Collaborative Execution Environment for Heterogeneous Parallel Systems – CHPS*

Aleksandar Ilić and Leonel Sousa

In 12th Workshop on Advances in Parallel and Distributed Computational Models (APDCM/IPDPS 2010), January 2010

Instituto de Engenharia de Sistemas e Computadores Investigação e Desenvolvimento em Lisboa

INSTITUTO SUPERIOR TÉCNICO

Outline

- Desktop heterogeneous systems
- Collaborative execution and programming challenges
- Unified execution model
- Case studies:
  - Dense matrix multiplication
  - Complex 3D fast Fourier transformation
- Experimental results
- Conclusions and future work

Motivation

- Commodity computers = **Heterogeneous systems**
  - Multi-core general-purpose processors (CPUs)
  - Many-core graphic processing units (GPUs)
  - Special accelerators, co-processors, FPGAs, DSPs

⇒ Huge **collaborative** computing power
  - Not yet explored in detail
  - In most research – one device is used at the time; domain-specific computations

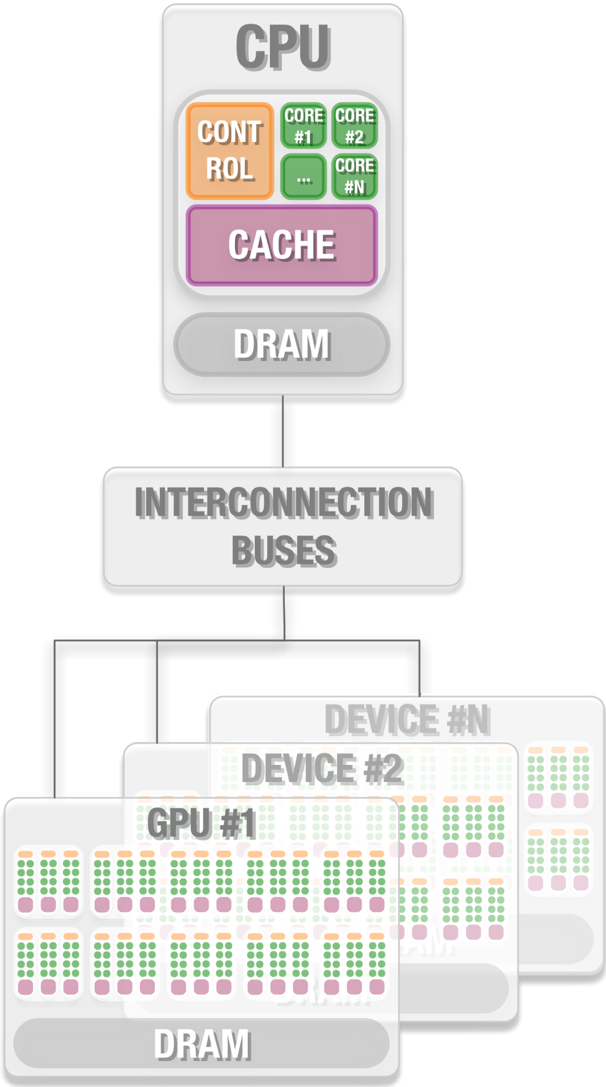
- Heterogeneity makes problems much more complex
  - many programming **challenges**

Heterogeneous Systems

Master-slave execution paradigm

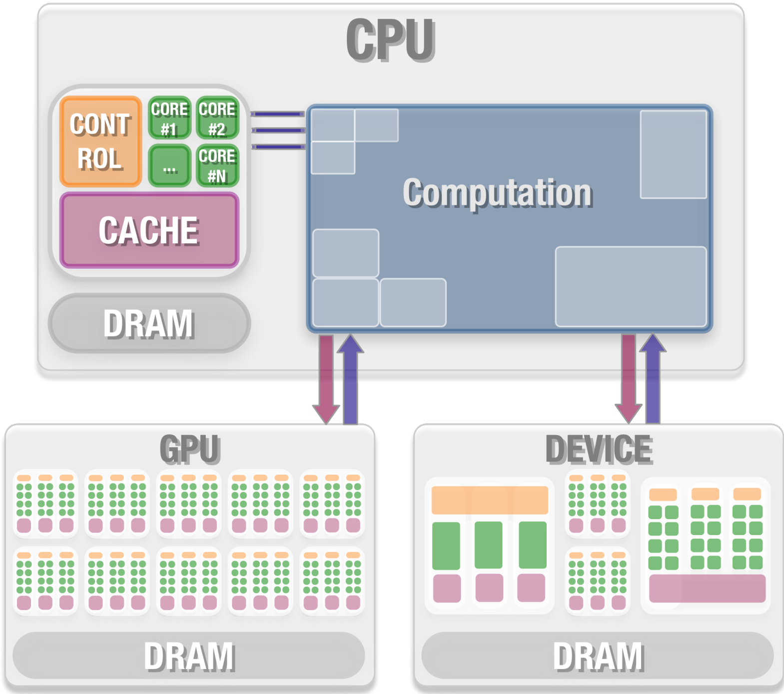
- Distributed-memory programming techniques

- **CPU** (Master)
  - Global execution controller
  - Access the whole global memory
- **Interconnection Busses**
  - Reduced communication bandwidth comparing to distributed-memory systems
- **Underlying Devices** (Slaves)
  - Different architectures and programming models
  - Computation performed using local memories



