Enhancing productivity on heterogeneous supercomputers with task-based programming model

Adrien Roussel, Mickael Boichot, Romain Pereira, Manuel Ferat

Enhancing Productivity on Heterogeneous Supercomputers with Task-based Programming Model

Adrien Roussel, Mickael Boichot, Manuel Ferat, Romain Pereira

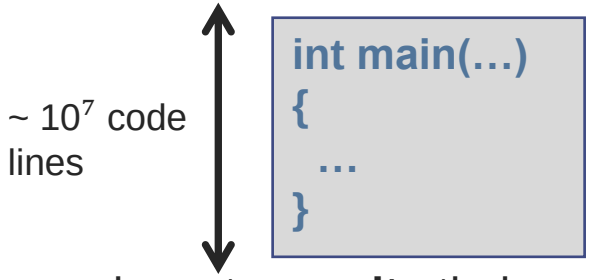
CEA, DAM, DIF

SIAM CSE, Amsterdam

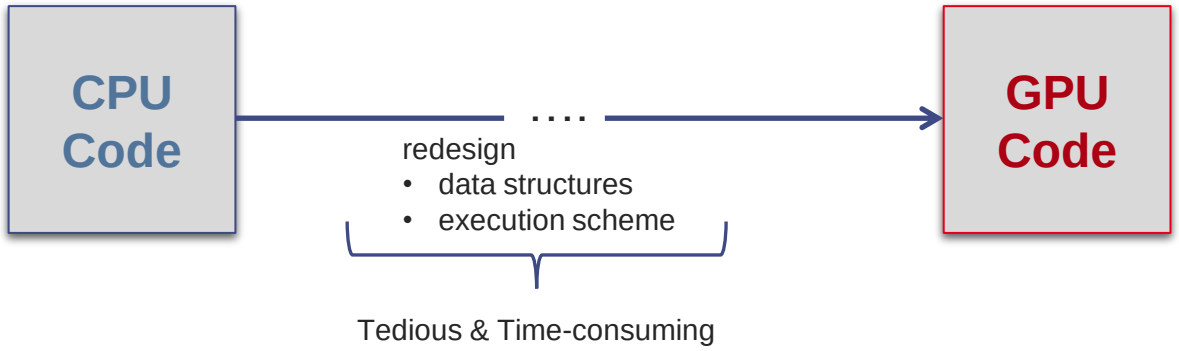
February 28, 2023

Introduction

- Most of the time, scientific applications are already parallelized for **CPU** (or partly for GPU)



- To take advantage of **GPUs**, developers have to **rewrite** their codes (in part? in whole?)



- Goals :
  - Build a **preliminary analysis** to determine the **potential** of the application for a **GPU port**
  - Port mini-application to **heterogeneous architectures** with this information

