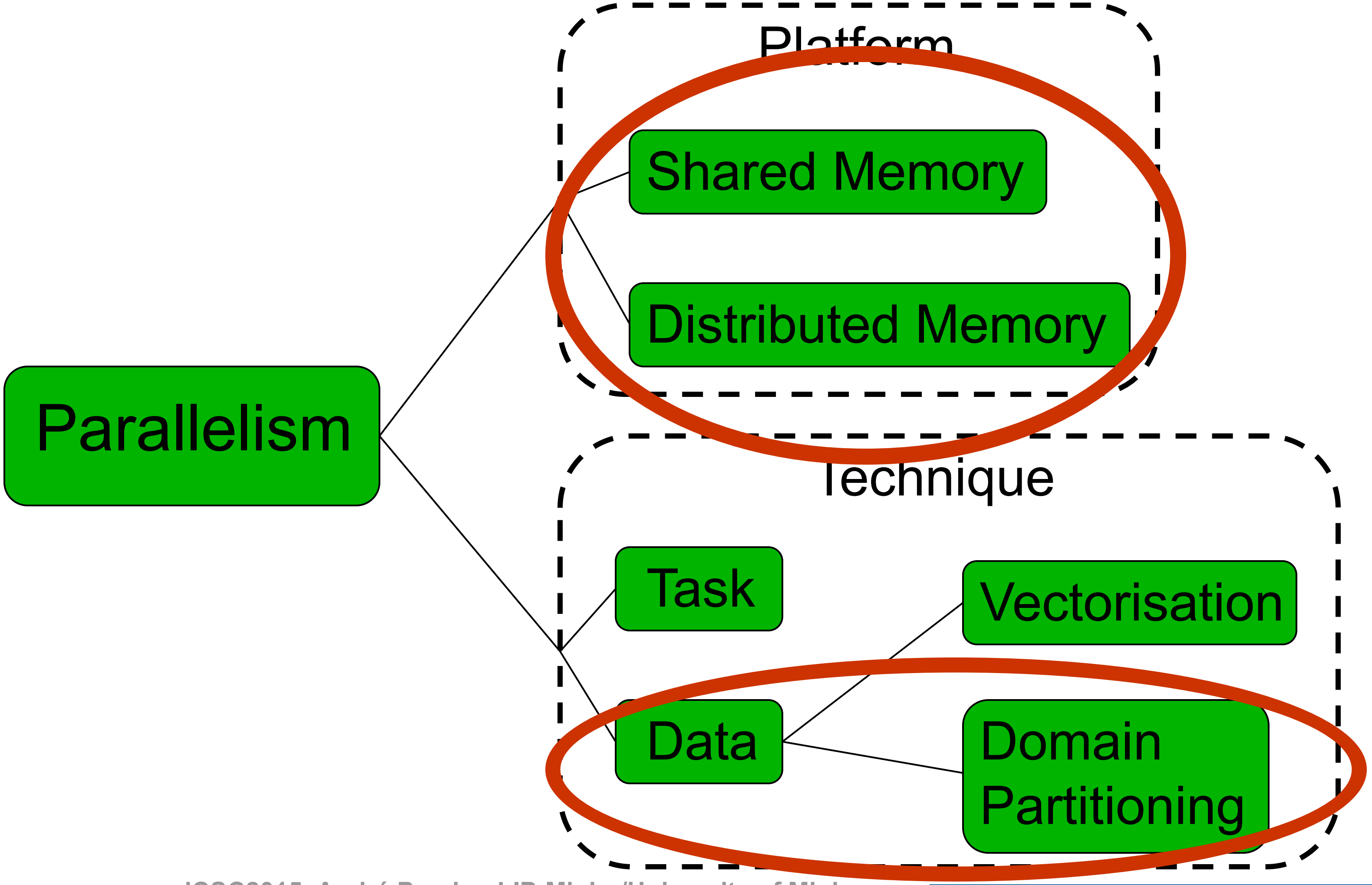
HetPlats Challenges

- **“I spent months optimising my code for HetPlats, I bet it will be super fast on this new system I just bought”**
    - No! You need to re-tune the code for each system…

- How is it possible to
    - achieve code scalability in each device?
    - simultaneously use both computing devices?
    - write the code once and guarantee its performance across different HetPlats?



Levels of Parallelism



Frameworks

- There are frameworks to help the development of code for heterogeneous platforms

- They provide several key features to the programmer
  - Abstraction of the distributed memory environment
  - Automatic workload balance among processing units
  - Coding the algorithm once to run on different processing units
  - Management of different task dependencies
  - Adaptation to the computing platform

- They are open source!
  - And provide several tutorials

Frameworks

- The downside is…
  - Steep learning curve for non-computer scientists
  - Production code has to be re-written to fit their programming model
  - Some frameworks require user configuration for each task/algorithm, which may have a huge impact on performance

- Different frameworks use different strategies to
  - Implement the algorithms
  - Minimise the costs of transferring the data among processing units
  - Handle **RAW**, **WAW**, and **WAR** task dependencies
  - Schedule the workload among processing units

Revisiting the Challenges

- **Different architectures**
  - Distinct designs of parallelism ✓
  - Distinct memory hierarchies
- **Different programming paradigms**
  - Distinct code for efficient algorithms among devices ✓
- **Workload management**
  - High latency communication between CPU and device ✓
  - Different throughputs among devices

Frameworks

StarPU

- *“Task programming library for hybrid architectures”*
- Implementation through the library API or compiler pragmas
- Uses a task-based parallelisation approach
  - Programmer codes *codelets* to run on the processing units
  - StarPU hides memory transfer costs by interleaving different tasks
  - Fixed workload grain size (defined by the user)
  - Also works on cluster environments with MPI

Frameworks

Legion

- *“Data-centric programming model for writing high performance applications”*
- A parallelisation approach focused on the data set
  - User specifies properties to the data structures, such as organisation, partitioning, and coherence
  - Legion handles the parallelism and data transfer, according to the specified properties
  - User maps the tasks to the processing units
  - Legion schedules the workload at runtime to handle irregular tasks