Programming Challenges

- **Computation Partitioning**
  - To fulfill device capabilities/limitations and achieve optimal load-balancing
- **Data Migration**
  - Significant and usually asymmetric
  - Potential execution bottleneck
- **Synchronization**
  - Devices can not communicate between each other => CPU in charge



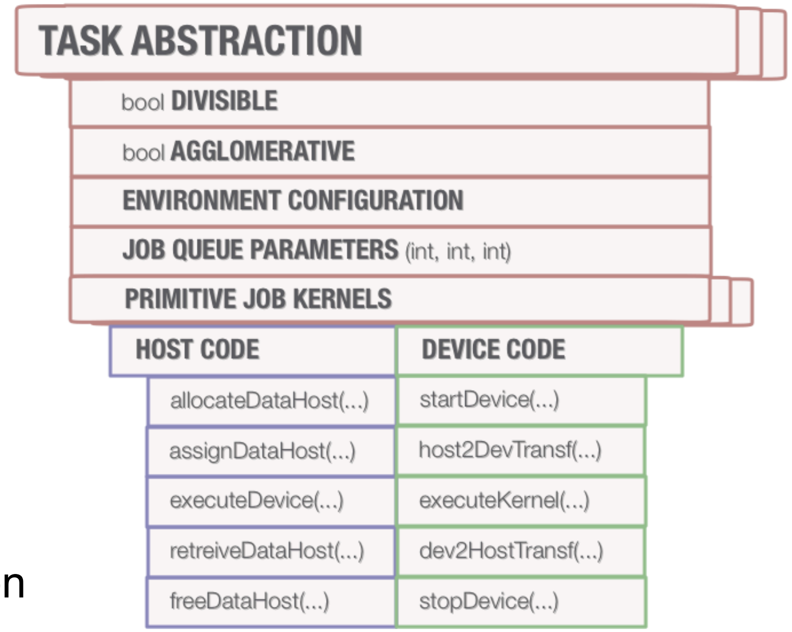
- **Different programming models**
  - *Per* device type and vendor-specific
  - High performance libraries and software
- **Application Optimization**
  - Very large set of parameters and solutions affects performance

Task Abstraction

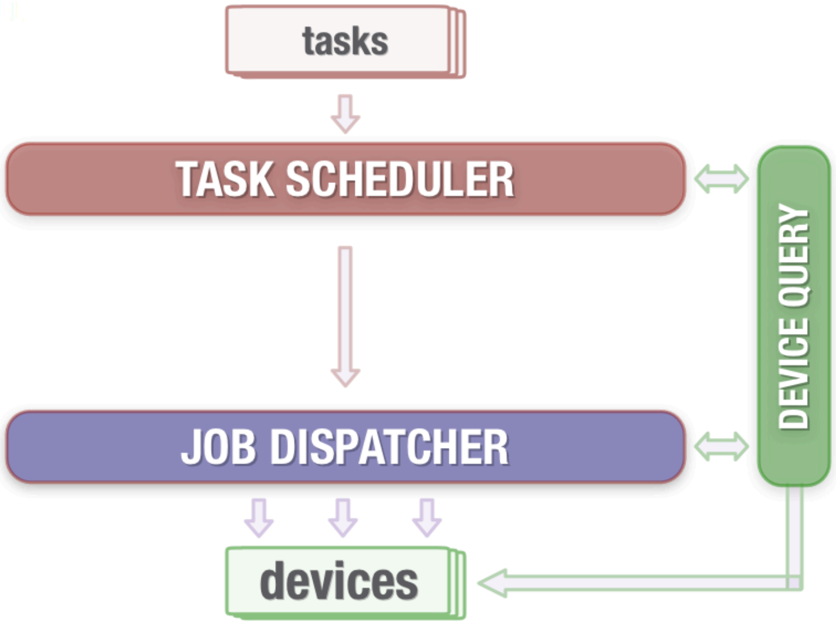
technology
from seed

Task - coarser-grained, basic programming unit

- **Task Extensions**:
  - *Environment Configuration Parameters*
    - Device type, number of devices…
  - ***Divisible*** – into finer-grained *Primitive Jobs*
  - ***Agglomerative*** – grouping of *Primitive Jobs*



- **Primitive Jobs**
  - **Minimal program** portions for parallel execution
  - **Balanced** granularity
  - Partitioned into *Host and Device Code*
    - Direct integration of different programming models and vendor libraries (peak performance)
    - Use of specific optimization techniques on per-device basis (data migration, execution etc.)



