Duksu Kim

- **Assistant professor, KORATEHC**
- **Education**
  - **Ph.D**. Computer Science, KAIST
    - Parallel Proximity Computation on Heterogeneous Computing Systems for Graphics Applications
- **Professional Experience**
  - **Senior researcher**, KISTI
    - 2014.07 – 2018.02
    - High performance visualization
- **Awards**
  - **The Spotlight Paper**, IEEE TVCG (Sept., 2013)
  - **Distinguished Paper Award**, Pacific Graphics 2009
  - **CUDA Coding Contest**
    - **2nd place**, NVIDIA Korea 2015
    - **Best programming award**, NVIDIA Korea 2010
  - **Student stipend award**, ACM symposium on Interactive 3D Graphics and Games, 2009

KSC 2018 Tutorial

Background on Heterogeneous Computing

Duksu Kim

Outline

- **Parallel Computing Architectures**
    - **Multi-core CPU** and **GPU**

- **Heterogeneous Parallel Computing**
    - Heterogeneous computing system
    - Heterogeneous parallel algorithm

- **Tools for Heterogeneous Computing**

Parallel Computing Architecture

- **Flynn's Taxonomy**

Single core processor

Vector processor

SISD Single instruction stream Single data stream	**SIMD** Single instruction stream Multiple data stream
MISD Multiple instruction stream Single data stream	**MIMD** Multiple instruction stream Multiple data stream

Not covered

Multi-core processor

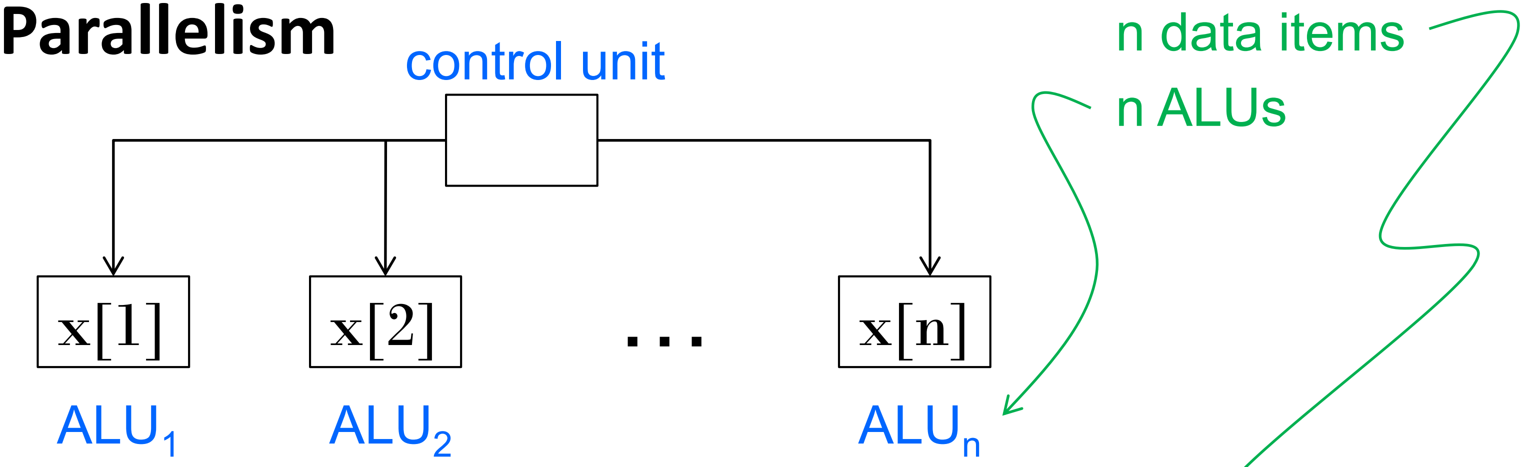
MIMD

- **M**ultiple **I**nstruction, **M**ultiple **D**ata
  - 여러 개의 명령어를 각각의 데이터에 적용

- **A set of independent processors**
  - E.g., Multi-core CPUs (up to 64 cores)
- **Thread-level parallelism**

SIMD

- **S**ingle **I**nstruction, **M**ultiple **D**ata
  - 하나의 명령어를 여러 개의 데이터에 적용
- **Data Parallelism**



```
for (i = 0; i < n; i++)
    x[i] += y[i];
```

SIMD

- **S**ingle **I**nstruction, **M**ultiple **D**ata
  - 하나의 명령어를 여러 개의 데이터에 적용
- **Data Parallelism**
- **E.g., 4 ALUs, 15 data**

Round	ALU_1	ALU_2	ALU_3	ALU_4
1	X[0]	X[1]	X[2]	X[3]
2	X[4]	X[5]	X[6]	X[7]
3	X[8]	X[9]	X[10]	X[11]
4	X[12]	X[13]	X[14]	

Vector Processors

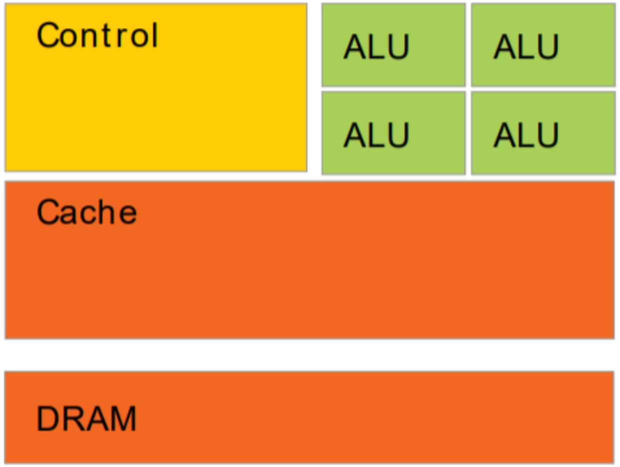
- **Work with a vector (or data array)**
- **Typical examples of SIMD architecture**
    - E.g., MXX/SSE/AVX(x86), XeonPhi, **GPU (SIMT)**

SIMT

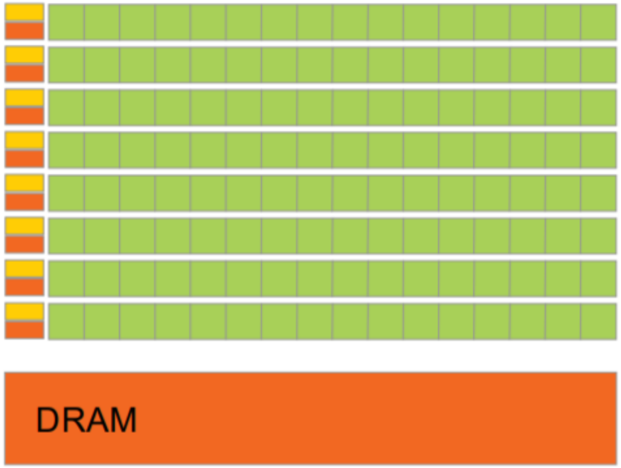
- **The architecture of GPU is called SIMT**
  - Rather than SIMD

- **S**ingle **I**nstruction, **M**ultiple **T**hreads
  - A group of threads is controlled by a control unit
    - E.g. 32 threads (= warp)
  - Each thread has its own control context
    - Different with traditional SIMD
  - Divergent workflow among threads in a group is allowed
    - With a little performance penalty (e.g., work serialization)

Parallel Computing Architecture



VS

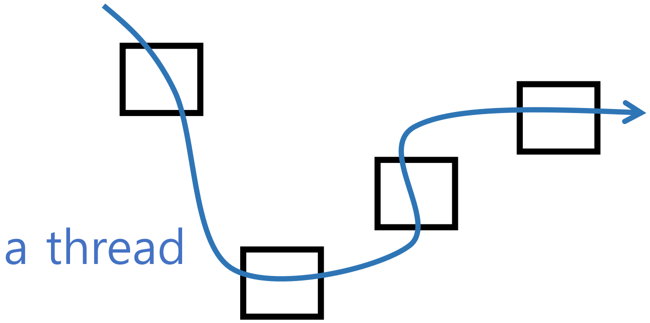


Multi-core CPU

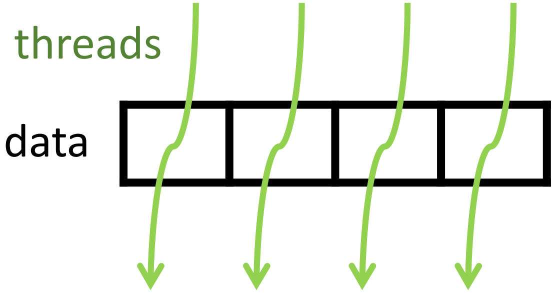
GPU

CPU vs GPU

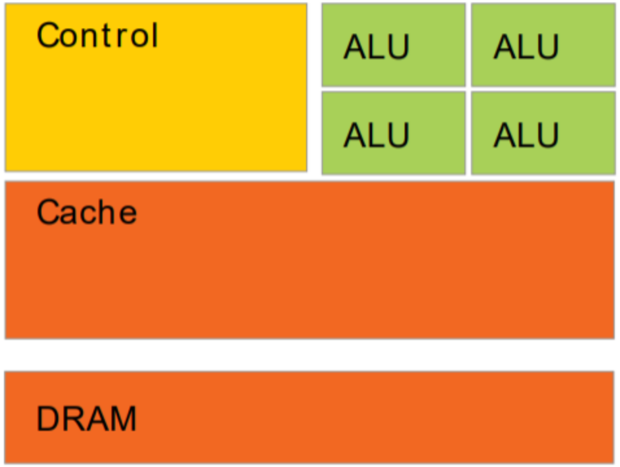
- **General Processing Unit**
  - Focus on the performance of a core
    - Clock frequency, cache, branch prediction, Etc.
- **Single/Multi-core**
  - 1 ~ 32 cores
- **SISD (or MIMD)**
  - Single instruction, Single Data



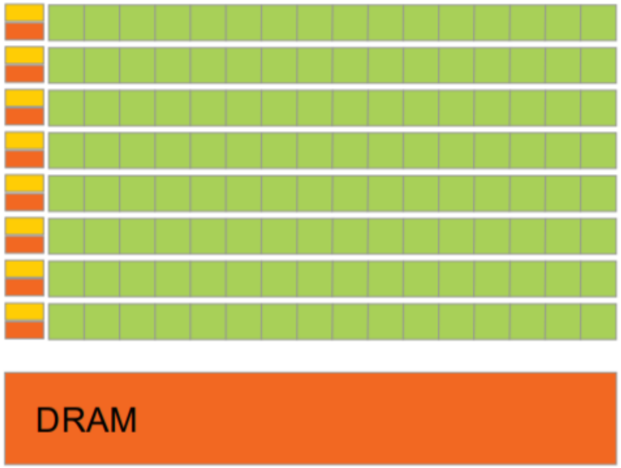
- **Graphics Processing Unit**
  - Focus on parallelization
    - Increasing the # of cores
- **Many core**
  - More than hundreds of cores
- **SIMT**
  - Single instruction, Multiple Threads