Frameworks

DICE

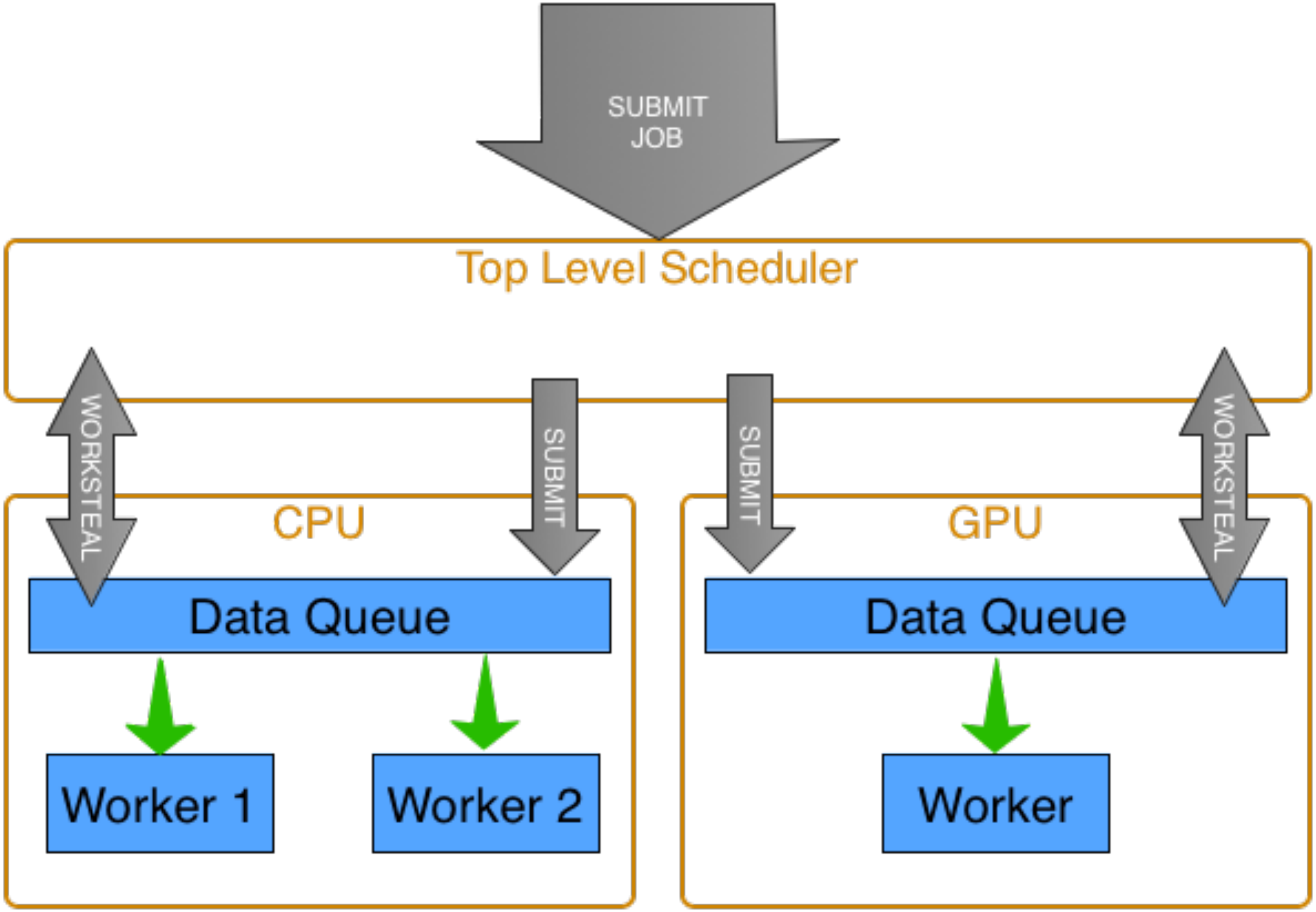
- Programming model and runtime system for irregular applications
  - **D**ynamic **I**rregular **C**omputing **E**nvironment

- Data parallelism approach with an unified memory space
  - Provides various data containers, with different properties
  - Allows to provide optimised code for each processing unit
  - The user has to code a dicing function – used to minimise the data transfers
  - Workload grain size adapts dynamically at runtime

- Requires some expertise to produce high performing code

Frameworks

DICE – Runtime System



A Small Example with DICE

- SAXPY – Single precision `alpha * x[i] + y[i]`
  - Linear complexity O(n)
  - No data dependencies

```
void saxpyCPU (float a, float *x,
               float *y, float *r, int N) {

  for (int i = 0; i < N; i++)
    r[i] = a * x[i] + y[i];
}
```

A Small Example with DICE

- Data Structures
  - Defined inside the `work` class
  - Global memory construct to be shared among processing units
  - Scalar variables do not need a special identifier
  - This belongs to the high level API

```
smartPtr<float> R;
smartPtr<float> X;
smartPtr<float> Y;

float alpha;
```

A Small Example with DICE

- Data properties
  - Assigned to the data structure when they are initialised
  - `smarPtr` are classes, implementing *getters* and *setters*
  - Properties: DEVICE, SHARED, READ_ONLY

```
smartPtr<float> R = smartPtr<float>(N, Property) ;
```

A Small Example with DICE

- Define the task properties
  - Give an unique identifier to each task

```
enum WORK_TYPE {
  /* !< Empty job description. DO NOT CHANGE */
  WORK_NONE = W_NONE,
  /* !< SAXPY job definition */
  WORK_SAXPY,
  /* TO DO: Add you job descriptions here */
  /* !< Total number of job definitions. DO NOT CHANGE */
  WORK_TOTAL,
  /* !< Reserved bit mask job. DO NOT CHANGE */
  WORK_RESERVED = W_RESERVED
};
```

A Small Example with DICE

- Fit the code to the `Worker` class
  - Declare an empty constructor with the job description
  - `W_REGULAR` indicates that the workload is irregular (as opposed to `W_IRREGULAR`)
  - `W_SAXPY` maps the defined identifier to the method

```
saxpy() : work(WORK_SAXPY | W_REGULAR) {

}
```

A Small Example with DICE

- Fit the code to the `Worker` class
  - `__HYBRID__` indicates that the code is to be simultaneously scheduled among all processing units
  - `__DEVICE__`, accompanied by a `template<DEVICE_TYPE>` specifies the code to be compiled for a specific device

```
__HYBRID__ saxpy() : work(WORK_SAXPY | W_REGULAR) {

}
```

A Small Example with DICE

- Fit the code to the `Worker` class
  - Create another construct that receives the inputs as `smartPtr` data structures
  - Length, lower, and upper?

```
__HYBRID__ saxpy(
          smartPtr<float> _R, smartPtr<float> _X, smartPtr<float> _Y,
          float _alpha, unsigned _LENGTH, unsigned lo, unsigned hi)
          : work(WORK_SAXPY | W_REGULAR),
          R(_R), X(_X), Y(_Y), alpha(_alpha),
          LENGTH(_LENGTH), lower(lo), upper(hi)
          {

 }
```

A Small Example with DICE

- The dicing function (the hard bit…)

```
template<DEVICE_TYPE>
__DEVICE__ List<work*>* dice(unsigned &number) {
   unsigned range = (upper-lower);
   unsigned number_of_units = range / number;

   if(number_of_units == 0) {
             number_of_units = 1;
             number = range;
   }
   unsigned start = lower;
   List<work*>* L = new List<work*>(number);

   for (unsigned k = 0; k < number; k++) {
             saxpy* s = new saxpy(R,X,Y,alpha,LENGTH,start,start+number_of_units);
             (*L)[k] = s;
             start += number_of_units;
   }
   return L;
}
```

A Small Example with DICE

- Finally, the SAXPY code!
  - tid will define the position to process (similar to CUDA)
  - The code takes the upper and lower bound of the vector into account

```
template<DEVICE_TYPE>
__DEVICE__ void execute() {
        if(TID > (upper-lower)) return;
        unsigned long tid = TID + lower;

        for(; tid < upper; tid+=TID_SIZE)
                r.set(tid, x.get(tid)*alpha+y.get(tid));
}
```