Contributions

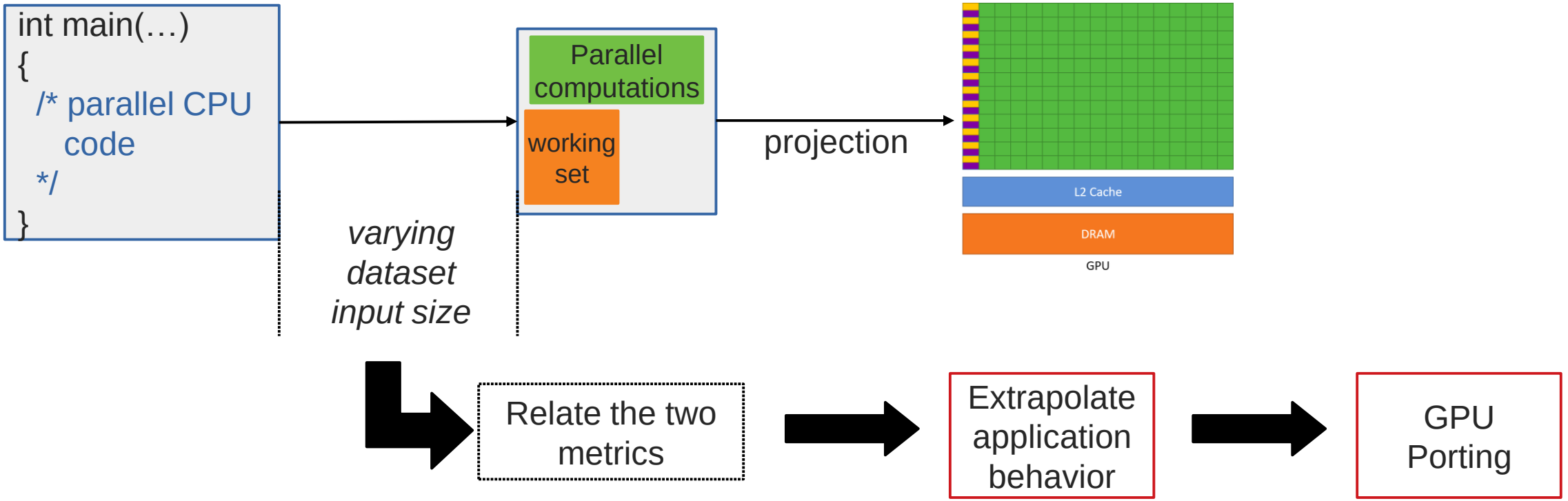
Goal : Inform the user

- that there exists a **test case** of the application **suitable** for a **GPU port**
- about the **strengths** and **weaknesses** of the application for a **GPU port**

- We analyzed the variations of two **metrics:**
  - the **degree of parallelism** (→ saturate GPU cores)
  - and the ***working set*** (→ bounded GPU memory)

→

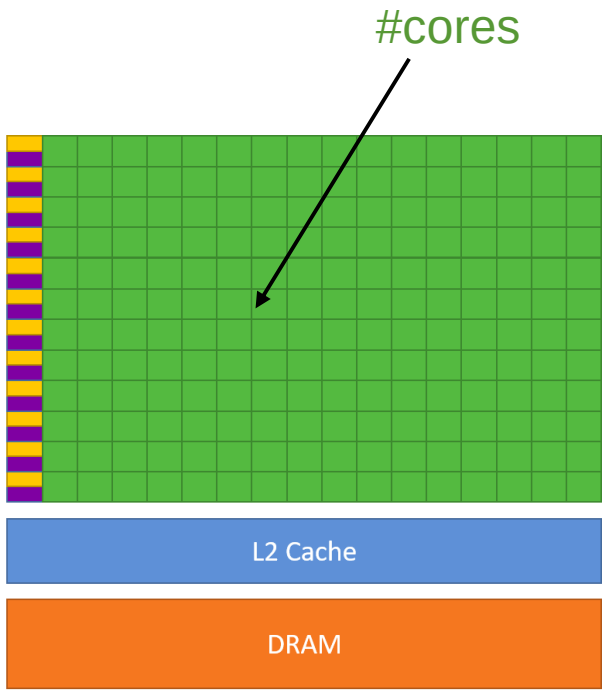
- Porting app. To GPU Computing
  - Using Task based prog. Model
    - MPI+OpenMP (tasks + target)



Contents

2. Application Characterization Method

Targeted GPU Characterization



Targeted GPU

Characterization

Required Parallelism (RP)

RAM (GB)