CPU vs GPU



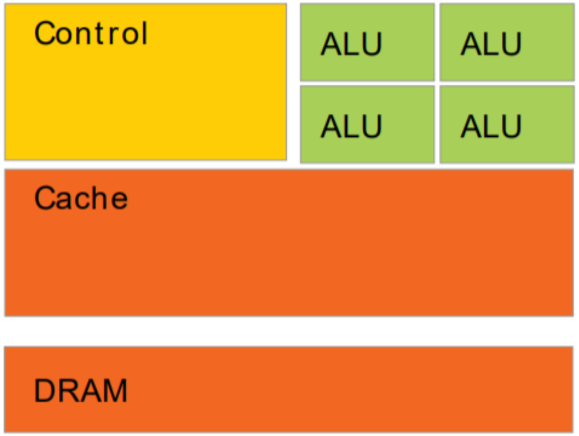
- **Allocate more to**
  - Cache
  - Control
- **Optimized for**
  - Latency
  - Sequential code



- **Allocate more to**
  - Functional units
  - Bandwidth
- **Optimized for**
  - Throughput
  - Streaming code

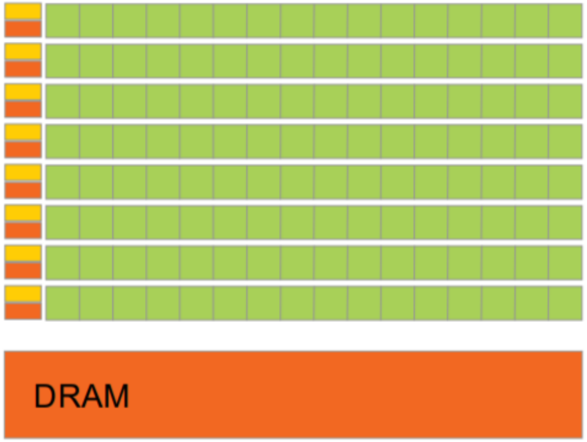
CPU

- **Strength**
  - High performance processing core
  - Efficient **irregular workflow** handling
    - Branch prediction
  - Efficient handling for **random memory access** pattern
    - Well-organized cache hierarchy
  - Large **memory space**
- **Weakness**
  - A small **number of cores** (up to 32)
    - More space for controls
  - Lower **performance** than GPU
    - In a perspective of FLOPS



GPU

- **Strength**
  - A massive **number of cores**
    - But, less powerful than CPU core
  - Much higher **performance** than CPU
    - In a perspective of FLOPS
- **Weakness**
  - Small **memory space**
    - High bandwidth memory = expensive
  - Performance penalty for **irregular workflow**
  - Weak for **random memory access** pattern

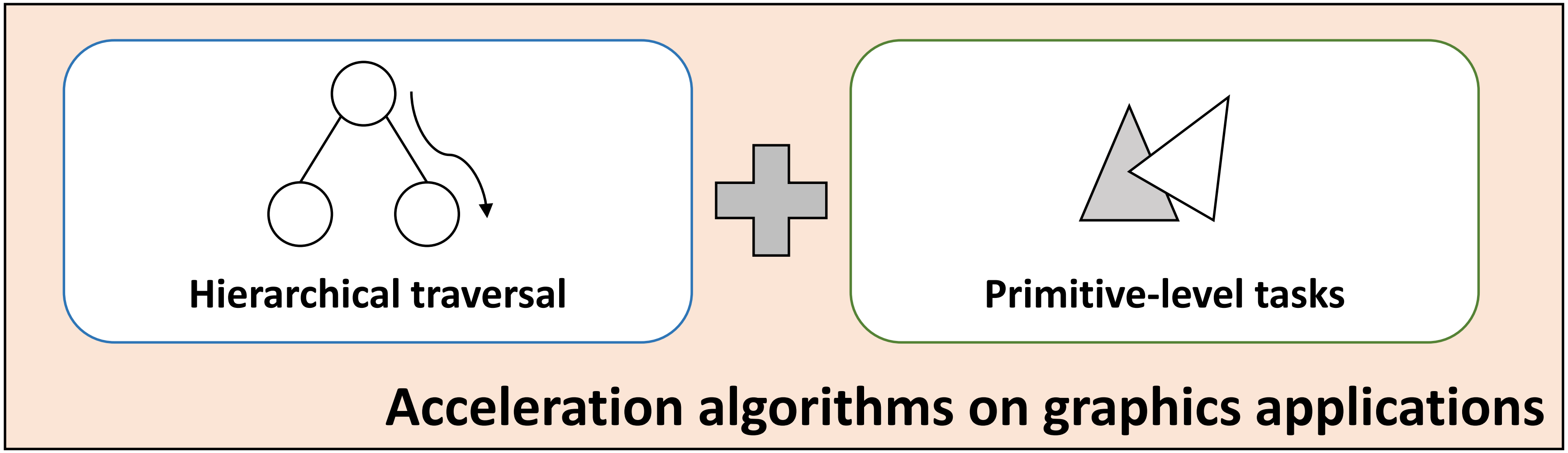


CPU

- Tasks with irregular workflow and random memory access pattern
- Large memory space

GPU

- Compute-intensive and regular streaming tasks
- High performance



Hierarchical traversal

Primitive-level tasks

Acceleration algorithms on graphics applications

Outline

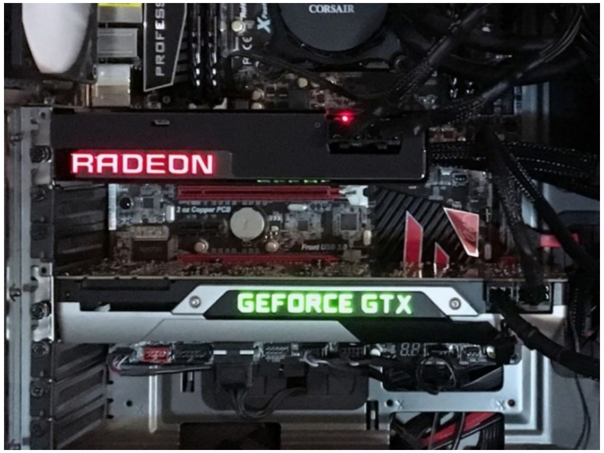
- **Parallel Computing Architectures**
  - Multi-core CPU and GPU

- **Heterogeneous Parallel Computing**
  - Heterogeneous computing system
  - Heterogeneous parallel algorithm

- **Tools for Heterogeneous Computing**

Heterogeneous Computing System

- **A computing system consisting of more than one type of computing resources**
- **Examples**
  - A desktop PC having both multi-core CPUs and GPUs
  - A multi-GPU system consisting of different types of GPUs



Heterogeneous Parallel Algorithm

- **Use multiple heterogeneous computing resources at once for solving a problem**

- **Advantage**
  - Fully utilize all available computing resources
  - Achieve high performance

Issues on Heterogeneous Algo.

- **How to distribute workload to available resources**
  - Workload balance
- **How to reduce communication overhead**

In this tutorial,
we will learn how prior works have solved these issues for proximity computation and rendering.

Outline

- **Parallel Computing Architectures**
  - Multi-core CPU and GPU

- **Heterogeneous Parallel Computing**
  - Technical issues

- **Tools for Heterogeneous Computing**

APIs for using Multi-core CPU

- **Pthreads (POSIX threads)**
  - **함수 라이브러리**
  - **Low-level API**
    - 사용자가 제어
      - 스레드 생성, 분배 등
  - **세밀한 제어 가능 (flexible)**
  - **구현이 복잡함**
    - 처음부터 병렬 알고리즘 작성 필요

- **OpenMP**
  - **지시어(directive)기반**
    - 컴파일러가 전처리 및 병렬 코드 생성
  - **High-level API**
    - 컴파일러 및 런타임의 제어
  - **구현이 간편함**
    - 지시어만 추가하여 serial 코드를 병렬화 가능
  - **제한적 제어 기능**
    - But enough!

- **Windows API, Intel TBB, Etc.**

APIs for using GPUs

- **CUDA**
  - Only support GPUs from Nvidia
  - Highly optimized for Nvidia GPUs
  - More control functions for Nvidia GPUs

NVIDIA
CUDA

- **OpenCL**
  - Support most GPUs (e.g., Nvidia, AMD)
  - Can utilize multiple GPUs with a same code
    - Efficiency is not guaranteed

OpenCL

- **Shader languages, OpenACC, Etc.**

OpenGL
OpenACC
More Science, Less Programming

Multi-core CPUs + GPUs

pthreads

OpenMP

Windows API

+

NVIDIA

CUDA

OpenCL

OpenACC
More Science, Less Programming

OpenGL

Summary

- **Heterogeneous systems are all around!**
  - E.g., multi-core CPUs + GPUs
- **With heterogeneous parallel algorithm,**
  - We can greatly improve the performance of our application

- **To design efficient heterogeneous parallel Algo.,**
  - Understand characteristics of devices and tasks
  - Two common issues
    - Workload balance
    - Communication overhead

Any Questions?

- bluekdct@gmail.com
- http://hpc.koreatech.ac.kr

Heterogeneous Computing on

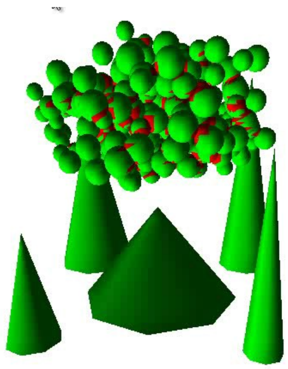
Proximity Computation

KSC 2018 Tutorial

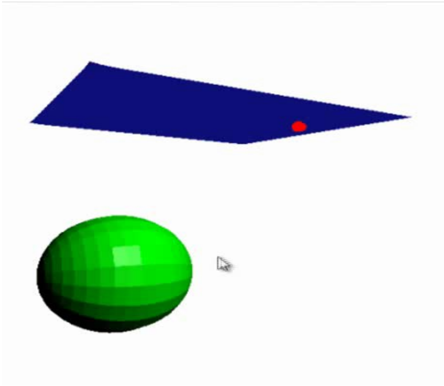
Duksu Kim

Proximity Computation

- **Compute relative placement or configuration of two objects**
  - Collision detection
  - Distance computation
    - Neighbor search



- **Basic operations in various applications**
  - Graphics, simulations, robotics, Etc.

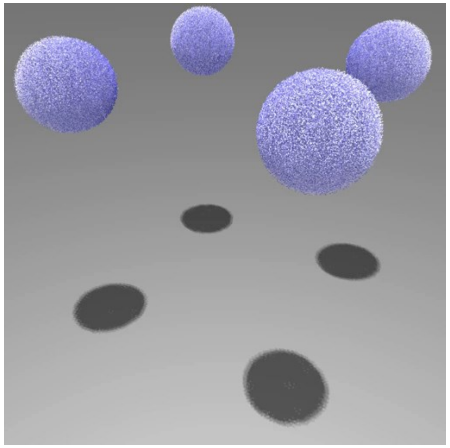


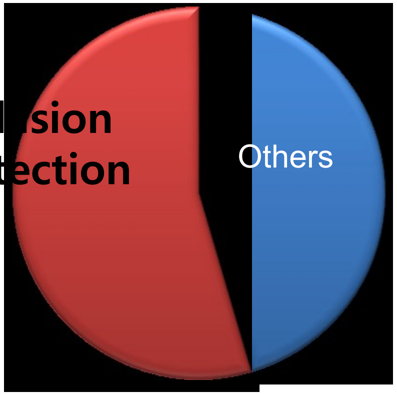
Proximity Computation in App.