A Small Example with DICE

- Initialise the runtime system, prepare the input data, and execute the code

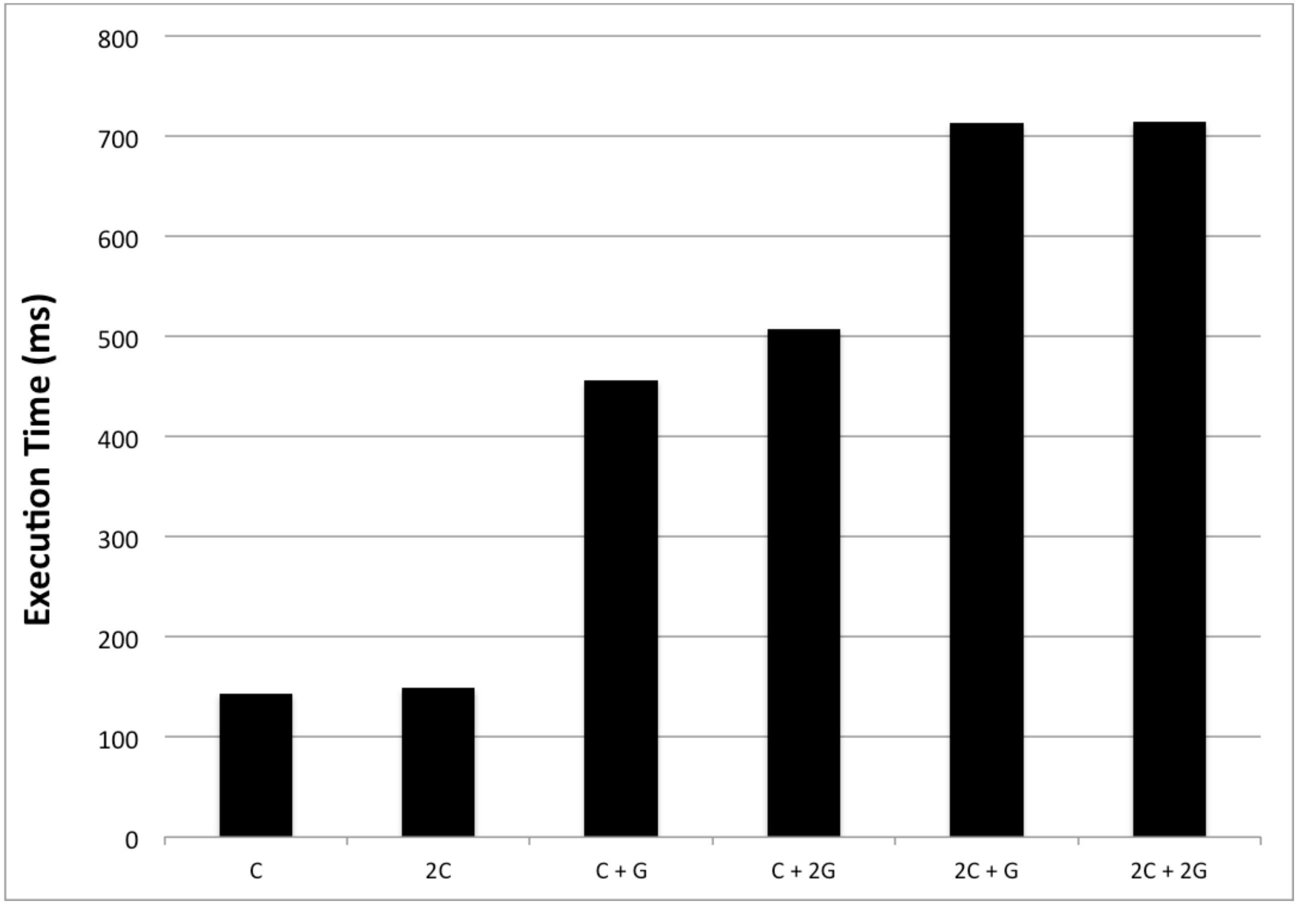
```
// Initialize runtime system
RuntimeScheduler* rs = new RuntimeScheduler();
// Create global memory space for shared vectors
smartPtr<float> R = smartPtr<float>(sizeof(float)*N, SHARED);
smartPtr<float> X = smartPtr<float>(sizeof(float)*N, SHARED);
smartPtr<float> Y = smartPtr<float>(sizeof(float)*N, SHARED);
// Initialise the data…
…
// Create work description
saxpy* s = new saxpy(R,X,Y,alpha,N,0,N);
// Submit work for execution and synchronize after execution
rs->submit(s);
rs->synchronize();
```

Testbed Environment

- Morpheus
  - 2x Intel Xeon 6-core CPUs @ 2.6 GHz
  - 2x NVidia Tesla C2070 4 GB DRAM
- MacBook Pro
  - Intel i7 Ivy Bridge 4-core CPU @2.6 GHz
  - NVidia 650M GPU
- Software
  - GNU compiler version 4.8.3
  - CUDA Toolkit 6.5

Performance Analysis

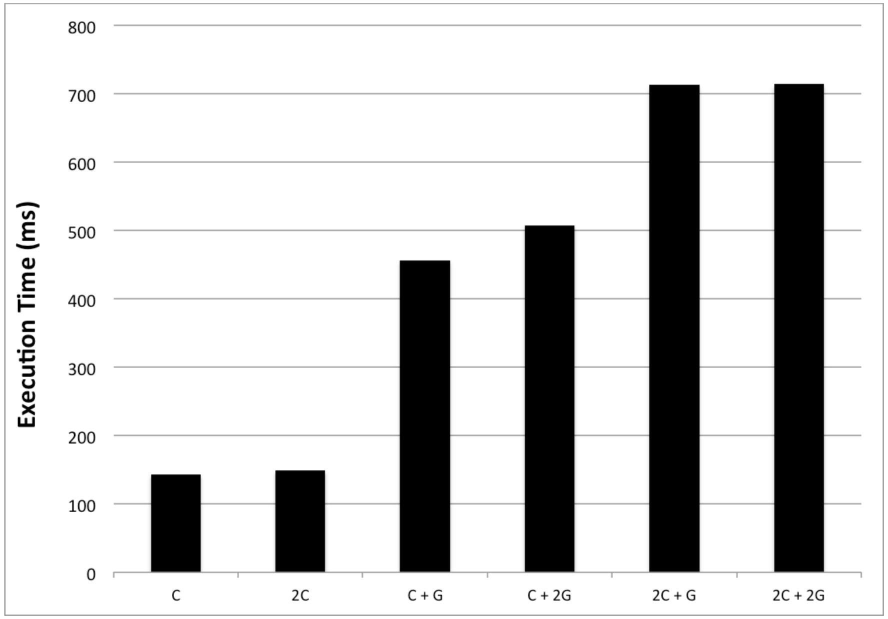
SAXPY with DICE



C – CPU
G – GPU

Scalability of SAXPY for various system configurations for a vector of 300M elements