Divisible	Agglomerative
NO	--

Task Scheduler

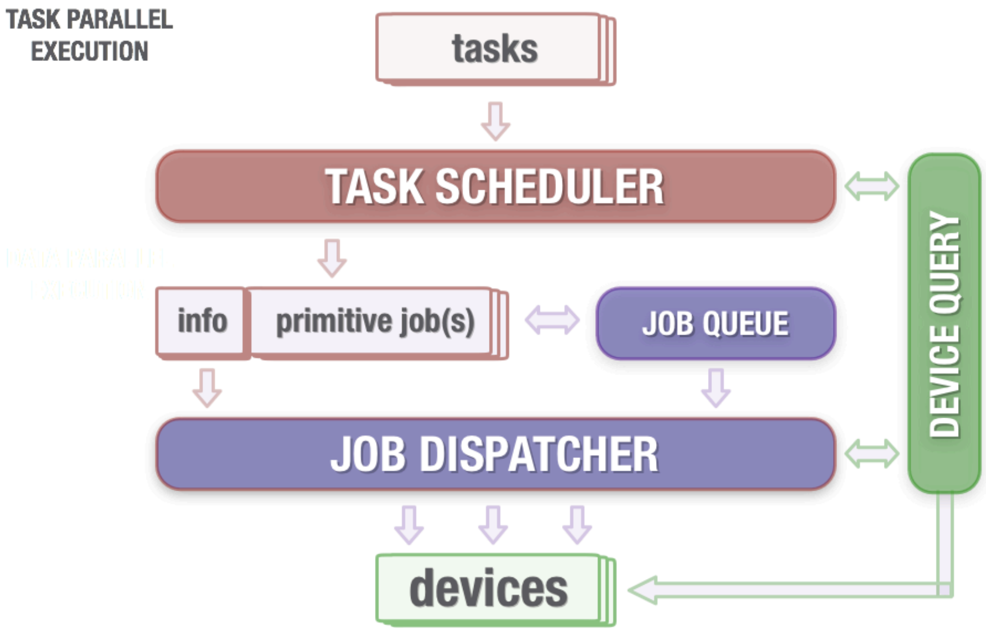
- Selects the next task for execution
  - according to the configuration parameters, device availability and dependencies
- Different scheduling schemes – list, DAG…

Job Dispatcher

- Assigns a requested device to the task
- Initiates and controls the on-device execution
- Synchronization between host and device

Device Query

- Identifies and examines all underlying devices
- Holds per-device information
  - resource type, status, memory management and performance history



Divisible	Agglomerative
NO	--
YES	NO

Task Scheduler

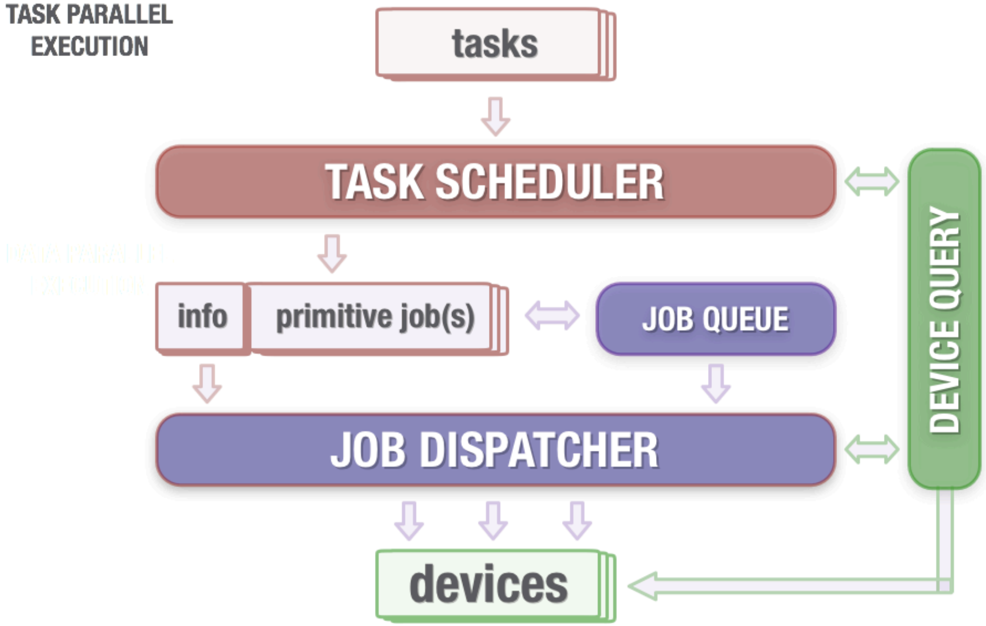
Job Queue

- Arranges the Primitive Jobs into structures
  - according to the parameters from the task properties
- Currently supports **grid** organization (1D–3D)

Job Dispatcher

- Search over a set of Primitive Jobs
- Mapping to the requested devices

Device Query



Divisible	Agglomerative
NO	--
YES	NO
YES	YES

Task Scheduler

Job Queue

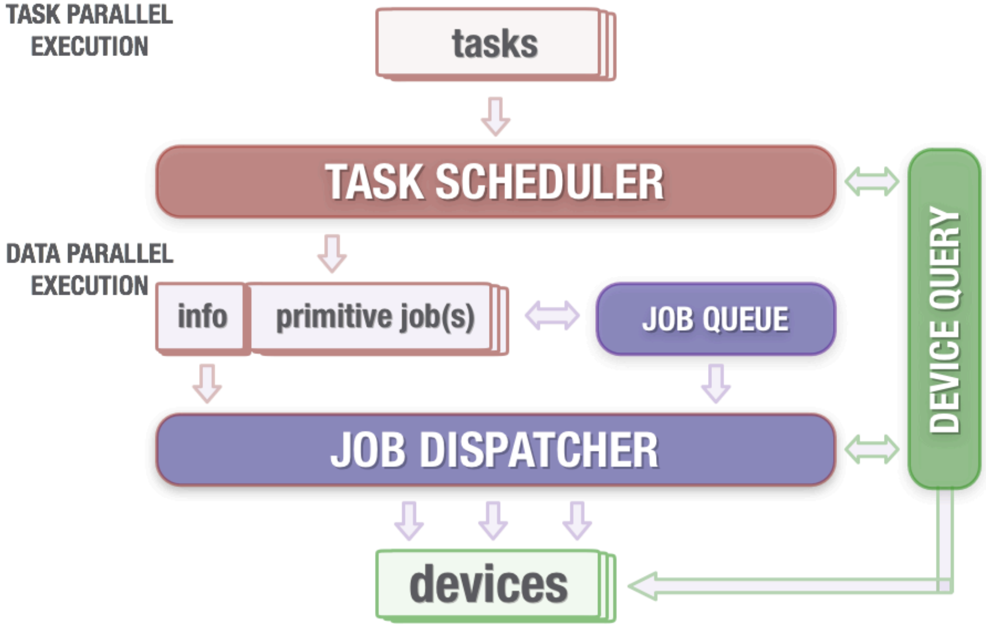
- Arranges the Primitive Jobs into structures
  - according to the parameters from the task properties
- Currently supports **grid** organization (1D–3D)