Motion planning

Realistic rendering

Particle-based Sim.



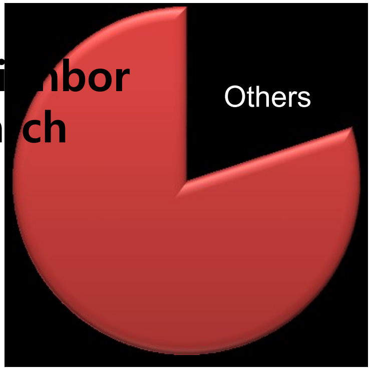


Col sion detection

[Jia 2010] [Liangjun 2008]

Ray racing

[Our in-house render]



Nei hbor sea ch

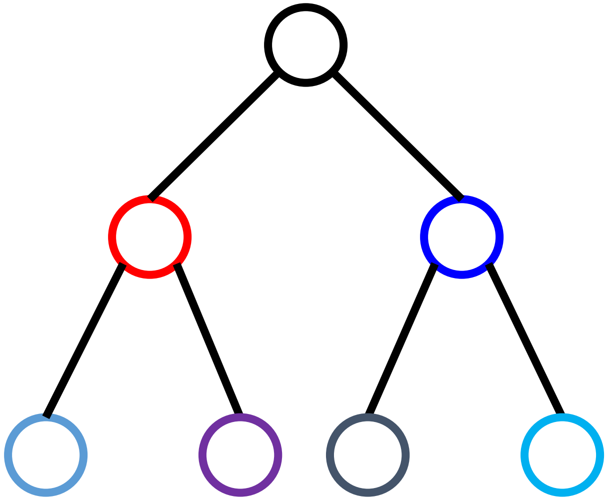
[Our in-house simulator]

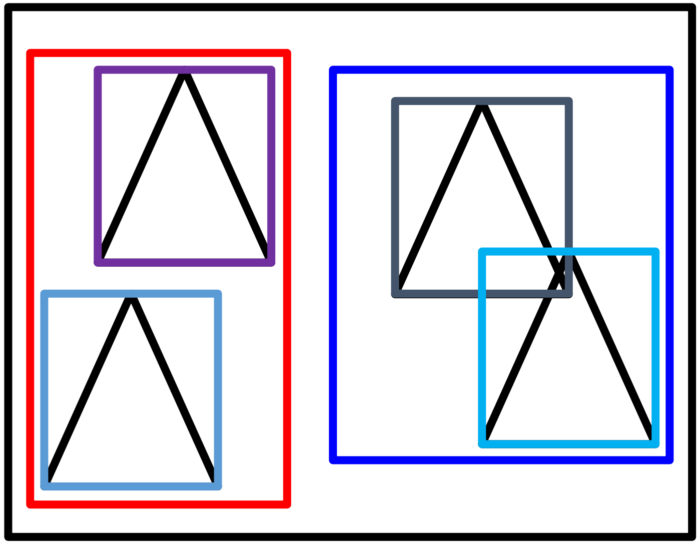
Proximity Computation Acceleration

- **Various acceleration techniques**
  - Acceleration hierarchies
  - Culling algorithms
  - Specialize algorithms for a target application
  - Approximation algorithms
- **Achieve several orders of magnitude performance improvement**

Collision Detection with BVH

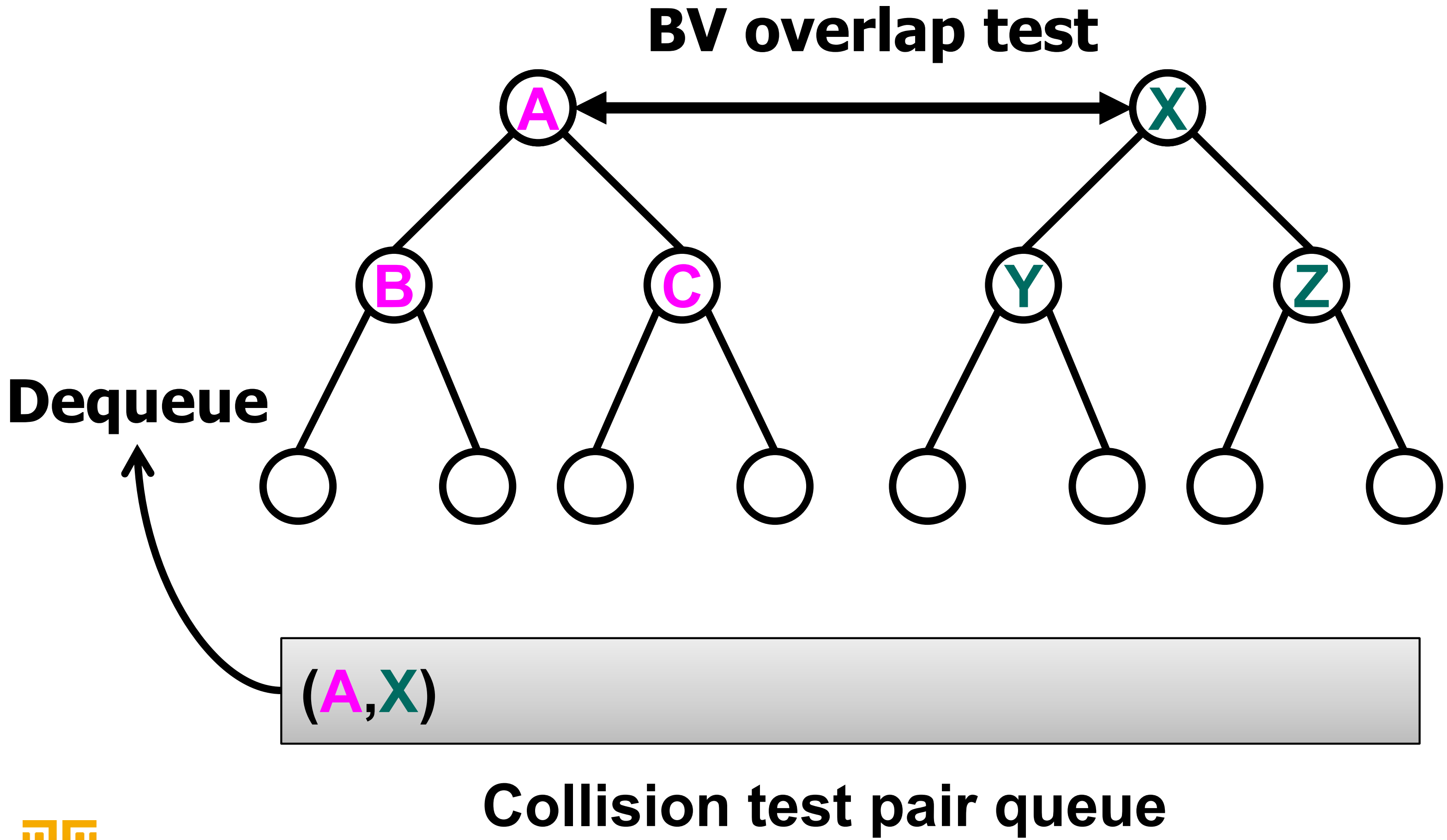
- **Bounding Volume Hierarchy (BVH)**
  - Organize bounding volumes as a tree
  - Leaf nodes have triangles