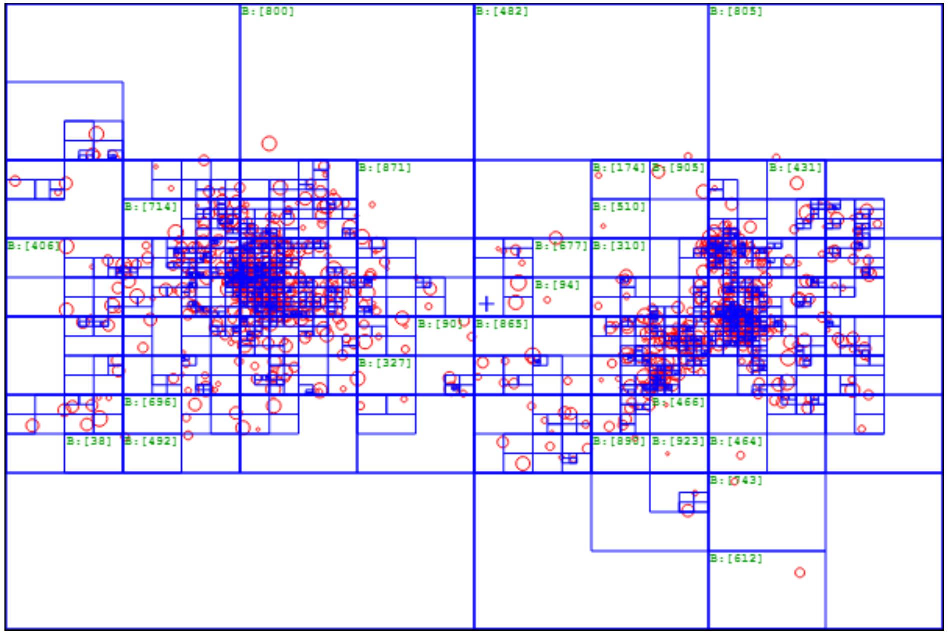
SAXPY with DICE



- This problem is not scalable…
- The overhead of communications and scheduling restricts performance
  - The problem is extremely regular and too simple (computationally)
  - Analyse your code first!

Performance Analysis

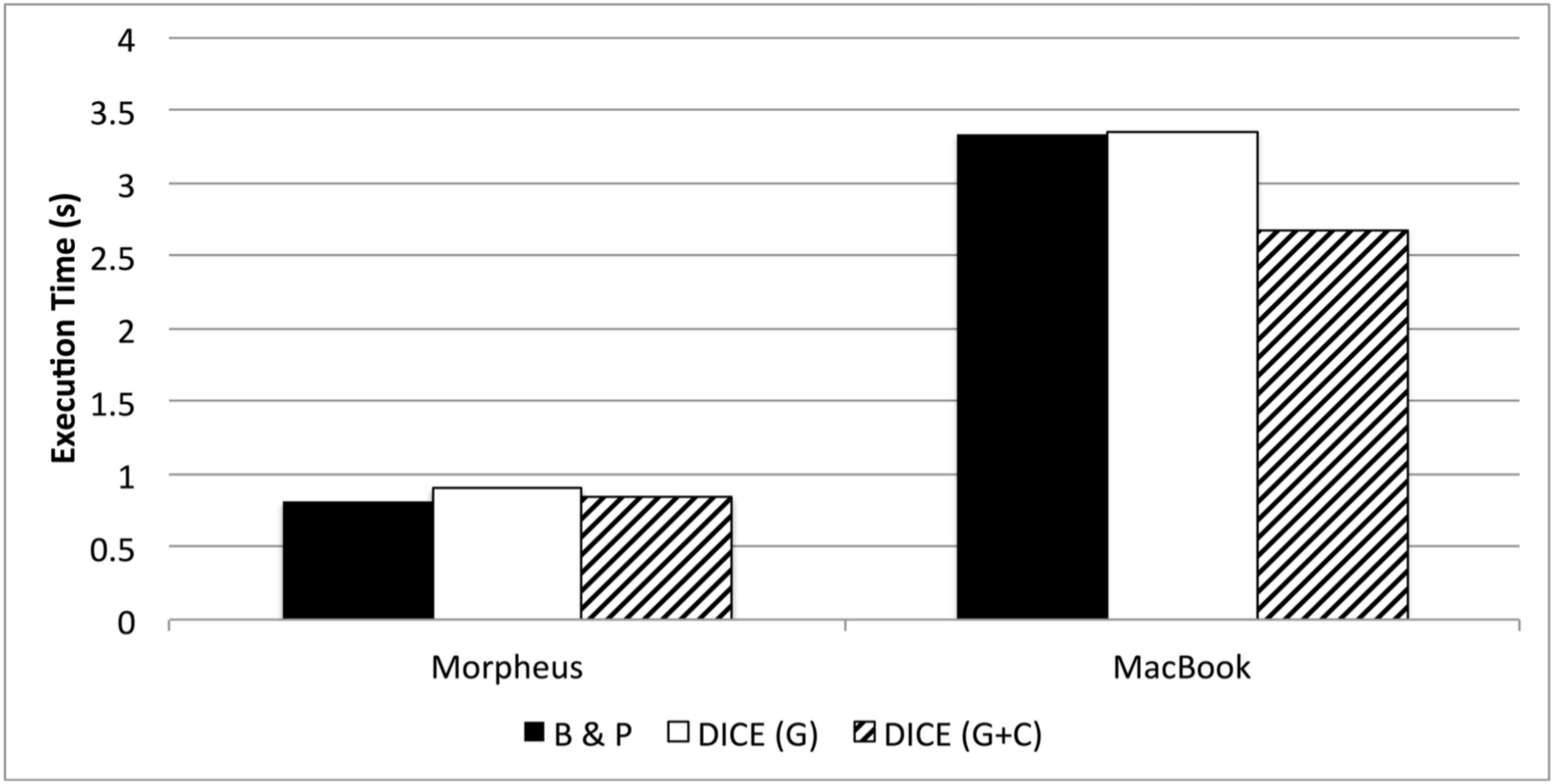
Barnes-Hut with DICE



- Barnes-Hut algorithm simulates n-body system interactions
  - Divides the space, creates an hierarchy to speedup particle interaction calculations, with a complexity of O(n log(n))
  - Particle clusters should be on the same processor
  - Workload is dynamic
  - Fastest GPU implementation by Burtscher and Pingali (B&P)