Job Dispatcher

- Search over a set of Primitive Jobs
- Mapping to the requested devices
- *Agglomeration* – select and group the Primitive Jobs into the Job batches

Device Query

Unified Execution Model



Divisible	Agglomerative
NO	--
YES	NO
YES	YES

Task Level Parallelism

– Scheduler free to send independent tasks to the Job Dispatcher

Data Level Parallelism

– Different portions of a single task are executed on several devices simultaneously

Nested Parallelism

– Multi-core device is viewed as a single device by the Job Dispatcher

– If provided by application

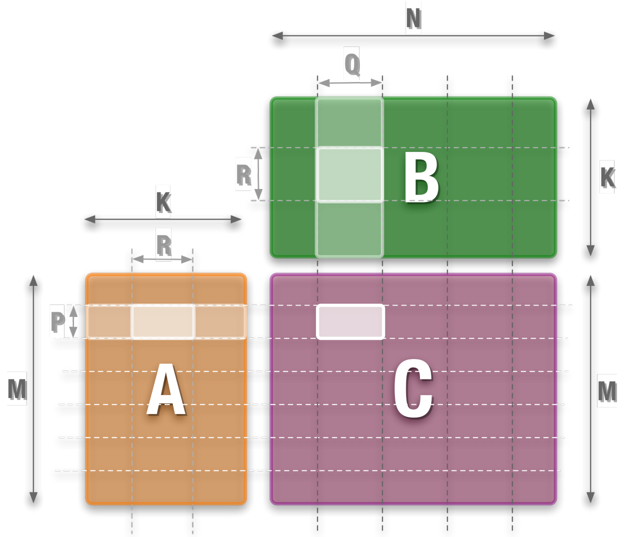
Case study I:
Dense Matrix Multiplication

technology
from seed

- General dense matrix multiplication $C_{M\times N} = A_{M\times K} \times B_{K\times N}$ is based on a **block decomposition**, where *A, B, C* matrices are partitioned into *PxR, RxQ, PxQ* sub-blocks, respectively

Divisible	**YES**
Agglomerative	**YES/NO**

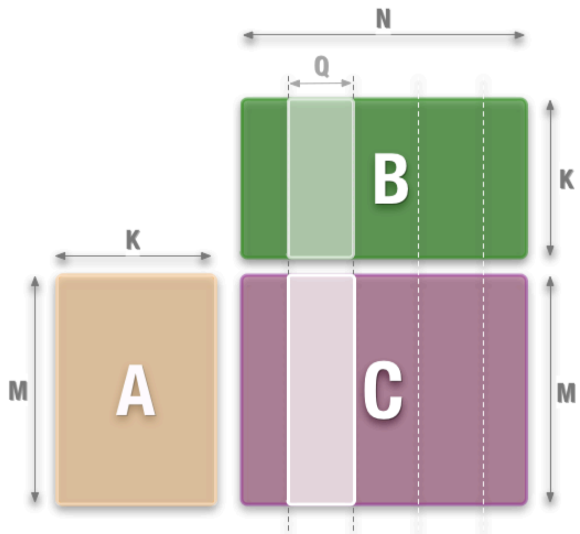
Problem size	**M**	**N**	**K**
Primitive Job size	**P**	**Q**	**R**
Job Queue size	$\frac{M}{P}\times\frac{N}{Q}\times\frac{K}{R}$		



Case study I: Dense Matrix Multiplication

- General dense matrix multiplication $C_{M \times N} = A_{M \times K} \times B_{K \times N}$ is based on a **block decomposition**, where *A, B, C* matrices are partitioned into *PxR, RxQ, PxQ* sub-blocks, respectively

Divisible	**YES**		
Agglomerative	**YES/NO**		
Problem size	**M**	**N**	**K**
Primitive Job size	**M**	**Q**	**K**
Job Queue size	$\frac{N}{Q}$		



- Special case implementation for **communication reduction**
  - each computational device is supplied with the A matrix,
  - agglomeration and distribution of the Primitive Jobs
- Implementation is bound to memory capacities of devices
  - device with the smallest amount of global memory sets the algorithm's upper bound