characteristics

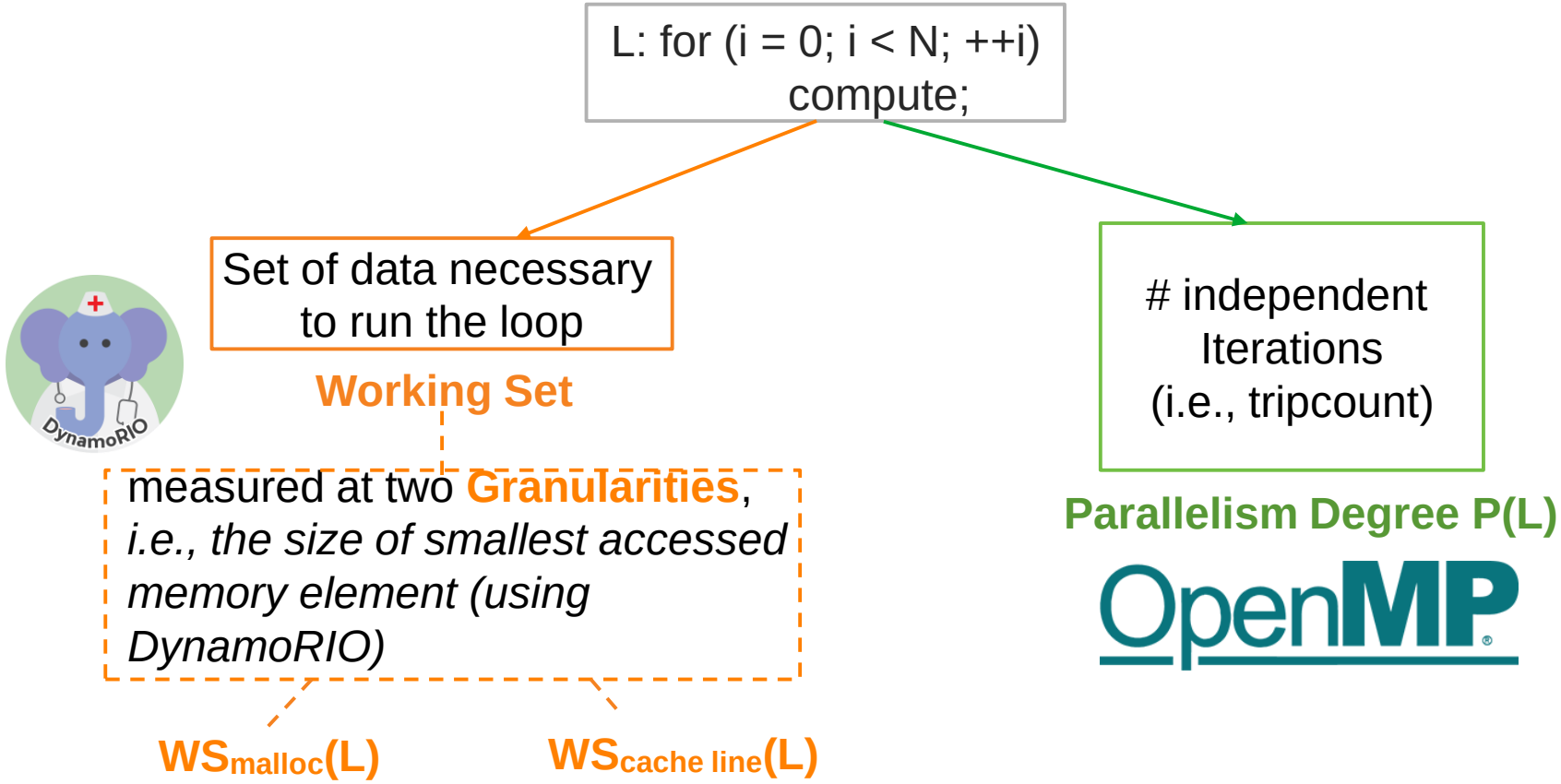
Maximizing FFMA throughput (“floating point fused multiple add”)
→ ***RP*** = #cores . L
where