Performance Analysis

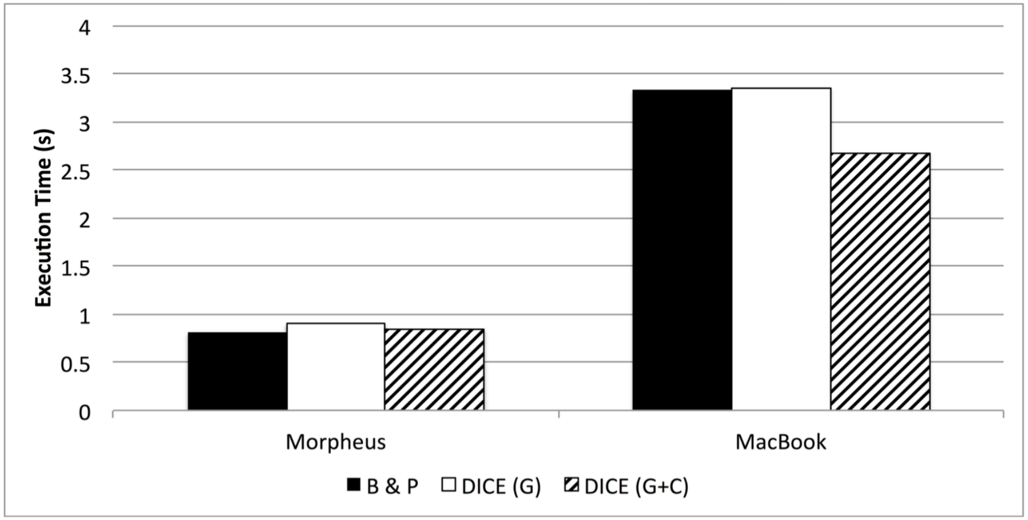
Barnes-Hut with DICE



C – CPU
G – GPU

Execution time of Barnes-Hut for 1M particles

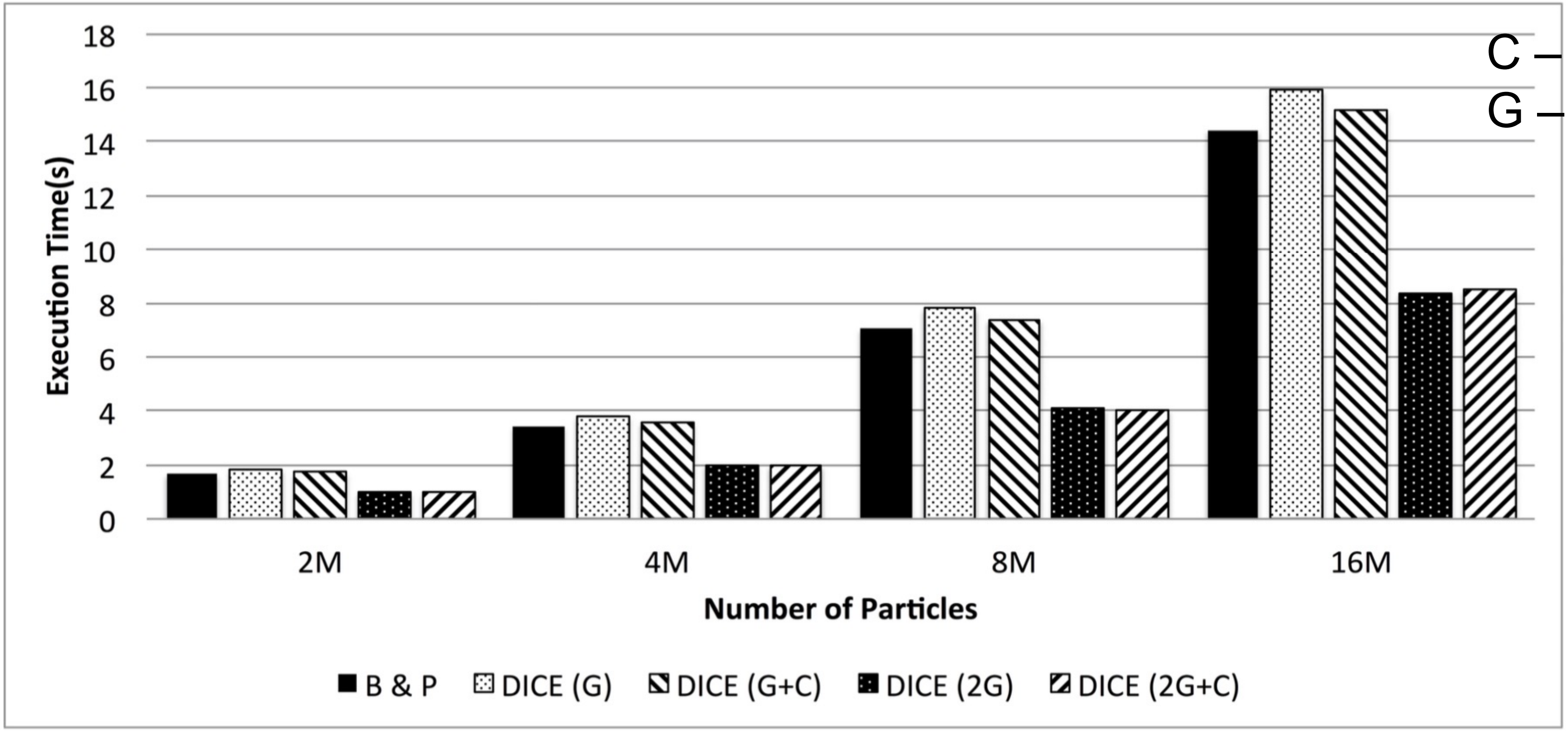
Barnes-Hut with DICE



- Not a big improvement over the best GPU implementation
- The problem size is not big enough
  - The communication and workload management overhead restricts the performance