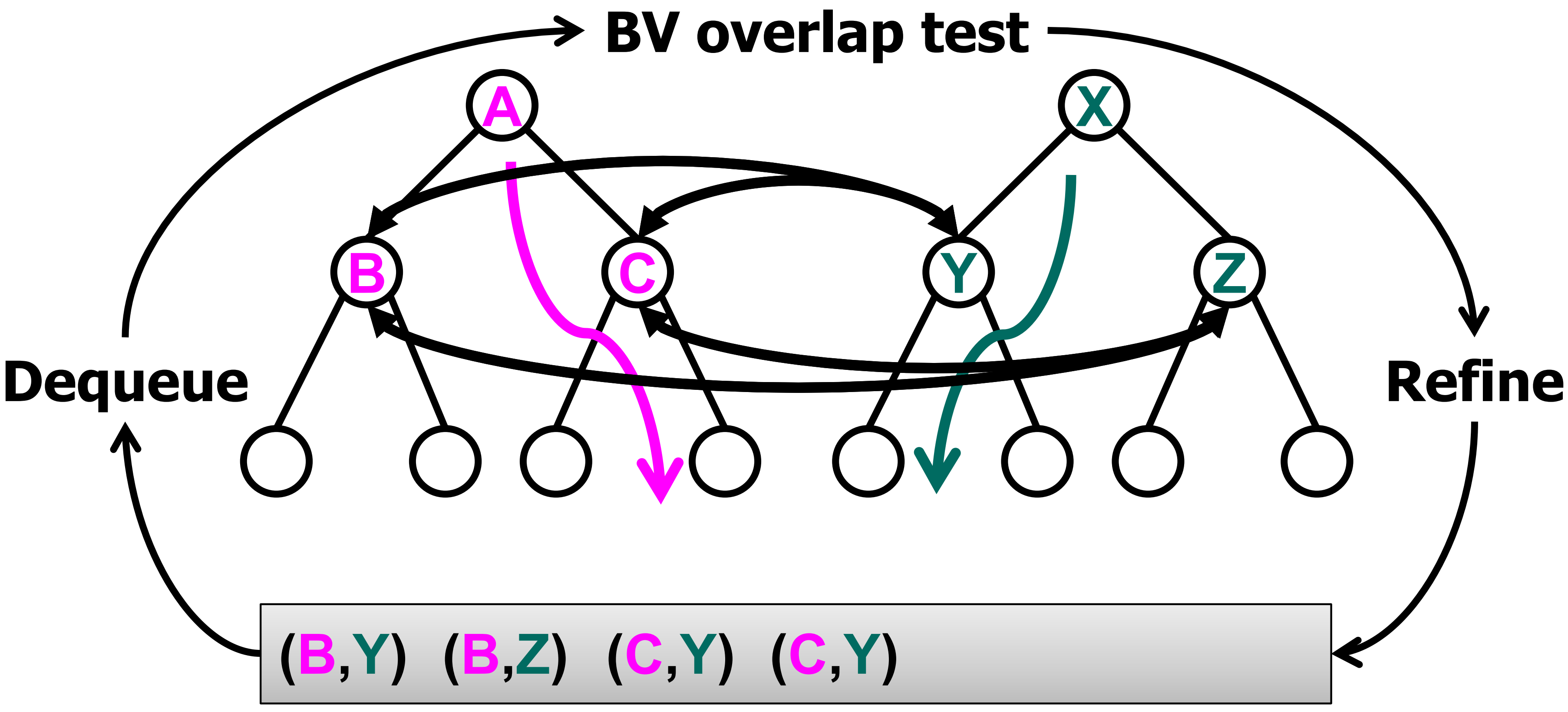
Collision Detection with BVH

- **Hierarchy traversal**



Collision Detection with BVH

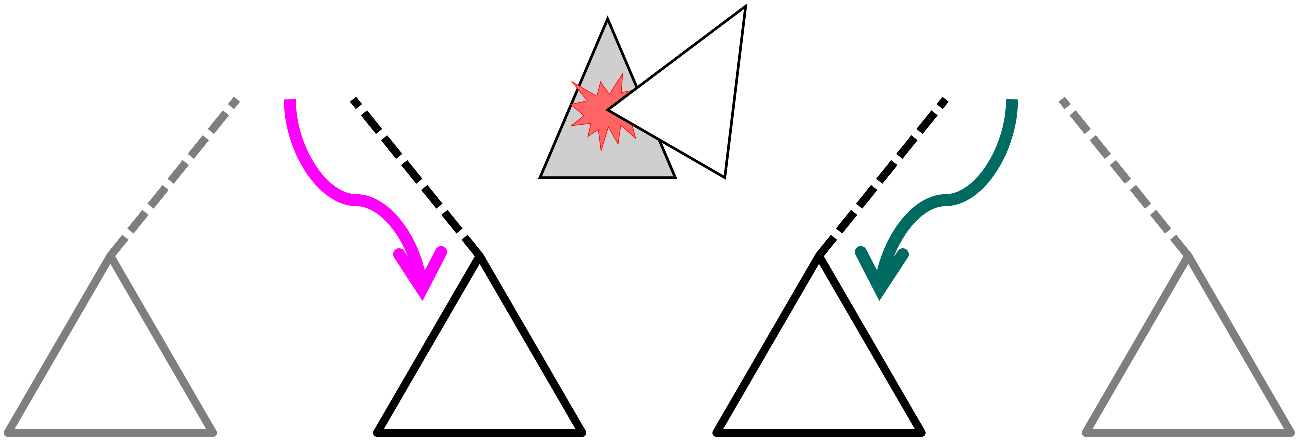
- Hierarchy traversal



Collision test pair queue

Collision Detection with BVH

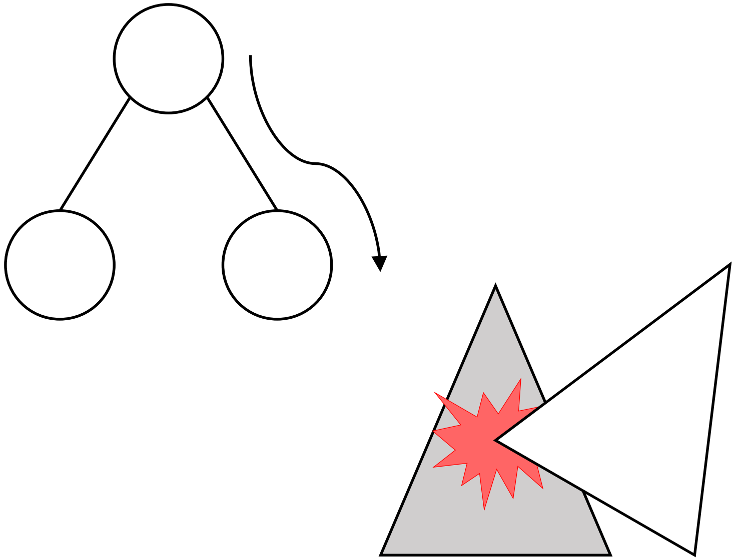
- Hierarchy traversal
- **Primitive-level test**
  - At leaf nodes, exact collision tests between two triangles
    - Solving equations



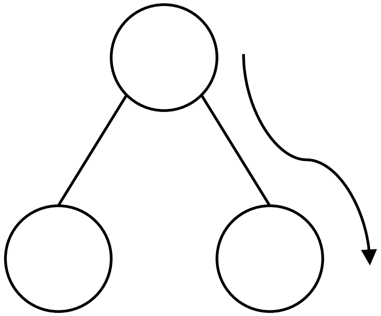
Hierarchy-based Acceleration Algo.

- **Widely used in many applications to improve the performance by reducing search space**

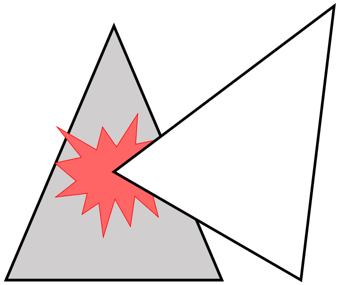
- **Two common task types**
  - Hierarchical traversal
  - Primitive-level test



Two Common Task Types



- **Hierarchical traversal**
  - Many branches
    - Irregular workflow
  - Random memory access pattern



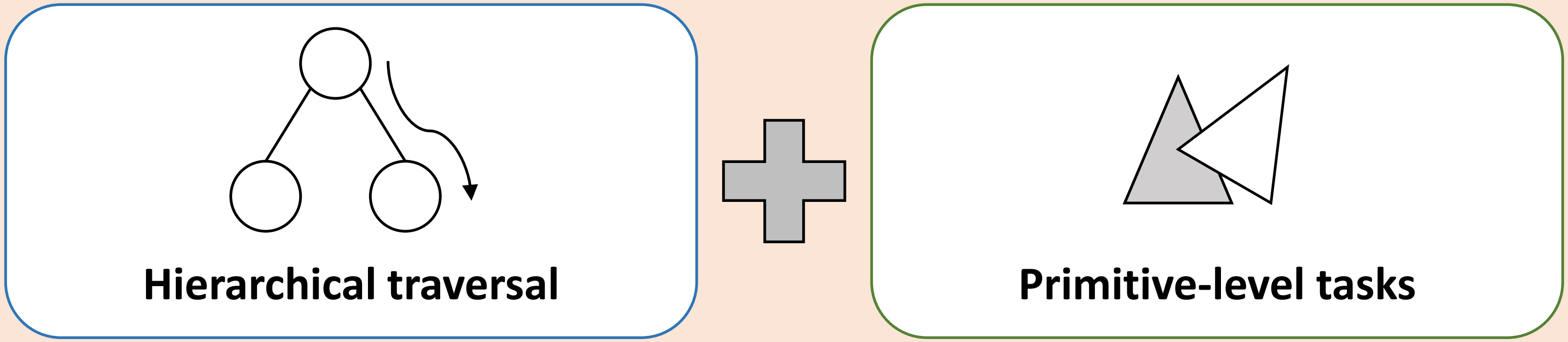
- **Primitive-level test**
  - Compute-intensive work
  - Regular memory access pattern
    - A set of tasks (streaming task)

CPU

- **Tasks with irregular workflow and random memory access pattern**
- **Large memory space**

GPU

- **Compute-intensive and regular streaming tasks**
- **High performance**



Hierarchical traversal

Primitive-level tasks

Acceleration algorithms on graphics applications

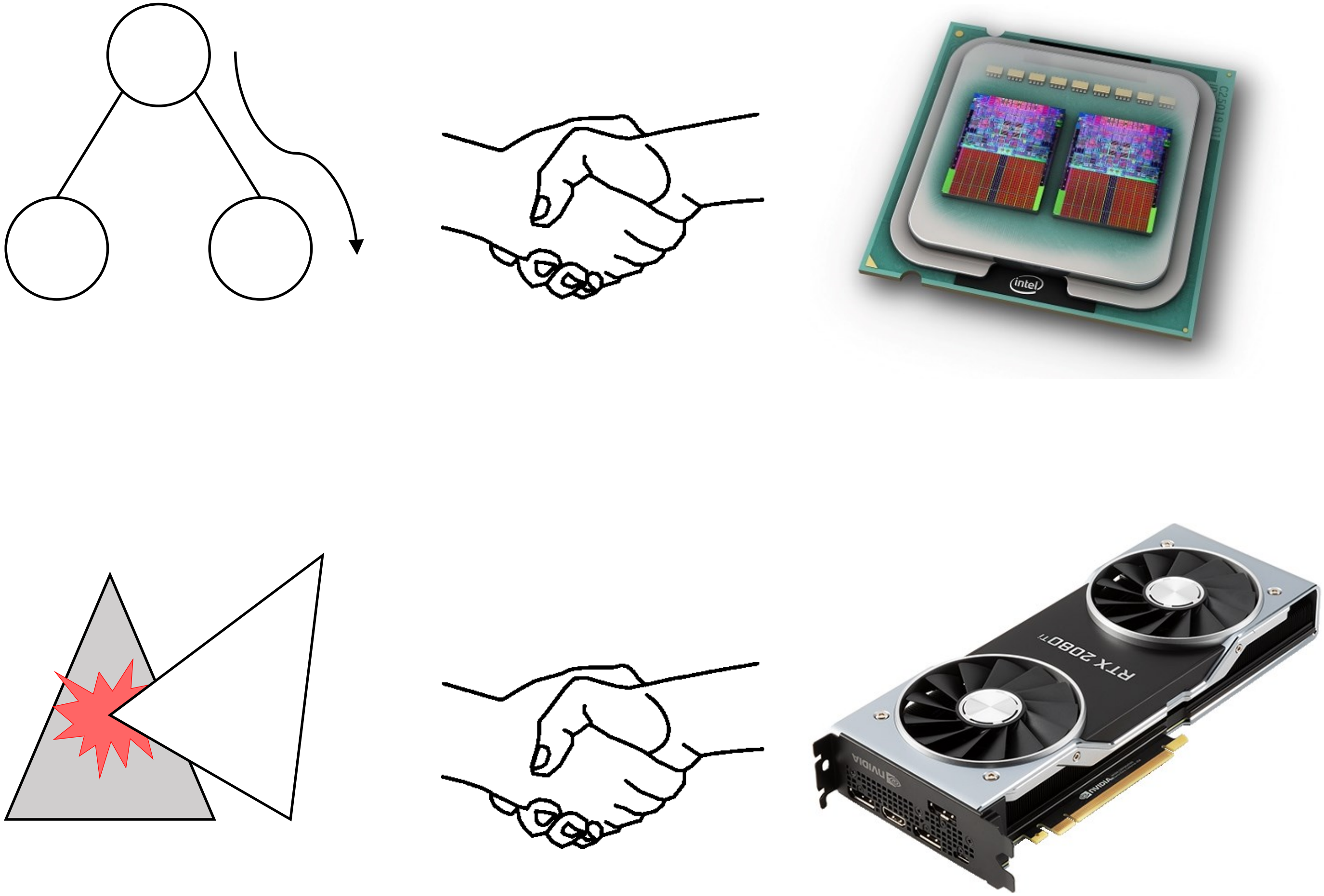
HPCCD:
Hybrid Parallel Continuous Collision Detection using CPUs and GPUs