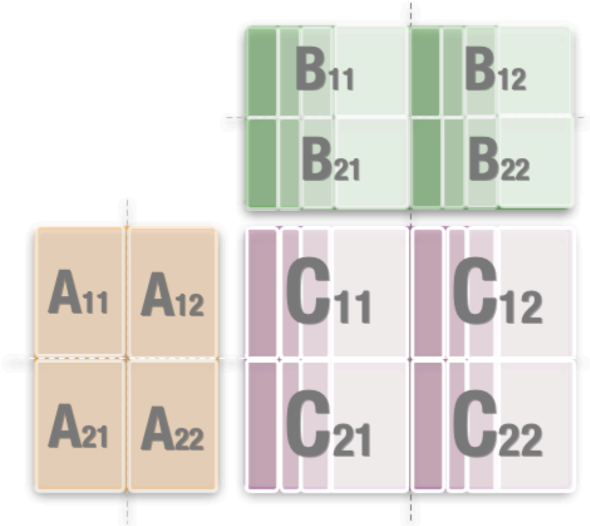
Case study I: Dense Matrix Multiplication

technology
from seed

inesc id lisboa

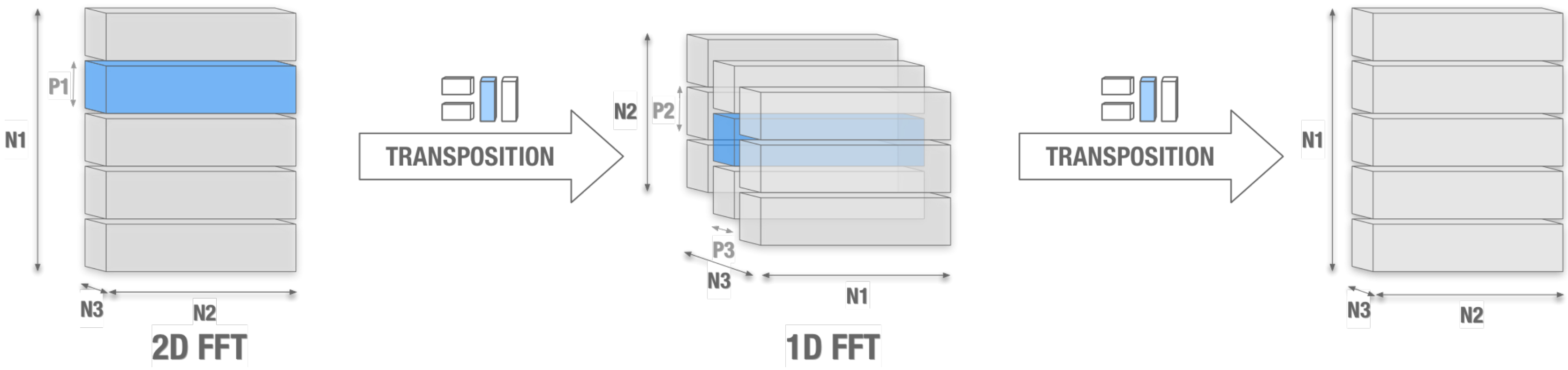
- General dense matrix multiplication $C_{M \times N} = A_{M \times K} \times B_{K \times N}$ is based on a **block decomposition**, where *A, B, C* matrices are partitioned into *PxR, RxQ, PxQ* sub-blocks, respectively

Divisible	**YES**		
Agglomerative	**YES/NO**		
Problem size	**M**	**N**	**K**
Primitive Job size	**M**	**Q**	**K**
Job Queue size	$\frac{N}{Q}$		



- **Horowitz scheme** to lessen memory restrictions of underlying devices
  - Set of block matrix multiplications to be performed
- List of *DGEMM* tasks as an input to the Scheduler

Case study II: 3D Fast Fourier Transform



$$H = FFT_{1D}(FFT_{2D}(h))$$

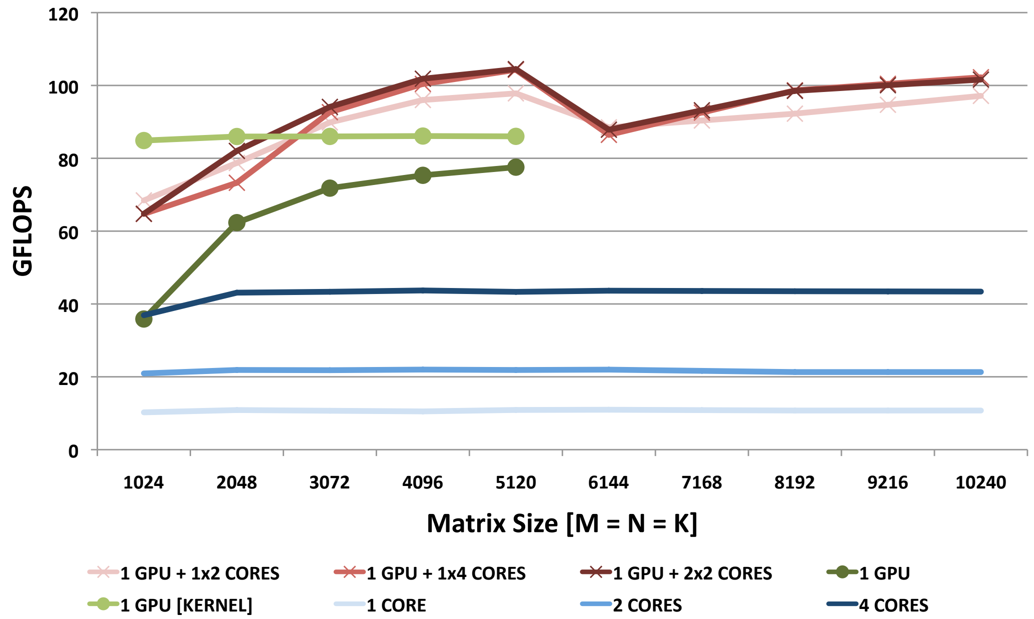
Parallel implementation requires inevitable **transpositions**