Performance Analysis

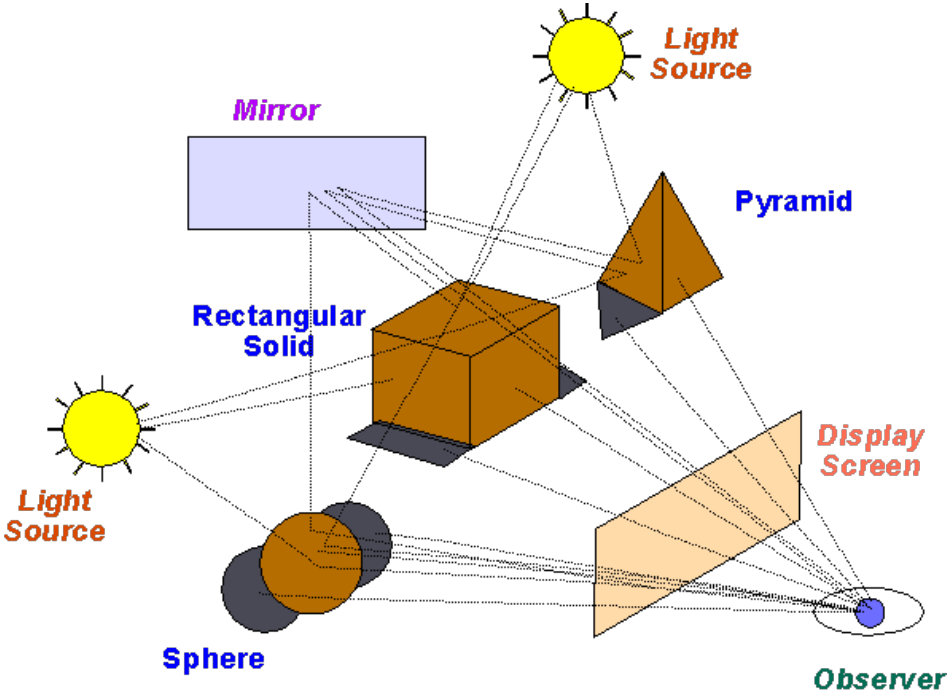
Barnes-Hut with DICE



C – CPU
G – GPU

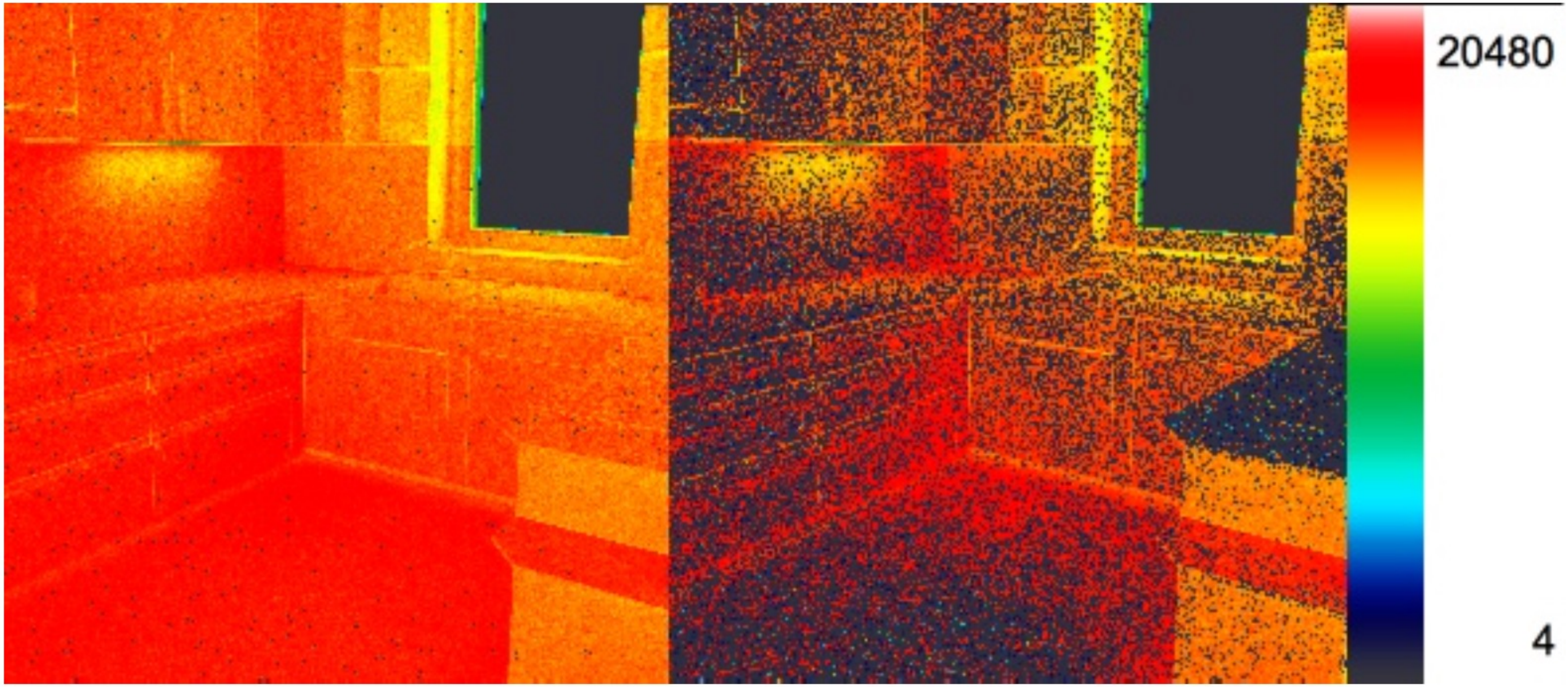
Scalability of Barnes-Hut for various problem sizes

Path Tracing with DICE



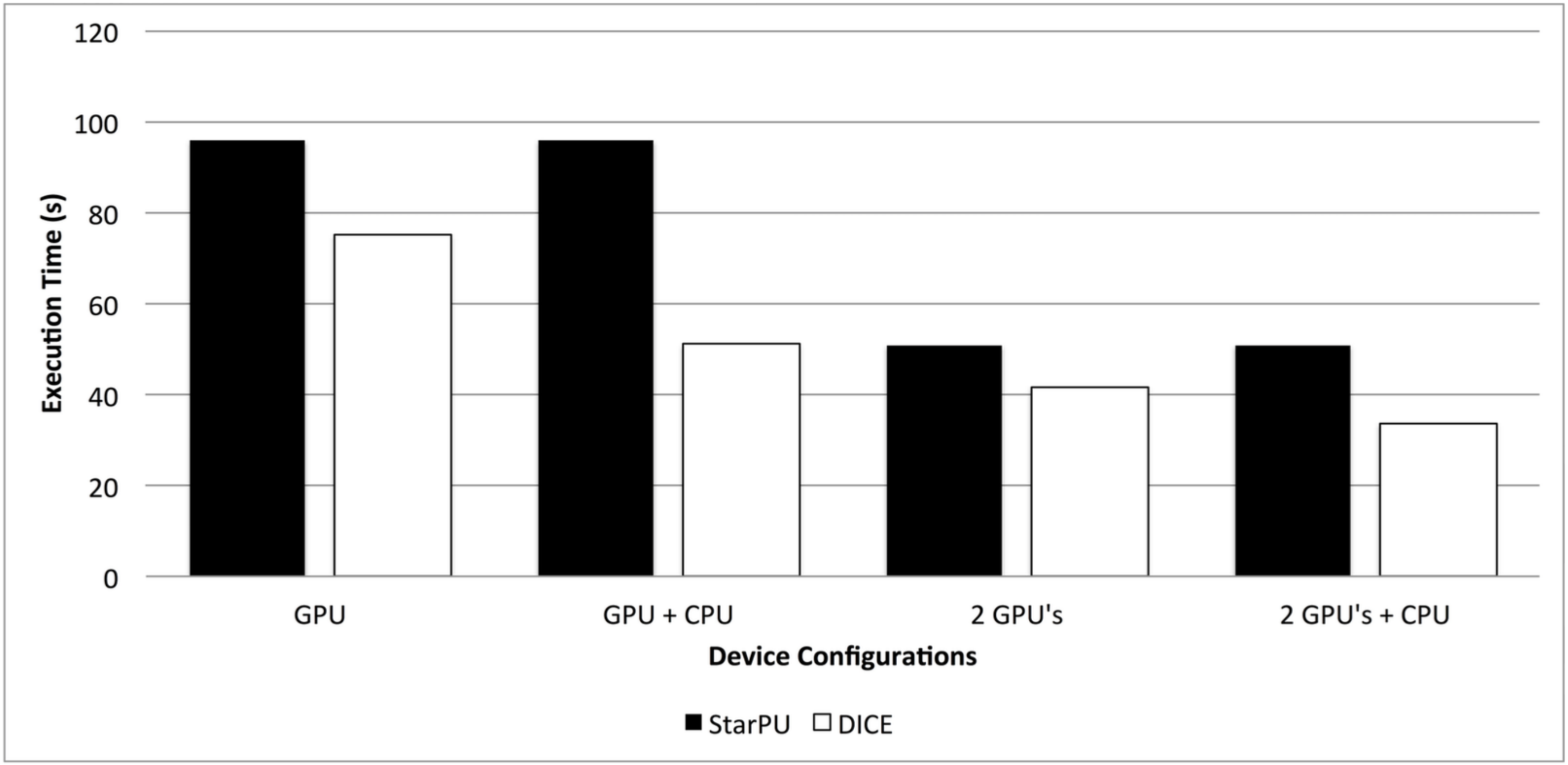
- Monte Carlo simulation to render physically accurate scenes
  - Recursive algorithm
  - Dynamic workload