Duksu Kim, Jae-Pil Heo, JaeHyuk Huh, John Kim, Sung-Eui Yoon

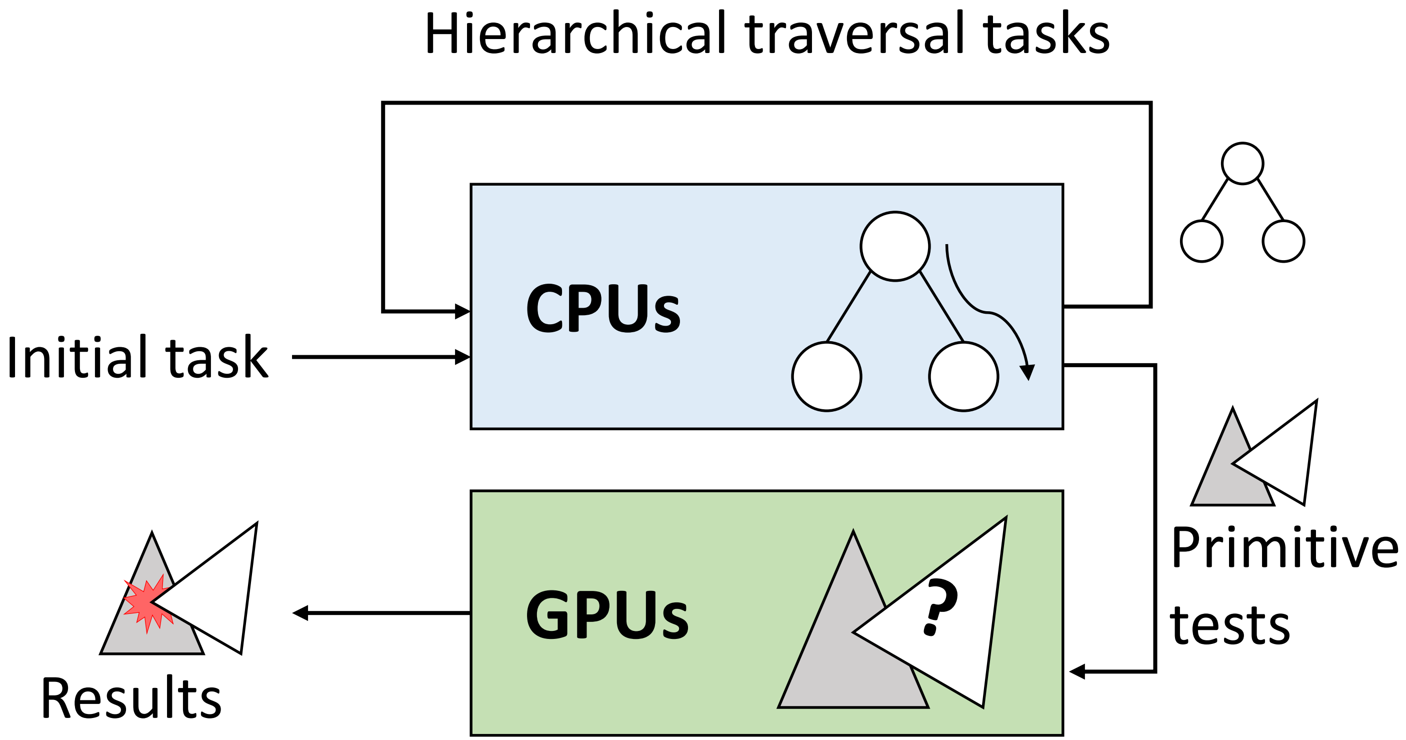
Computer Graphics Forum (Pacific Graphics), 2009

Received a **distinguished paper award** at the conference

Observation

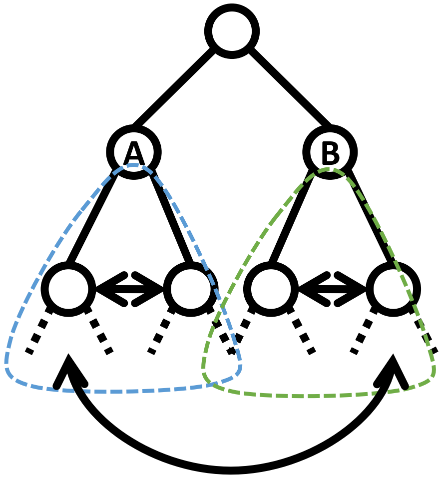


Workload Distribution



Reduce Communication Overhead

- **Identify disjoint tasks**
  - Remove synchronization in the main loop of the algorithm



Accessed nodes are disjoint

- **Optimize data communication between CPU and GPU**

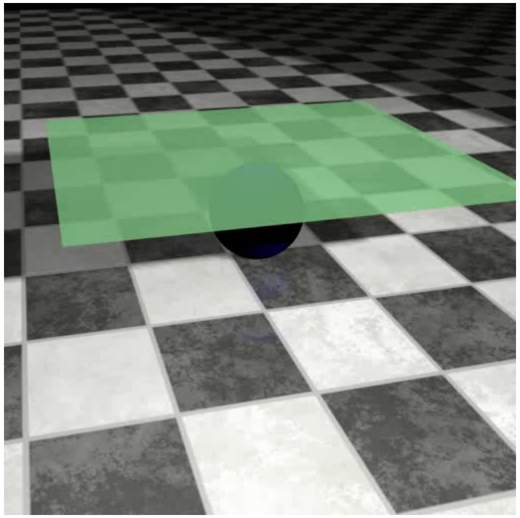
Please see the paper for the details

Results

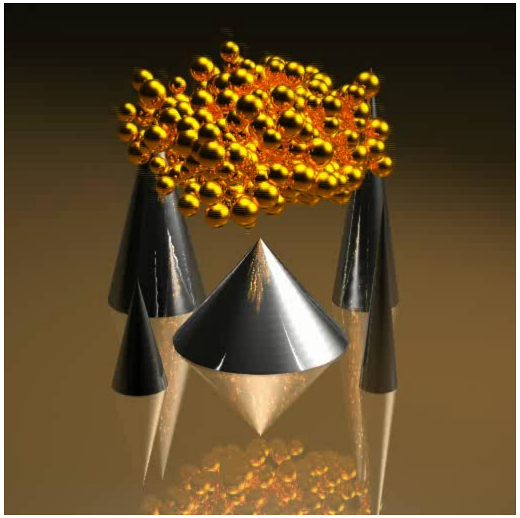
- **Testing Environment**
    - **One quad-core CPU** (Intel i7 CPU, 3.2 GHz)
    - **Two GPUs** (NVIDIA GeForce GTX285)
    - Run eight CPU threads by using Intel's hyper threading technology

- **Compare the performance over using a single CPU-core**

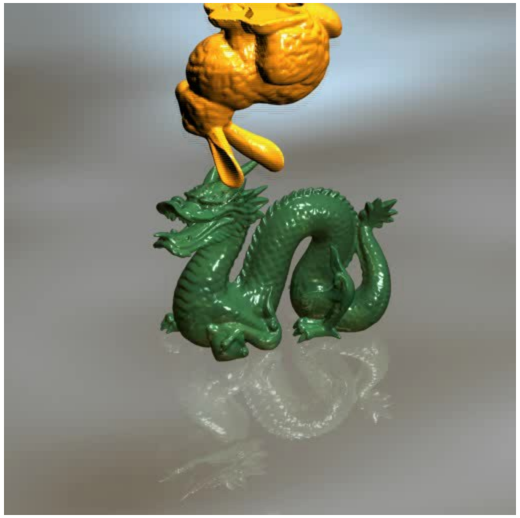
Results



- 94K triangles
- 10.4 X speed-up
- 23ms (43 FPS)

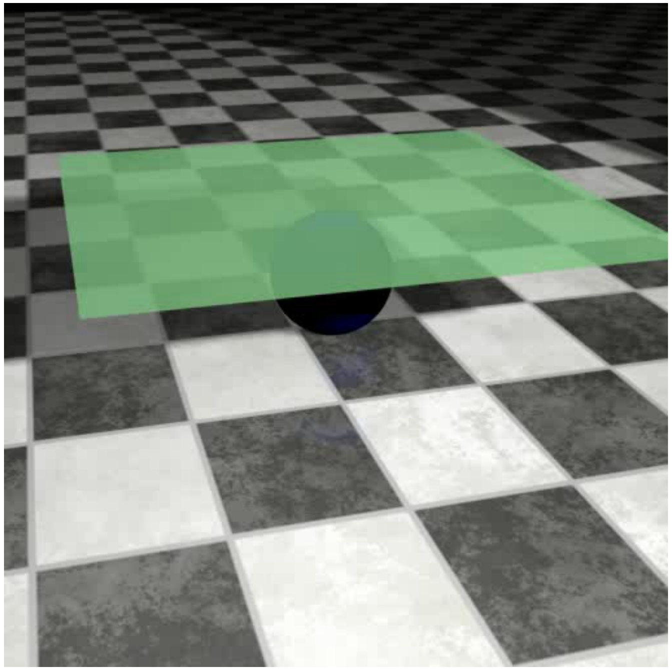


- 146K triangles
- 13.6 X speed-up
- 54ms (19 FPS)

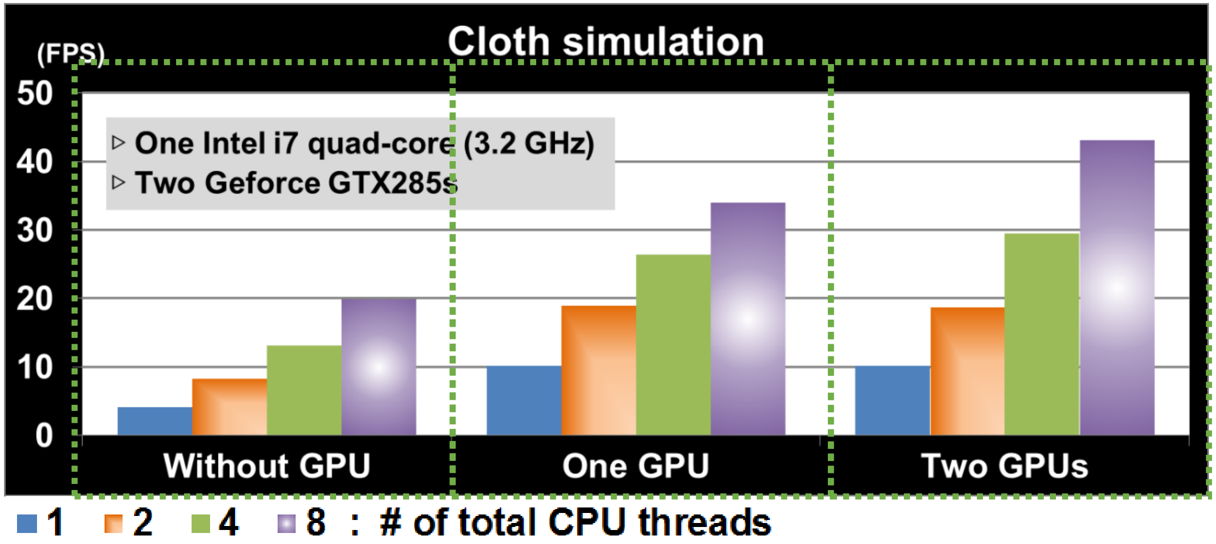


- 252K triangles
- 12.5 X speed-up
- 54ms (19 FPS)