- L : 32 bits FFMA instruction latency

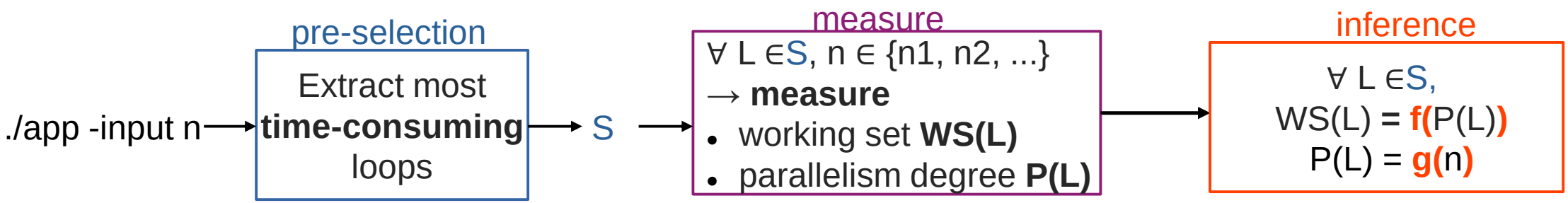
Already known (GPU documentation)

Metrics to Characterize the Application

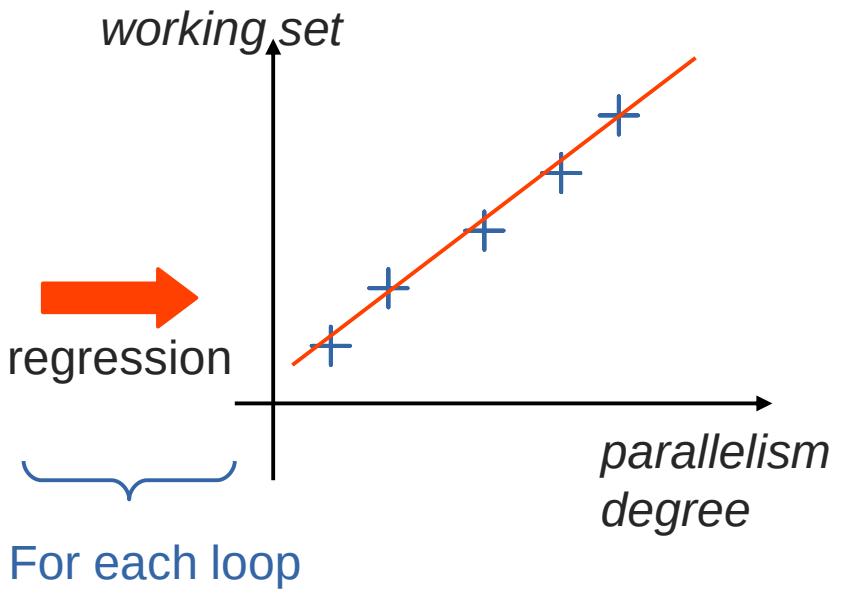
- **Applications analyzed at Loop Granularity** (already parallelized with OpenMP)



Extrapolation Methodology



loop	*working set*	*parallelism degree*	input size
L	WS(L)	P(L)	n_1
L	WS(L)	P(L)	n_2
...	...	...	...



Extrapolation Methodology

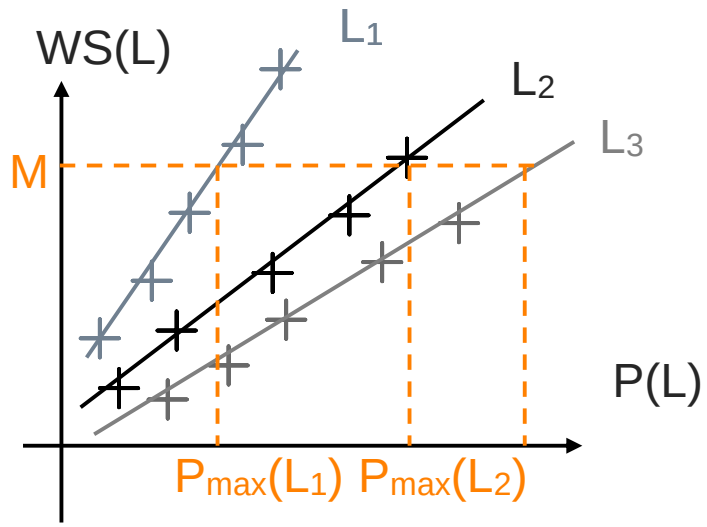
./app -input n → pre-selection: Extract most **time-consuming** loops → S

measure:
∀ L ∈S, n ∈ {n1, n2, ...}
→ **measure**
- working set **WS(L)**
- parallelism degree **P(L)**

inference:
∀ L ∈S,
WS(L) = **f**(P(L))
P(L) = **g**(n)

target GPU:
Required Parallelism (RP)
RAM : M GB

projection:



Test cases fitting target GPU memory and exposing enough parallelism

3. Proxy Applications Analysis

LULESH, miniFE

Experimental Protocol

- Evaluation Machine

processor	Intel Xeon E5-2680
#sockets	2
#cores/socket	8
RAM (GB)	128
OS	Linux 3.10
Compiler	Intel 20.0.0

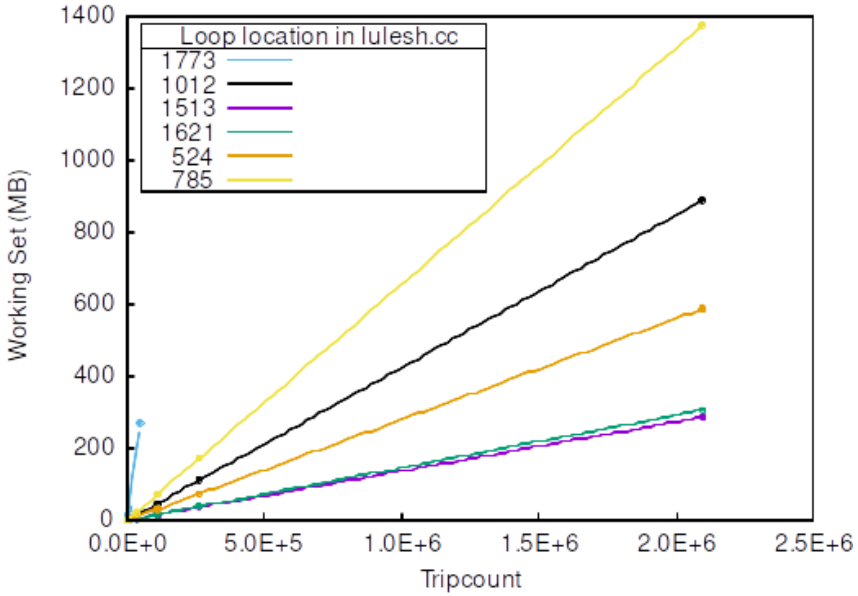
- Evaluated mini-applications
  - **LULESH** (hydrodynamic simulation)
  - **MiniFE** (finite element problem)
  - Quicksilver (Monte Carlo particle transport)
  - XSBench (Monte Carlo particle transport kernel)

All these applications have CPU OpenMP, GPU CUDA and HIP versions.

- Considered targeted GPUs

GPU	RAM	#cores	minimal required parallelism RP
V100	16 GB	5120	20480
A100 40GB	40 GB	6912	27648
A100 80GB	80 GB	6912	27648
MI100	32 GB	7680	30720

LULESH



Working set of LULESH selected loops as a function of their tripcount

	working set granularity					
	cache line			malloc		
target GPU	s_{max}	ρ (loop 1773)	ρ (other loops)	s_{max}	ρ (loop 1773)	ρ (other loops)
V100 16 GB	285	19.2	1130.3	289	19.4	1178.6
A100 40GB	387	19.3	2099.4	393	19.6	2195.4
A100 80GB	487	24.3	4198.7	495	24.7	4386.8
MI100 32GB	359	16.1	1511.5	365	16.4	1586.6

Projection of dataset input size s_{max} saturating GPU memory and corresponding ρ ratio
(ρ(L) = P(L) / RP)

- Sufficient exposed parallelism (ρ ≫ 1)
- Limiting factor : memory size
- Advised data transfers granularity : malloc

- *Similar conclusions for **miniFE***

4. Porting Proxy App to GPU: LULESH

Porting LULESH to MPI+OpenMP(tasks)