Path Tracing with DICE



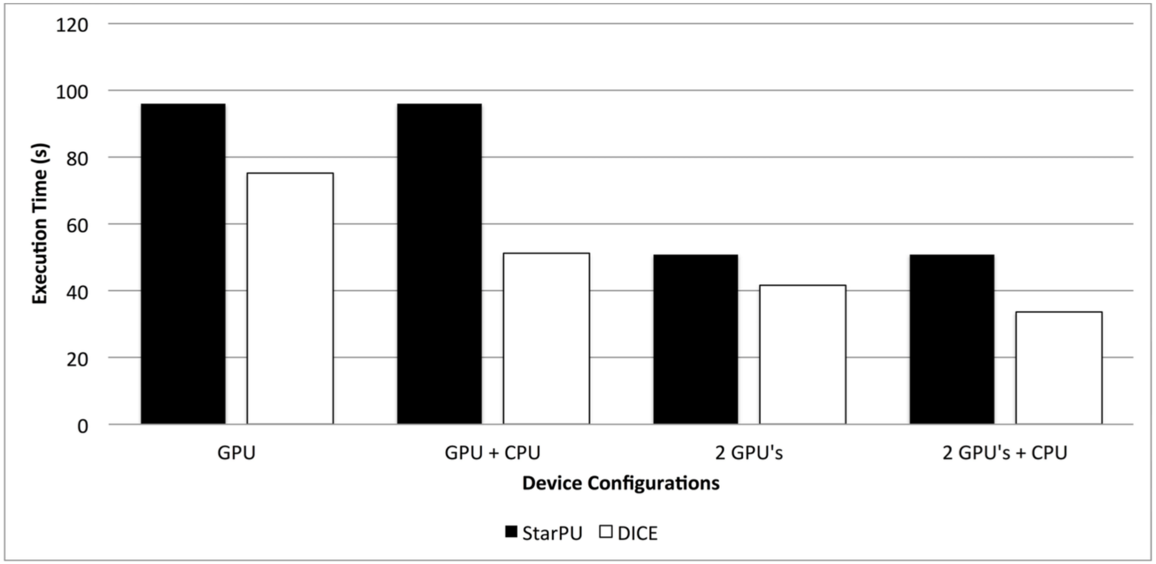
Ray count for progressive pathtracer with variance threshold of 1% (left) and 25%(right)

Path Tracing with DICE



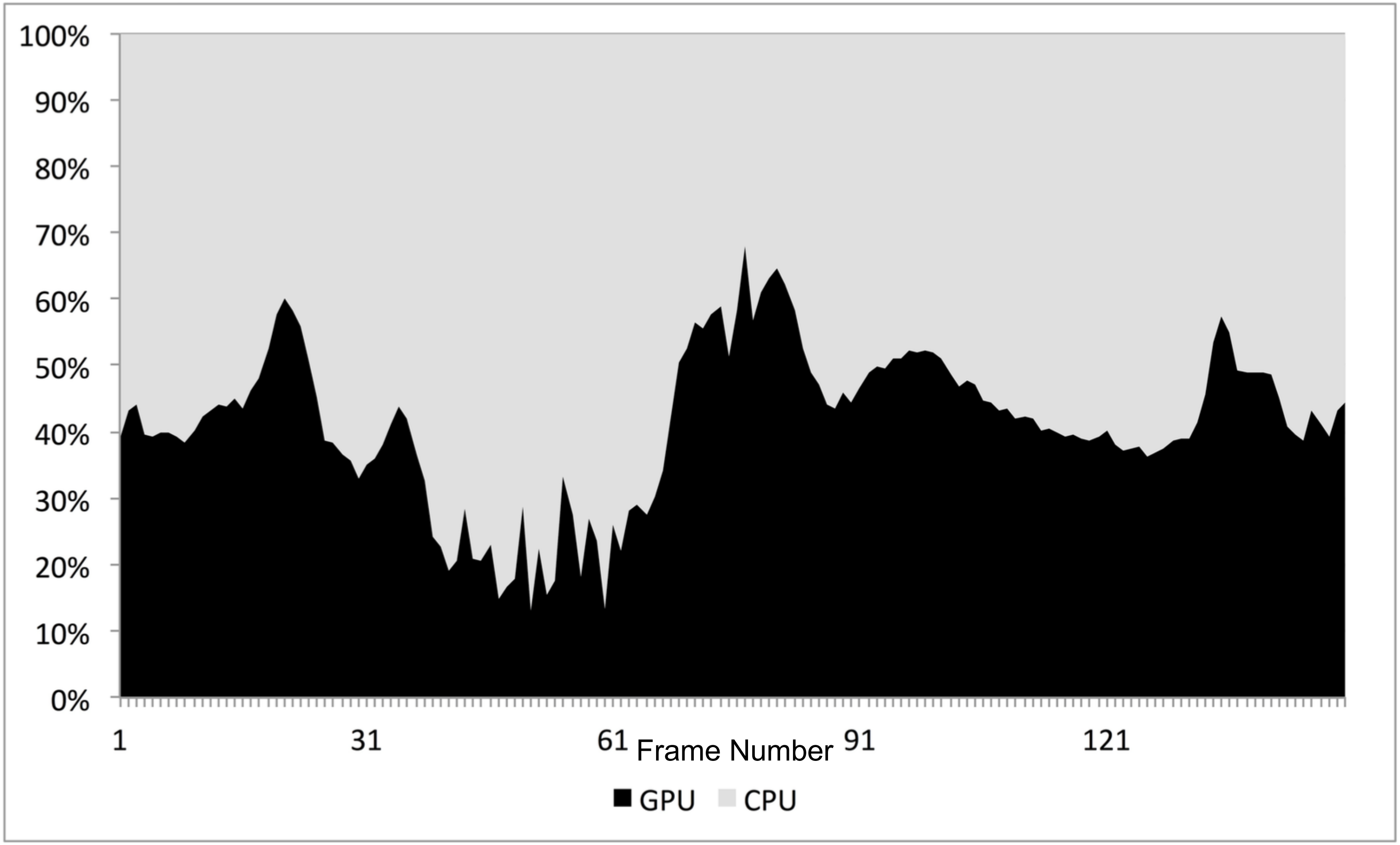
Comparison of the StarPU and DICE implementations of the Path Tracer running on Morpheus

Path Tracing with DICE



- DICE provides a 20% performance improvement
- DICE is 2x faster than StarPU in the best case
- Handles CPU+GPU parallelisation better than StarPU