Results



94K triangles



Scheduling in Heterogeneous Computing Environments for Proximity Queries

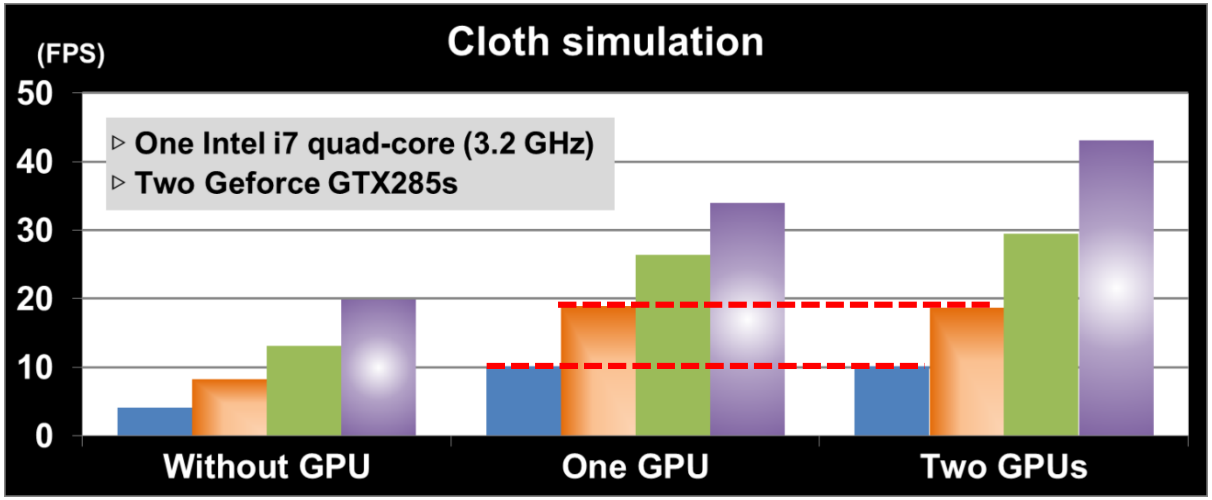
Duksu Kim, Jinkyu Lee, Junghwan Lee, Insik Shin, John Kim, Sung-Eui Yoon

IEEE Transactions on Visualization and Computer Graphics, Sept., 2013

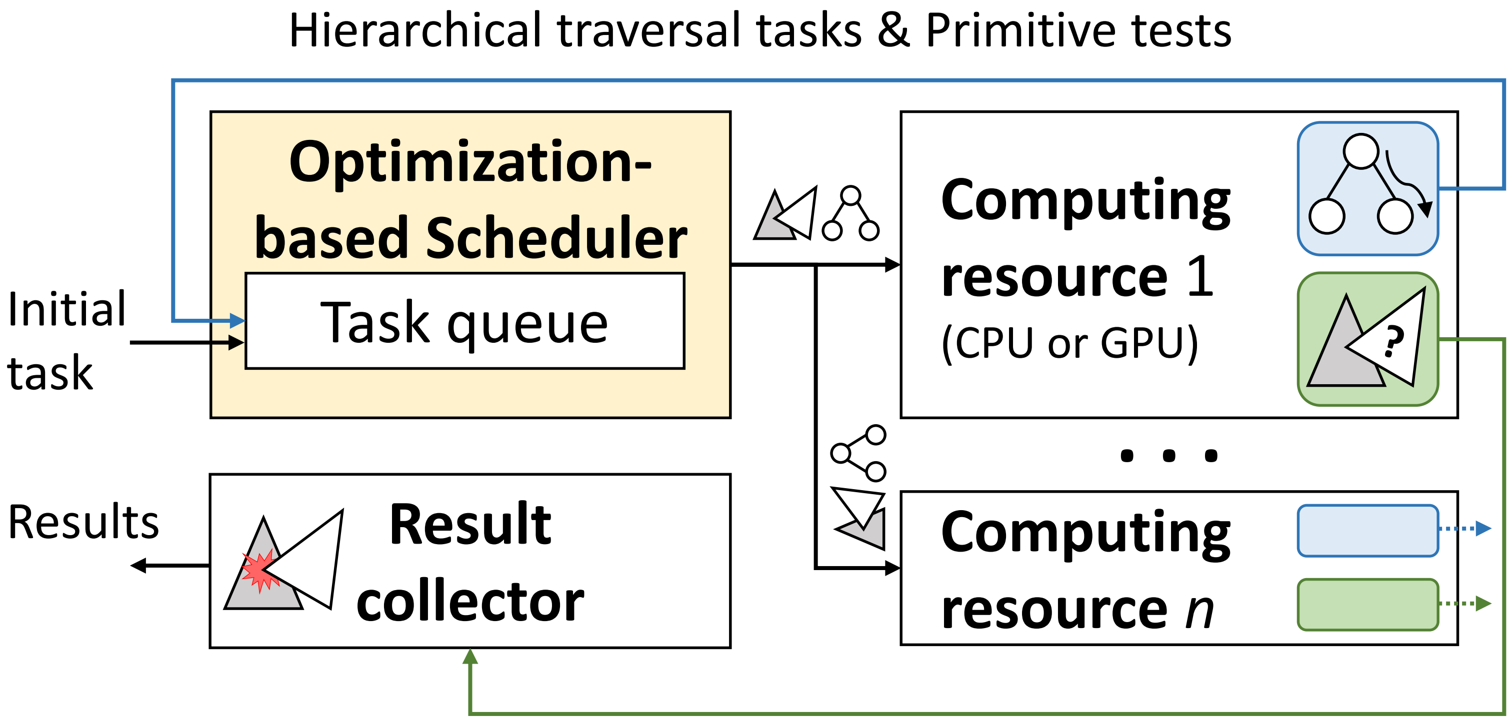
Selected as the **Spotlight Paper** for the issue

Observation

- HPCCD = Manual workload distribution
- No guarantee to efficient utilization of computing resource



Approach Overview



Optimization-based Scheduling

Expected running time model

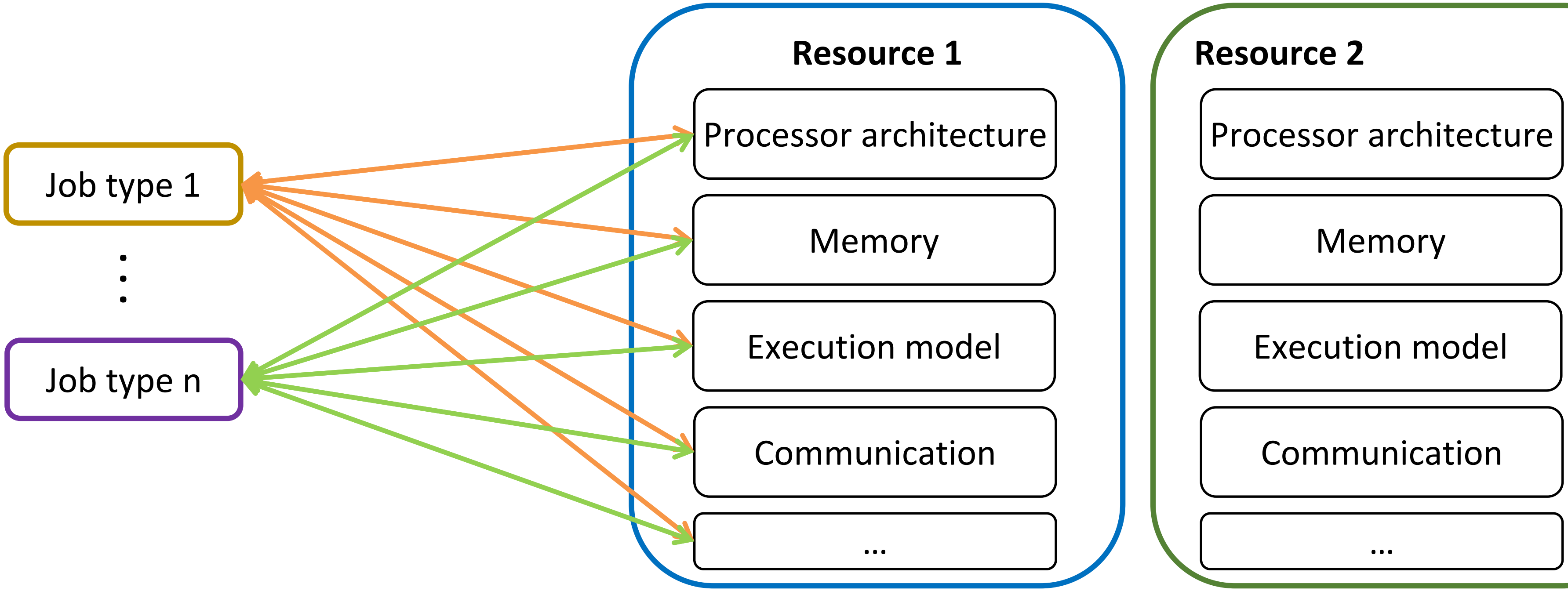
Optimization formulation

Iterative LP solver

- **Design an accurate performance model**
  - Predict how much computation time is required to finish jobs on a resource
  - Important to achieve the optimal scheduling result