- between FFTs applied on different dimensions and after executing the final FFT
- List of 4 different and dependent tasks to be scheduled one after another:
  1. **2D FFT Batch** – Divisible and/or Agglomerative; 1D Job Queue of the size N_1
  2. *Transposition* – Depending on the matrix storage method (In-situ/Out-of-place, parallel/sequential)
  3. **1D FFT Batch** – Divisible and/or Agglomerative; 2D Job Queue of the size $N_2 x N_3$
  4. *Transposition* – To bring back the original matrix layout

Experimental Results: Dense Matrix Multiplication

technology from seed

Experimental Setup	CPU	GPU
	Intel Core 2 Quad	nVIDIA GeForce 285GTX
Speed/Core (GHz)	2.83	1.476
Global Memory (MB)	4096	1024
High Performance Software		
Matrix Multiplication	Intel MKL 10.1	CUBLAS 3.0
FFT		CUFFT 3.0

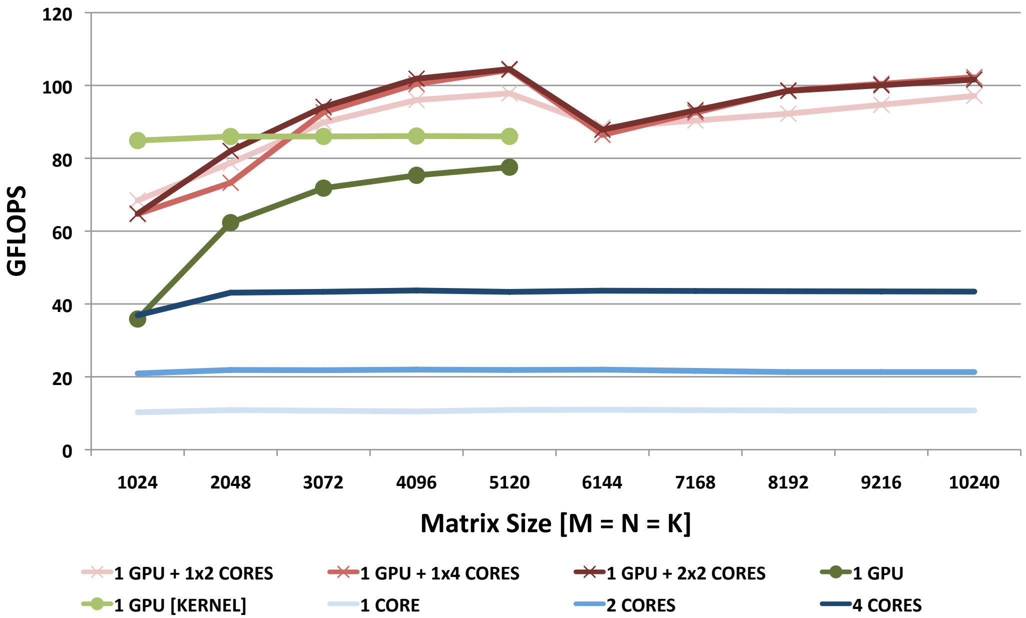


without nested parallelism

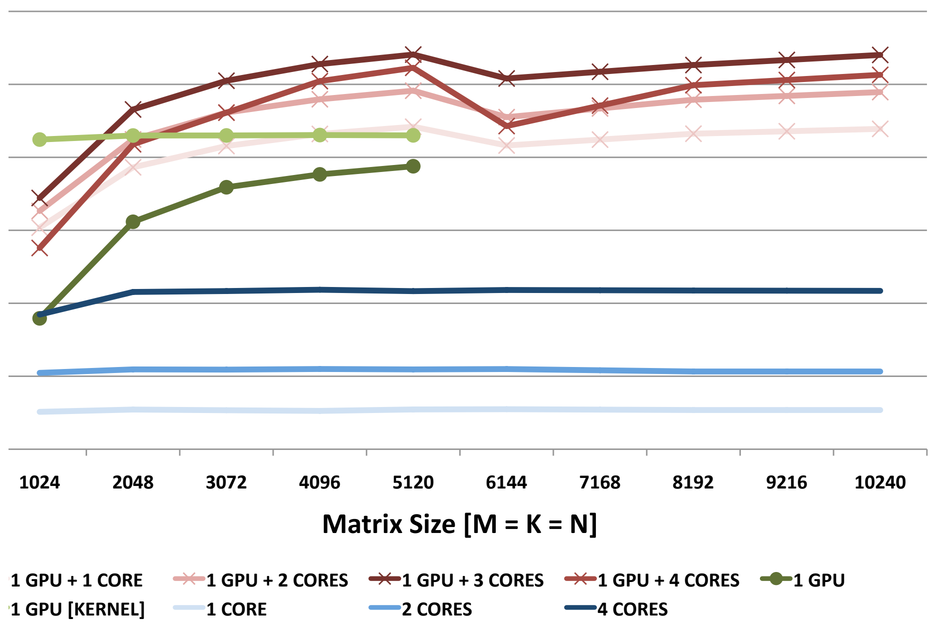
- Double-precision float-point arithmetic
- No modifications to the original high-performance libraries
- Load Balancing via exhaustive search
- CHPS **outperforms** both GPU-only and 4-core CPU execution