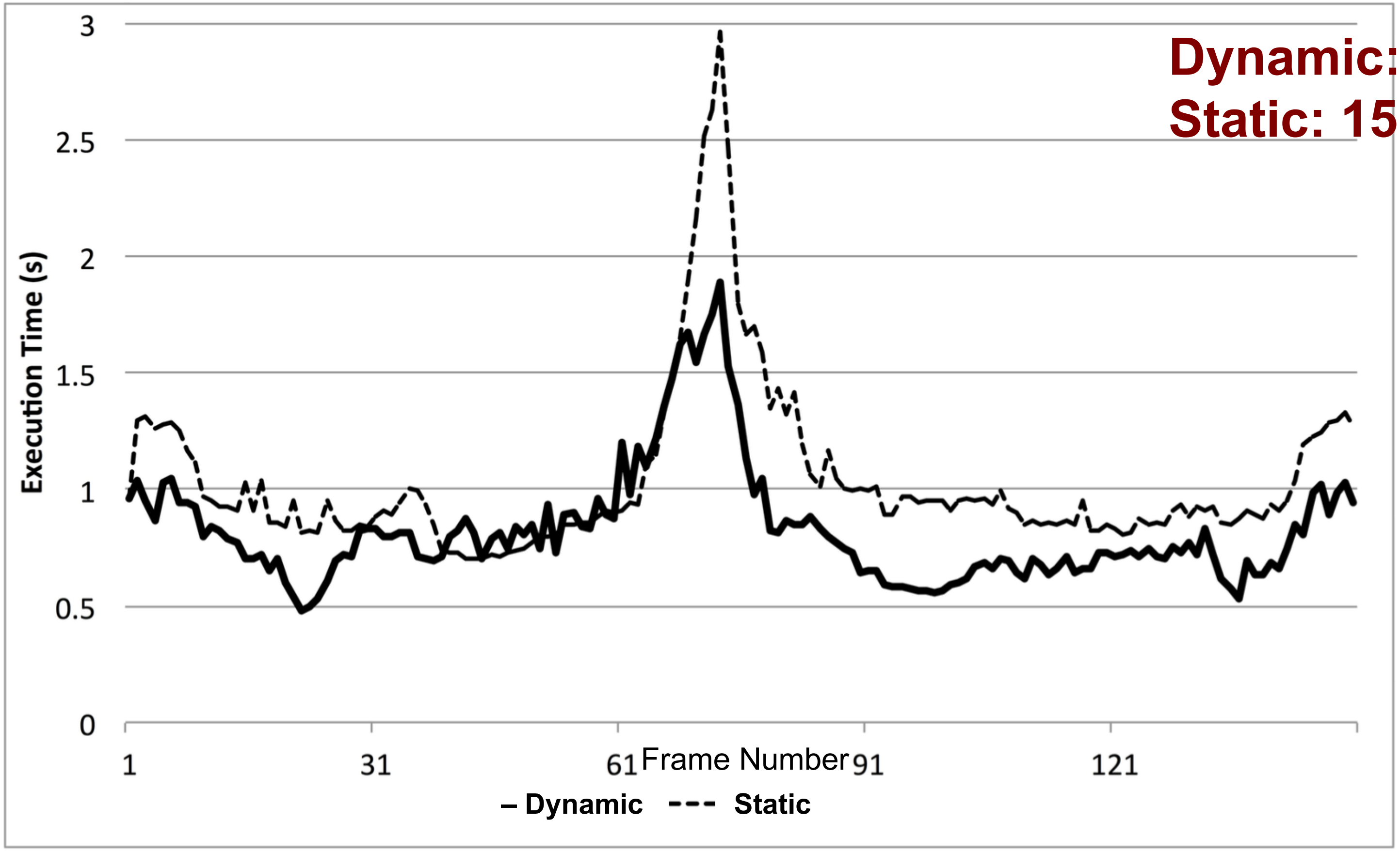
Path Tracing with DICE



Workload distribution between the CPU and GPU for the Adaptive Path Tracer (irregular workload)

Performance Analysis

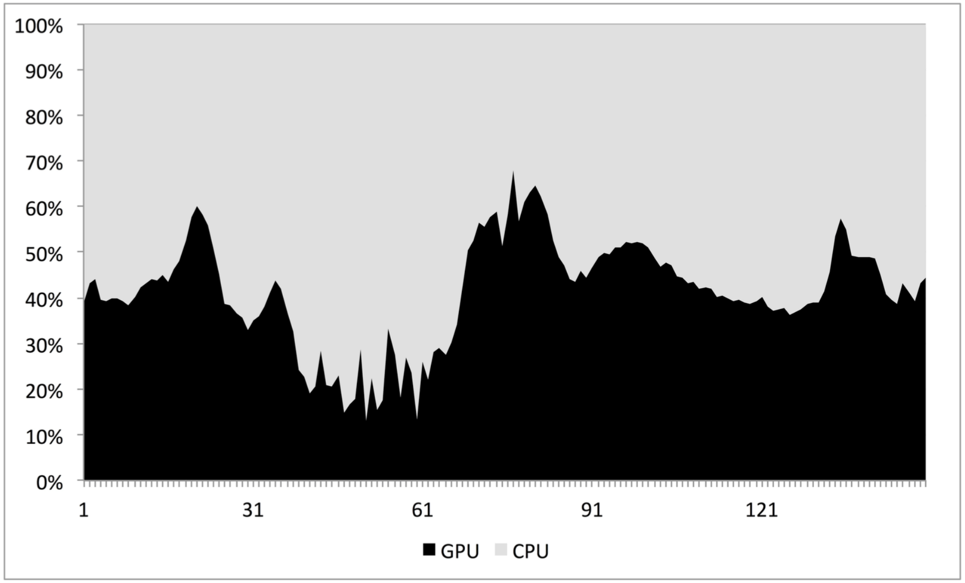
Path Tracing with DICE

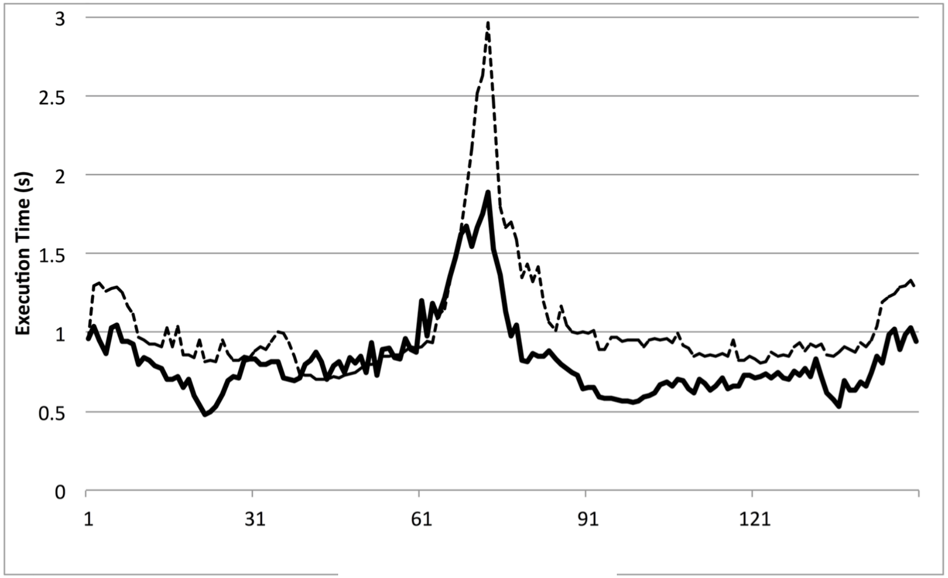


Dynamic: 122 s
Static: 157 s

Frame by frame execution time for dynamic vs static (40% CPU, 60% GPU) schedulers

Path Tracing with DICE





- DICE dynamically adapts to the change of task execution time
  - It is always tuning the amount of work that the CPU/GPU processes
- Dynamic scheduling is 30% faster than a tuned static scheduling technique

Conclusions

- Coding for HetPlats is complex and time consuming
  - Simultaneously deal with different levels of parallelism

- There are frameworks to help code development
  - Some effort is required to get familiar with
  - Automatically balance the workload among CPUs and GPUs
  - Adapt to the computing platform and irregular tasks at runtime

Acknowledgment

- A special thanks to the DICE developers for letting me use the framework – which is still in beta
  - João Barbosa – UMinho & University of Texas
  - Roberto Ribeiro – UMinho
  - Donald Fussell – University of Texas
  - Calvin Lin – University of Texas
  - Luís Paulo Santos – UMinho
  - Alberto Proença – UMinho

Development of High Performance Computing Applications Across Heterogeneous Systems

*Lecture **2***

Frameworks to Aid Code Development and Performance Portability

André Pereira

LIP-Minho/University of Minho

Inverted CERN School of Computing, 23-24 February 2015