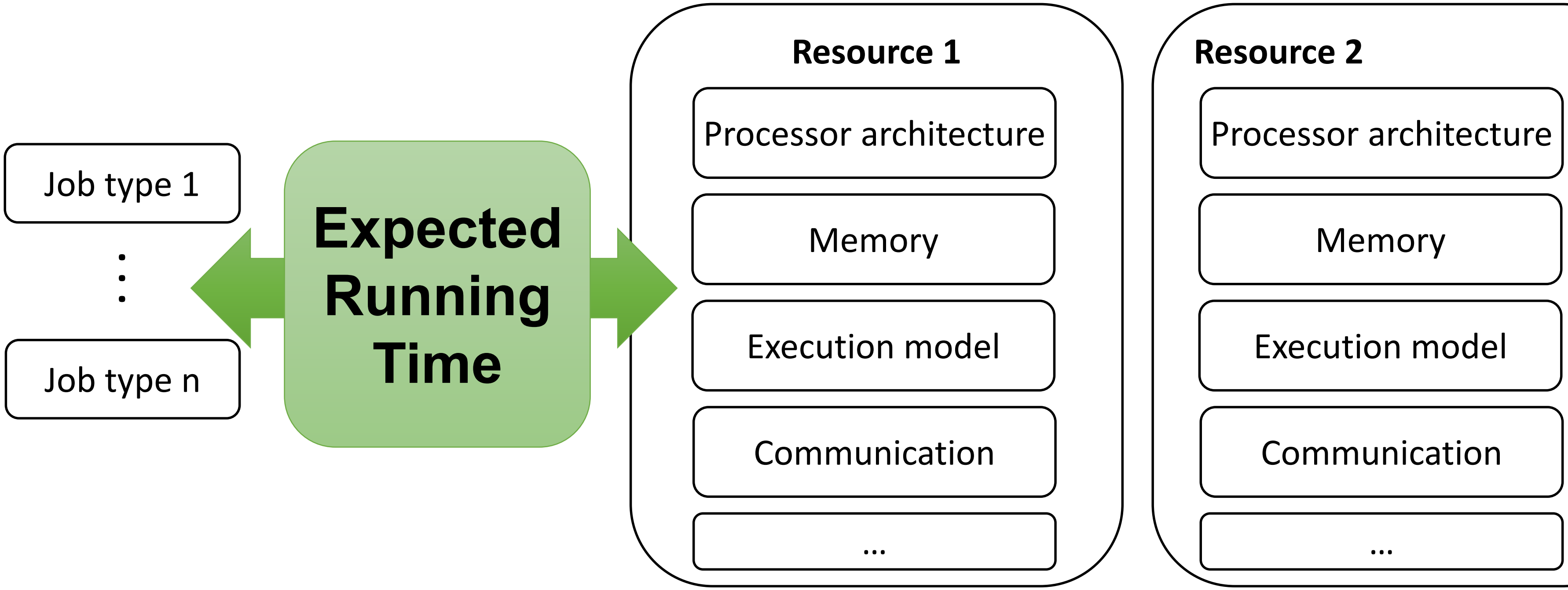
Performance Model

- **Performance relationship between jobs and resources is complex**



Performance Model

- **Abstract the complex relationship as an expected running time model**



Job type 1

⋮

Job type n

Expected Running Time

Resource 1

Processor architecture

Memory

Execution model

Communication

…

Resource 2

Processor architecture

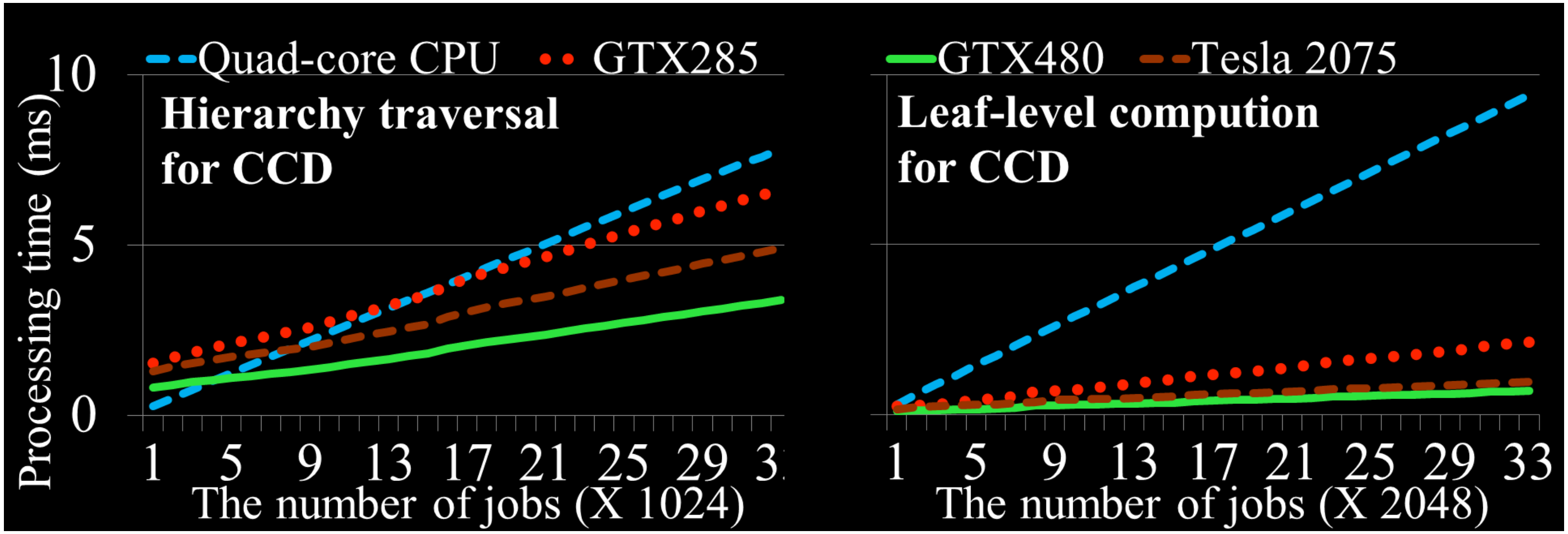
Memory

Execution model

Communication

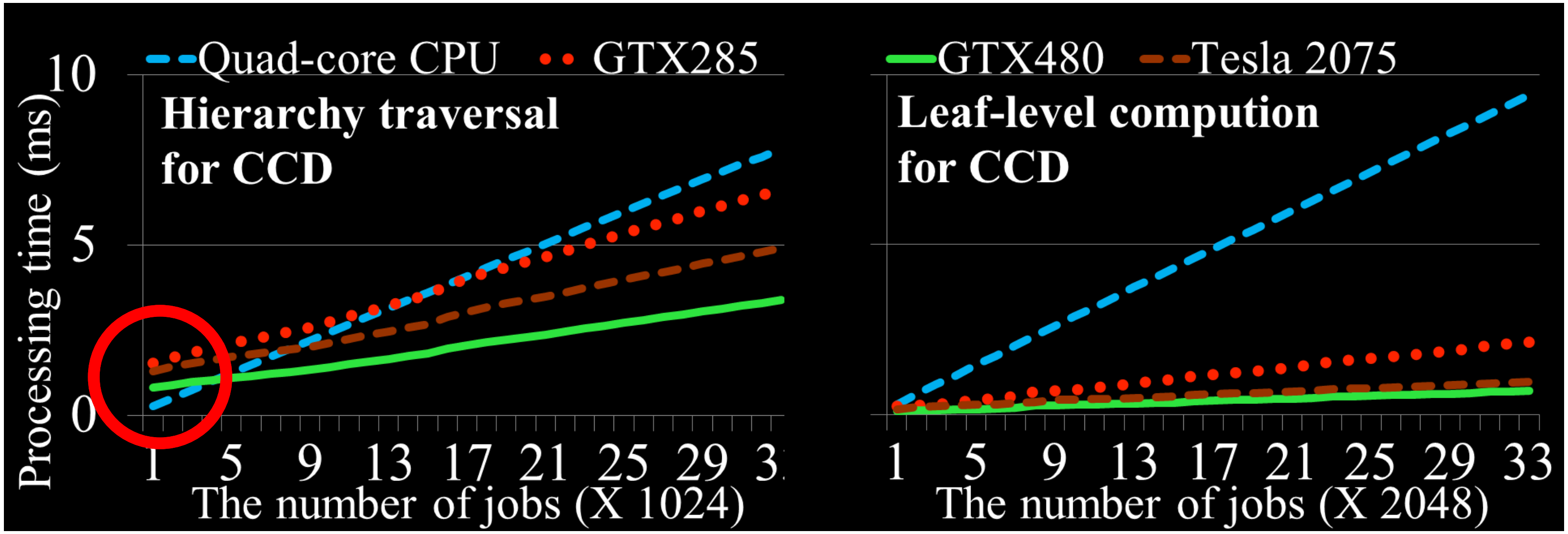
…

Performance Model



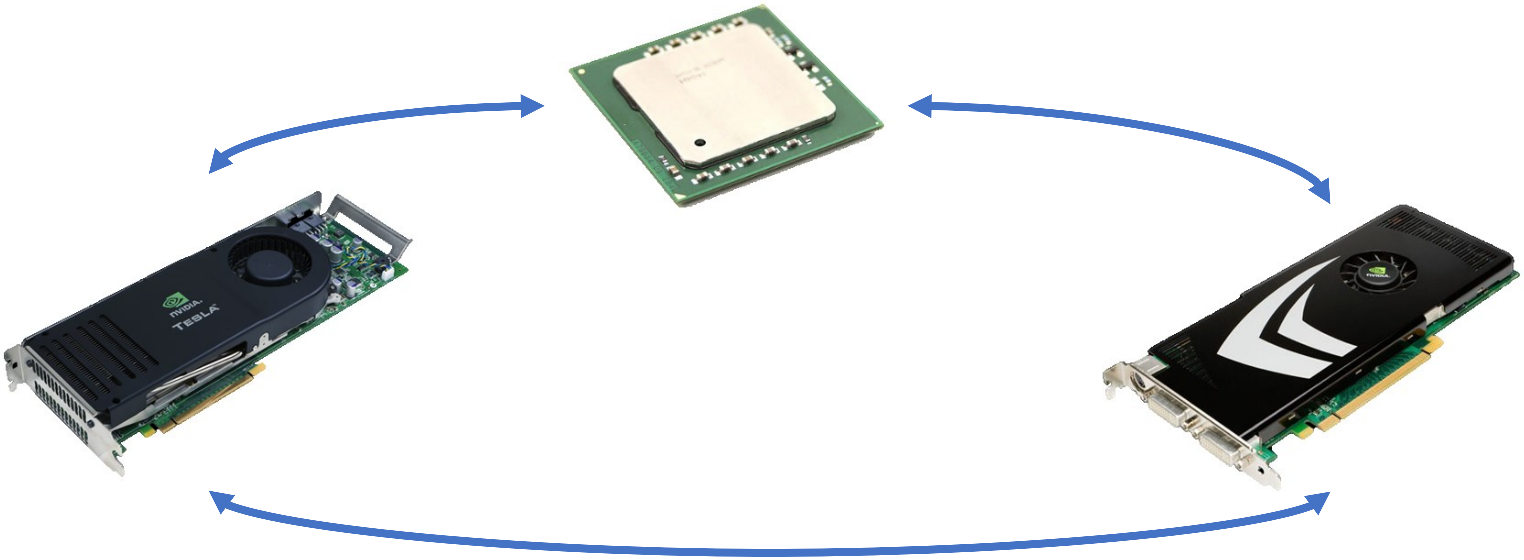
- **Running time is linearly increased as the number of jobs is increased**

Performance Model



- Running time is linearly increased as the number of jobs is increased
- Each computing resource requires a specific amount of setup cost

Performance Model



- Inter-device data transfer time depends on the pair of devices
- Data transfer time is linearly increased as the number of jobs is increased

Expected Running Time Model

- **T() : Expected running time on computing resource *i* for processing *n* jobs of job types *j* that are generated from computing resource *k***

$$T(k \rightarrow i, j, n_{ij}) = \begin{cases} 0, & \text{if } n_{ij} \text{ is } 0 \\ \underbrace{T_{setup}(i,j)}_{\text{Setup time}} + \underbrace{T_{proc}(i,j) \times n_{ij}}_{\text{Processing time}} \\ \quad + \underbrace{T_{trans}(k \rightarrow i, j) \times n_{ij}}_{\text{Data transfer time}}, & \text{otherwise.} \end{cases}$$