Experimental Results: Dense Matrix Multiplication

technology
from seed



without nested parallelism



with nested parallelism

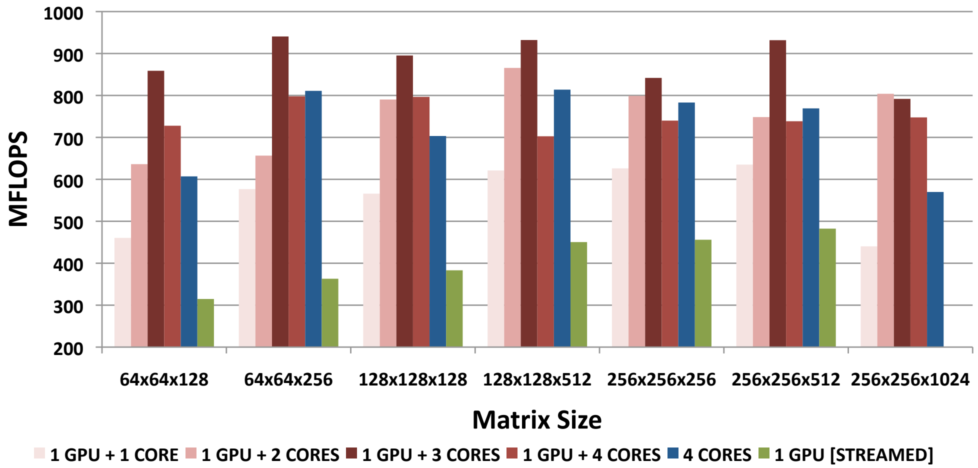
- Double-precision float-point arithmetic
- No modifications to the original high-performance libraries
- Load Balancing via exhaustive search
- CHPS **outperforms** both GPU-only and 4-core CPU execution

Experimental Results: 2D FFT Batch

technology from seed

inesc id lisboa

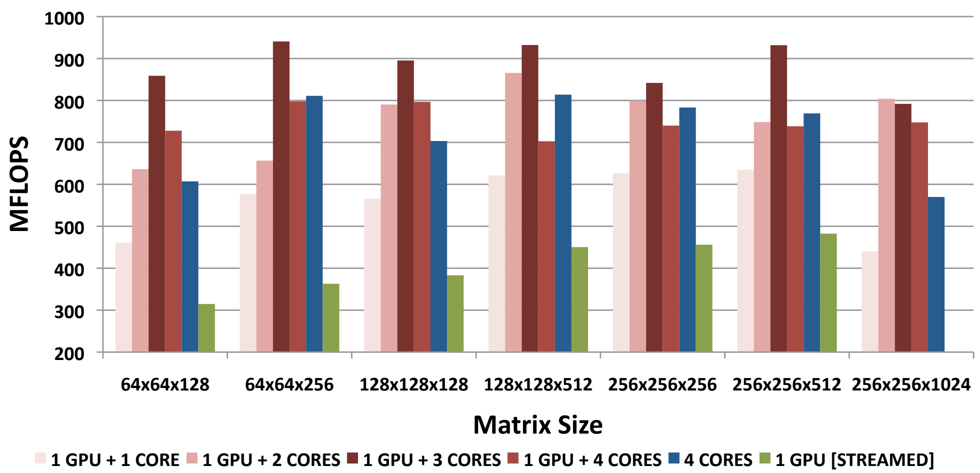
Experimental Setup	CPU	GPU
	Intel Core 2 Quad	nVIDIA GeForce 285GTX
Speed/Core (GHz)	2.83	1.476
Global Memory (MB)	4096	1024
High Performance Software		
Matrix Multiplication	Intel MKL 10.1	CUBLAS 3.0
FFT		CUFFT 3.0



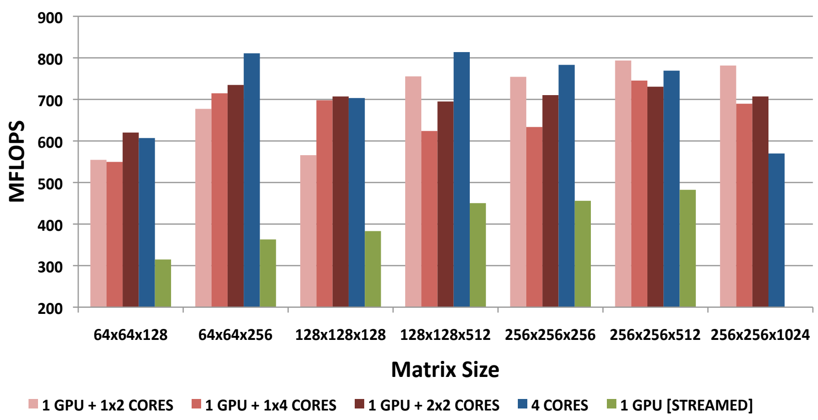
without nested parallelism

- Double-precision complex arithmetic
- Optimizations:
  - Data allocated in **pinned** (page-locked) **memory** regions
  - Communication overlapped with the computation using **CUDA streams** (exhaustive search to find the optimal number of streams)