- **LULESH**
  - Hydrodynamic problems solver - **Iterative** simulation
  - Original code express **loop-based parallelism (parallel-for)**
- **Porting to MPI+OpenMP (tasks)**
  - Loops splitted into **tasks** with **dependencies**
  - Single producer/multi-consumer scheme
  - MPI **communications** nested within OpenMP tasks
    - Overlap communications by computations

```
1    # pragma omp parallel for
2    for (int i = 0 ; i < n_nodes; i++)
3        work_on_nodes(mesh->nodes[i]);
4
5    # pragma omp parallel for
6    for (int i = 0 ; i < n_cells; i++)
7        work_on_cells(mesh->cells[i]);
```

```
1    for (int j = 0 ; j < n_nodes; j += BS)
2    {
3        # pragma omp task depend(out : mesh->nodes[j])
4        {
5            for (int i = j ; i < j + BS ; ++i)
6                work_on_nodes(mesh->nodes[i]);
7        }
8     }
9
10   for (int j = 0 ; j < n_cells; j += BS)
11   {
12       # pragma omp task depend(in : […])
13       {
14           for (int i = j ; i < j + BS ; ++i)
15               work_on_cells(mesh->cells[i]);
16       }
17   }
```

Concept Generalization for Heterogeneous Architectures with OpenMP Target

- **Porting to MPI+OpenMP (tasks + GPU target)**
  - GPU « super-tasks » merging several CPU tasks
  - From CPU tasks to GPU offloading while enhancing asynchronism
    - Sending/Retrieving data to/from GPU with ‘update’ clause
    - Executing GPU kernels through ‘nowait’ clause
  - Offloading **6 loops given by application characterization**
    - IntegrateStressForElems, CalcKinematicsForElems, CalcAccelerationForNodes, CalcFBHourglassForceForElems1, CalcHourglassControlForElems, CalcMonotonicQGradientsForElems
  - Use of **cudaMallocHost** for the GPU buffers
    - Pinned memory to enable asynchronous execution

```
1  for (int j = 0 ; j < n_nodes; j += BS)
2  {
3      # pragma omp task depend(out : mesh->nodes[j])
4      {
5          for (int i = j ; i < j + BS ; ++i)
6              work_on_nodes(mesh->nodes[i], buffer[i]);
7      }
8  }
```

```
1  buffer = cudaMallocHost(…) ;

   […]

2  for (int j = 0 ; j < n_nodes; j += BS)
3  {
4      # pragma omp target update to(buffer[j:BS]) \
5                depend(out : mesh->nodes[j])
6
7      # pragma omp target \
8                      teams distribute parallel for nowait \
9                      depend(iterator(i=0:N_BS), out: mesh->nodes[j+i*BS])
10     {
11         for (int i = j ; i < j + N_BS * BS ; ++i)
12            work_on_nodes(mesh->nodes[i], buffer[i]);
13     }
14
15      # pragma omp target update from(buffer[j:BS]) \
16                     depend(out: mesh->nodes[j])
17 }
```

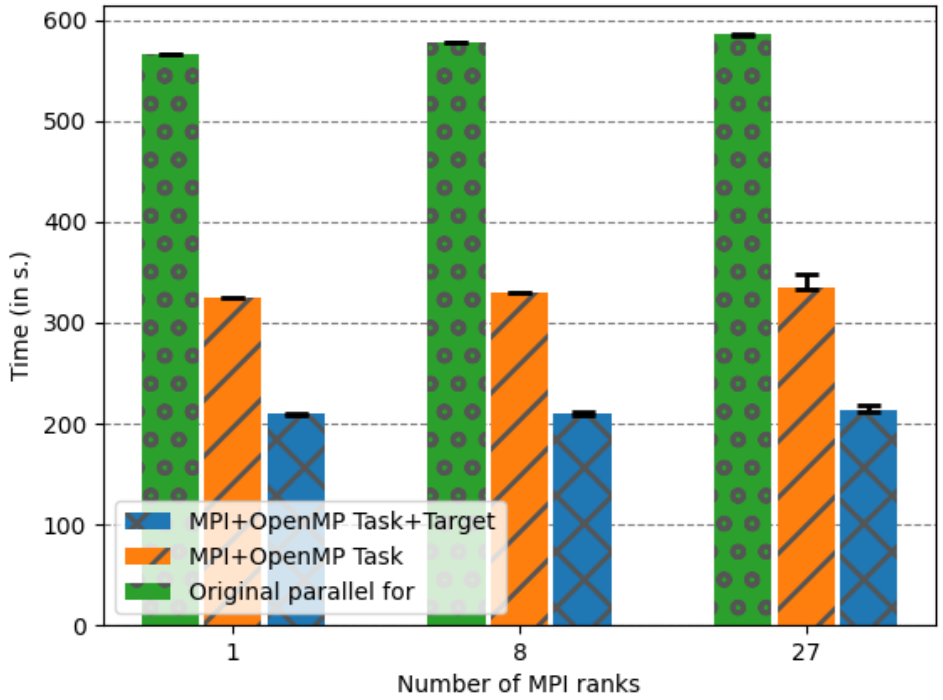
Experimental Protocol

Processor	AMD EPYC 7H12
#sockets	2
#cores/socket	64
GPU	NVIDIA A100
#GPU/socket	2
RAM (GB)	256

MPC-OpenMP (No Offloaded part)	4.1.0
LLVM (Offloaded part)	16.x
Open MPI	4.0.5

LULESH Performance Evaluation

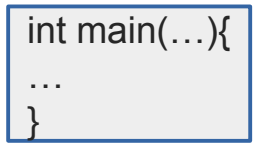
- Problem size
  - 264^3 elements
    - 80% of the NUMA domain memory capacity
    - 20% of the GPU memory capacity
  - 128 iterations
- Placement
  - Per MPI Process:
    - 16 threads (on 16 cores)
    - + 1 GPU
- Weak scaling
  - Median runtime of 10 runs
  - Each version scale very well
  - **OpenMP tasks** version significantly improves performances due to **cache reuse**
  - Significant performance gain with **GPU offloading**

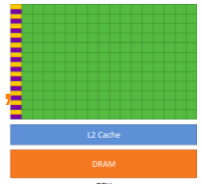


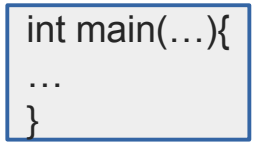
	Median Time (s)		
ranks	parallel-for	no-offloading	offloading
1	565.84	325.02	209.31
8	572.45	331.09	209.43
27	581.26	333.98	212.78

5. Conclusion

Conclusion

- An **approach** : (

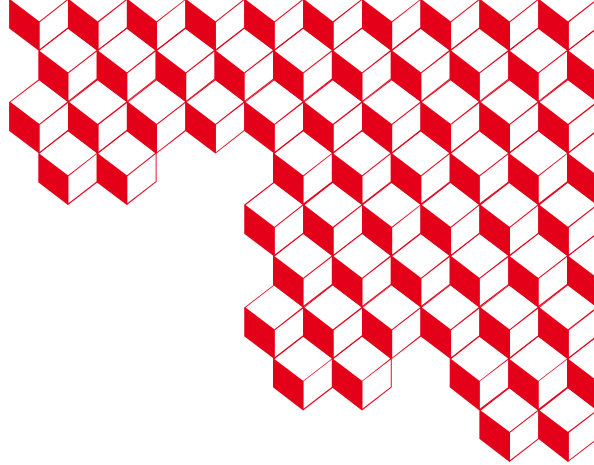


) ↦ { test cases such that


fit on RAM
and do not lack parallelism
}

  - **complementary** to existing work (analysis of several application runs → extrapolation)

- We have **evaluated** our approach and shown that we can **qualitatively guide developer** in their GPU porting
- We have **ported** LULESH on heterogeneous machine with a **task-based programming model**
  - Significant gains are observed



Thank you!

Questions?