Experimental Results: 2D FFT Batch



without nested parallelism

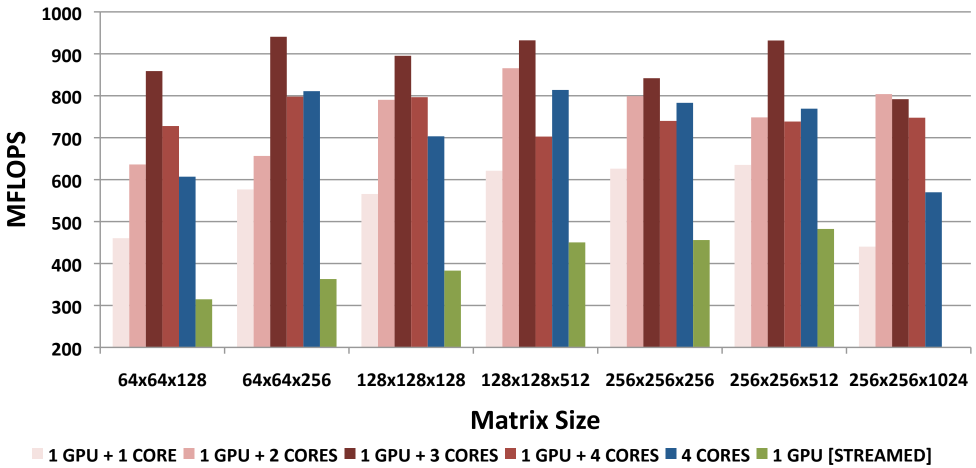


with nested parallelism

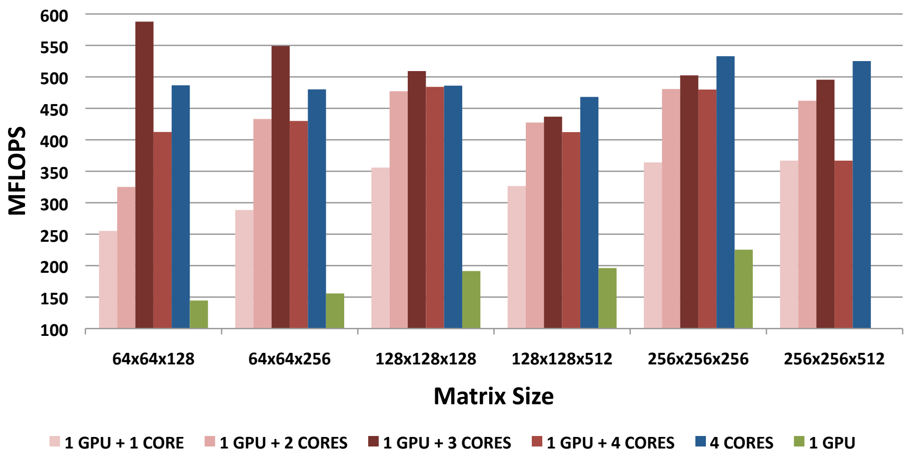
- "*Slight*" performance gains for 2D FFT implementation
- High instability of results for nested parallelism
  - Limited ability of memory subsystem to serve both FSB and PCIe requests at the same time

Experimental Results: 2D & 1D FFT Batches

technology
from seed



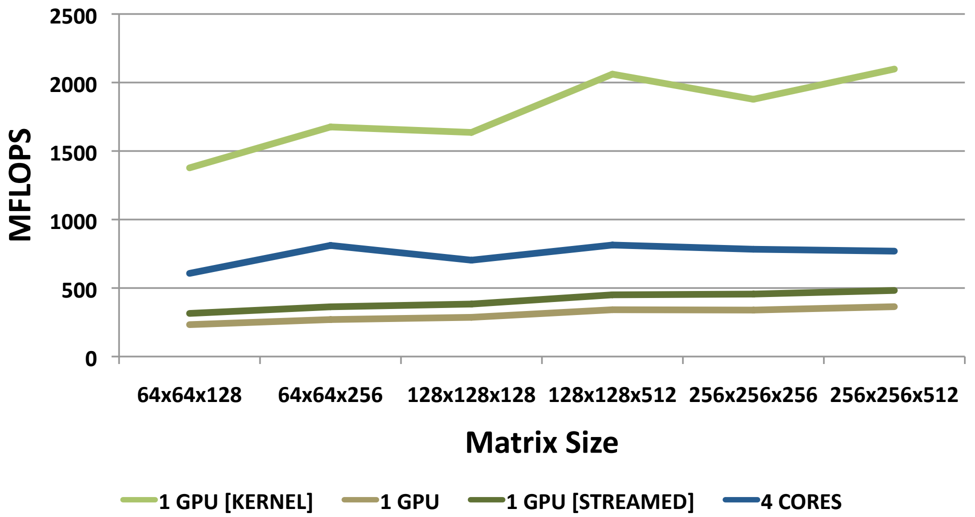
2D FFT without nested parallelism



1D FFT without nested parallelism

- Double-precision floating-point complex arithmetic
- Optimizations:
  - Data allocated in **pinned** (page-locked) **memory** regions
  - Communication overlapped with the computation using **CUDA streams** (exhaustive search to find the optimal number of streams)

Experimental Results: Memory transfers (2D FFT)



- Limited interconnection **bandwidth**
  - Execution **bottleneck** in the tested environment
- 3D FFT parallel execution
  - With transposition times included, no performance gains are expected in the tested environment

Conclusions and Future work

The proposed *unified execution environment*

- exploits both **task** and **data parallelism (+ nested)**
- significant **performance gains** for matrix multiplication
- interconnection bandwidth limits the performance of FFT batches

- **Future work**:
  - Systems with higher level of heterogeneity (more GPUs, FPGAs, or special-purpose accelerators)
  - Performance **modeling** and application **self-tuning**
  - Adoption of advanced **scheduling policies**
  - Identification of performance limiting factors to accommodate on-the-fly device selection (e.g GPU vs. CPU)

Questions?

Thank you