Optimization-based Scheduling

Expected running time model

Optimization formulation

Iterative LP solver

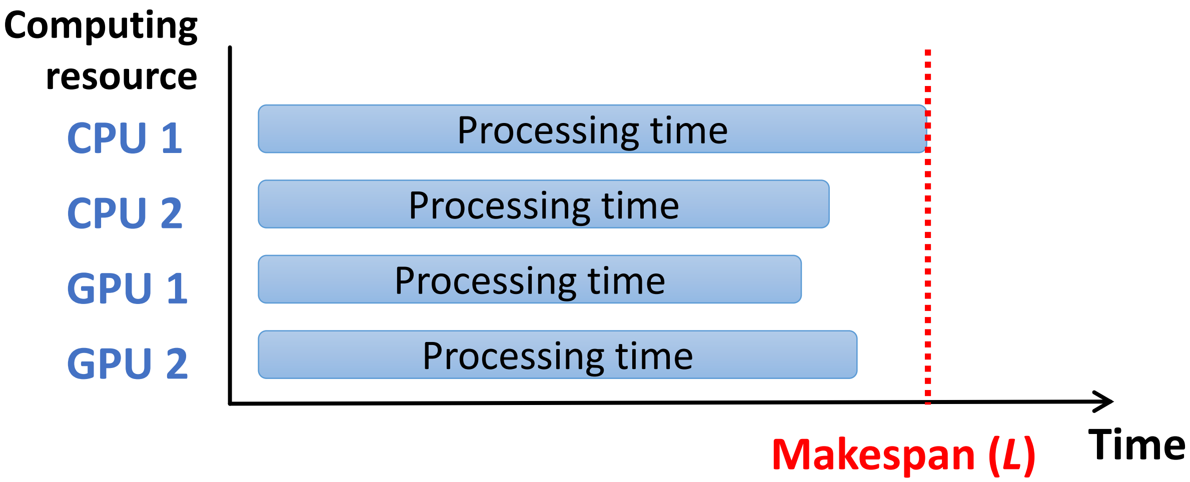
$$T(k \rightarrow i,j,n_{ij}) = \begin{cases} 0, & \text{if } n_{ij} \text{ is } 0 \\ T_{setup}(i,j) + T_{proc}(i,j) \times n_{ij} & \\ \quad +T_{trans}(k \rightarrow i,j) \times n_{ij}, & \text{otherwise.} \end{cases}$$

- **Formulate an optimization problem**
  - Based on the expected running time model
  - Need to represent the scheduling problem as a form of optimization problem

Optimization Formulation

- **Minimize the makespan (L) problem**

$$Minimize\ L,$$



Optimization Formulation

- **Calculate the optimal job distribution with the expected running time**

$$Minimize\ L,$$

$$subject\ to\ \boxed{T_{rest}(i)} + \boxed{\Sigma_{j=1}^{|J|} T(i,j,n_{ij})} \leq L, \forall i \in R \quad ①$$

Expected processing time

Resource i

Rest time for completing already assigned jobs

Processing time for jobs will be assigned

① The expected processing time of computing resources is equal or smaller than the makespan

Optimization Formulation

- **Calculate the optimal job distribution with the expected running time**

$$Minimize\ L,$$

$$subject\ to\ T_{rest}(i) + \Sigma_{j=1}^{|J|} T(i, j, n_{ij}) \leq L, \forall i \in R \quad ①$$

$$\Sigma_{i=1}^{|R|} n_{ij} = n_j, \forall j \in J \quad ②$$

① The expected processing time of computing resources is equal or smaller than the makespan

② There are no missing or duplicated jobs

Optimization Formulation

- **Calculate the optimal job distribution with the expected running time**

$$Minimize\ L,$$

Job distribution

$$subject\ to\ T_{rest}(i) + \Sigma_{j=1}^{|J|} T(i, j, n_{ij}) \leq L, \forall i \in R \quad \text{①}$$

$$\Sigma_{i=1}^{|R|} n_{ij} = n_j, \forall j \in J \quad \text{②}$$

$$n_{ij} \in \mathbb{Z}^{+} (zero\ or\ positive\ integers). \quad \text{③}$$

① The expected processing time of computing resources is equal or smaller than the makespan

② There are no missing or duplicated jobs

③ Each job is atomic

Optimization-based Scheduling

Expected running time model → **Optimization formulation** → **Iterative LP solver**

NP-hard Problem!

$$\begin{aligned}
&Minimize\ L,\\
&subject\ to\ T_{rest}(i) + \Sigma_{j=1}^{|J|} T(i, j, n_{ij}) \leq L, \forall i \in R\\
&\qquad \Sigma_{i=1}^{|R|} n_{ij} = n_j, \forall j \in J\\
&\qquad n_{ij} \in \mathbb{Z}^{+} (zero\ or\ positive\ integers).
\end{aligned}$$

- **High computational cost**
  - Jobs are dynamically generated at runtime
  - Optimization process takes long time for interactive or real-time applications

Optimization-based Scheduling

Expected running time model → **Optimization formulation** → **Iterative LP solver**

$$T(k \rightarrow i, j, n_{ij}) = \begin{cases} 0, & \text{if } n_{ij} \text{ is } 0 \\ T_{setup}(i,j) + T_{proc}(i,j) \times n_{ij} & \\ \quad + T_{trans}(k \rightarrow i, j) \times n_{ij}, & \text{otherwise.} \end{cases}$$

Designed an iterative LP solving algorithm to handle the piece-wise condition

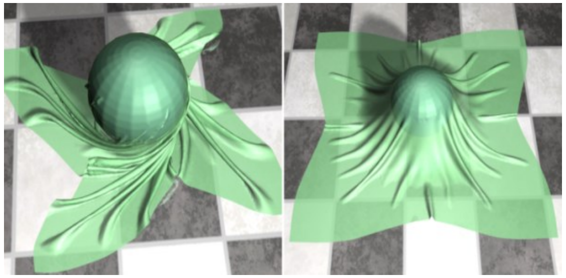
$$\begin{aligned} & Minimize\ L, \\ & subject\ to\ T_{rest}(i) + \Sigma_{j=1}^{|J|} T(i, j, n_{ij}) \leq L, \forall i \in R \\ & \qquad \Sigma_{i=1}^{|R|} n_{ij} = n_j, \forall j \in J \\ & \qquad n_{ij} \in \mathbb{Z}^{+}(zero\ or\ \cancel{positive\ integers}). \end{aligned}$$

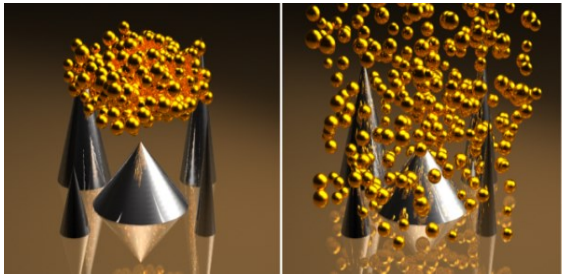
↓ Positive floating-point numbers

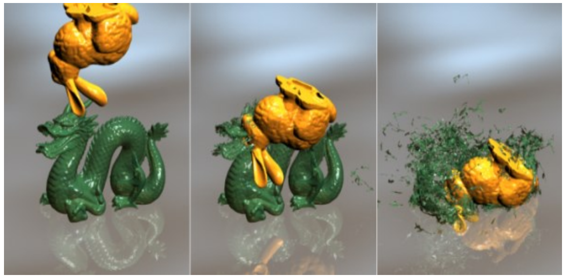
Please see the paper for the details

Results

- **Tested with various applications**
  - Simulations (Continuous collision detection)
  - Motion planning (Discrete collision detection)
  - Global illumination (Ray-Triangle intersection)

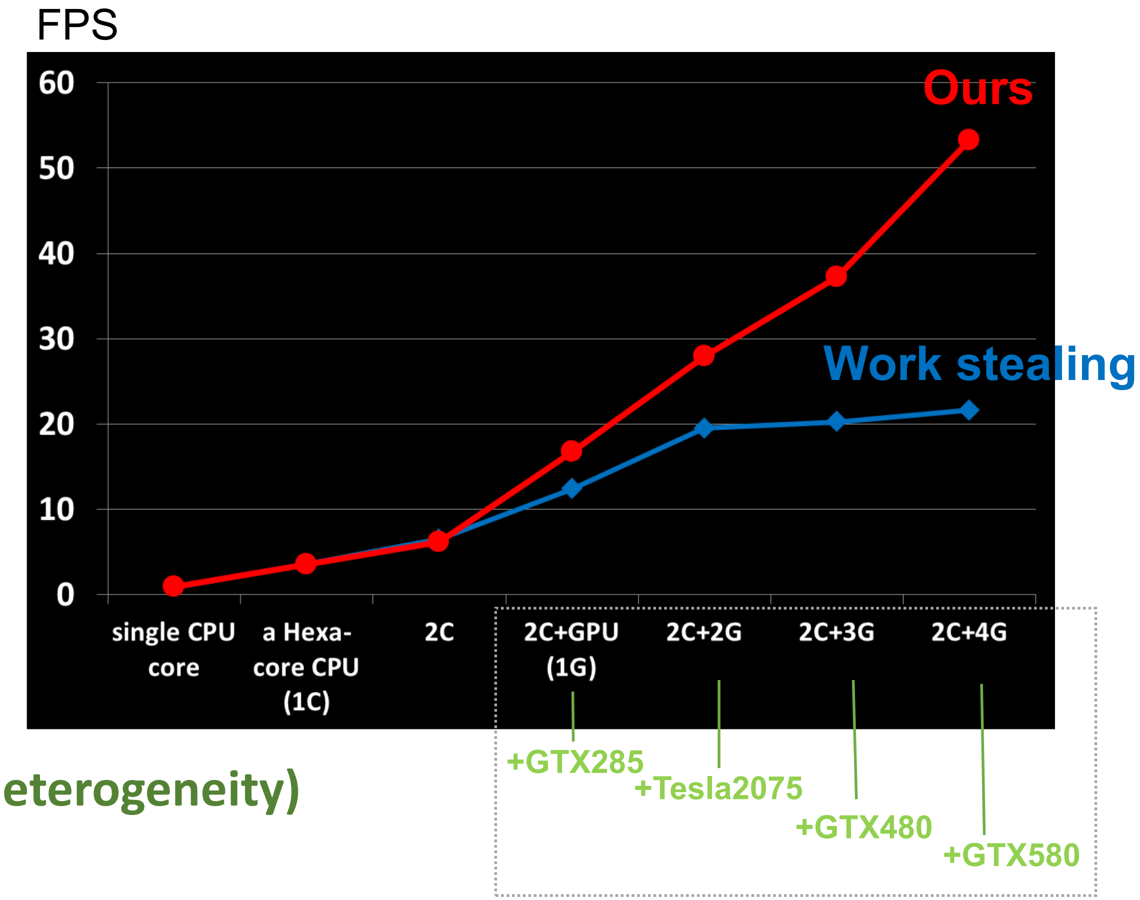






Results

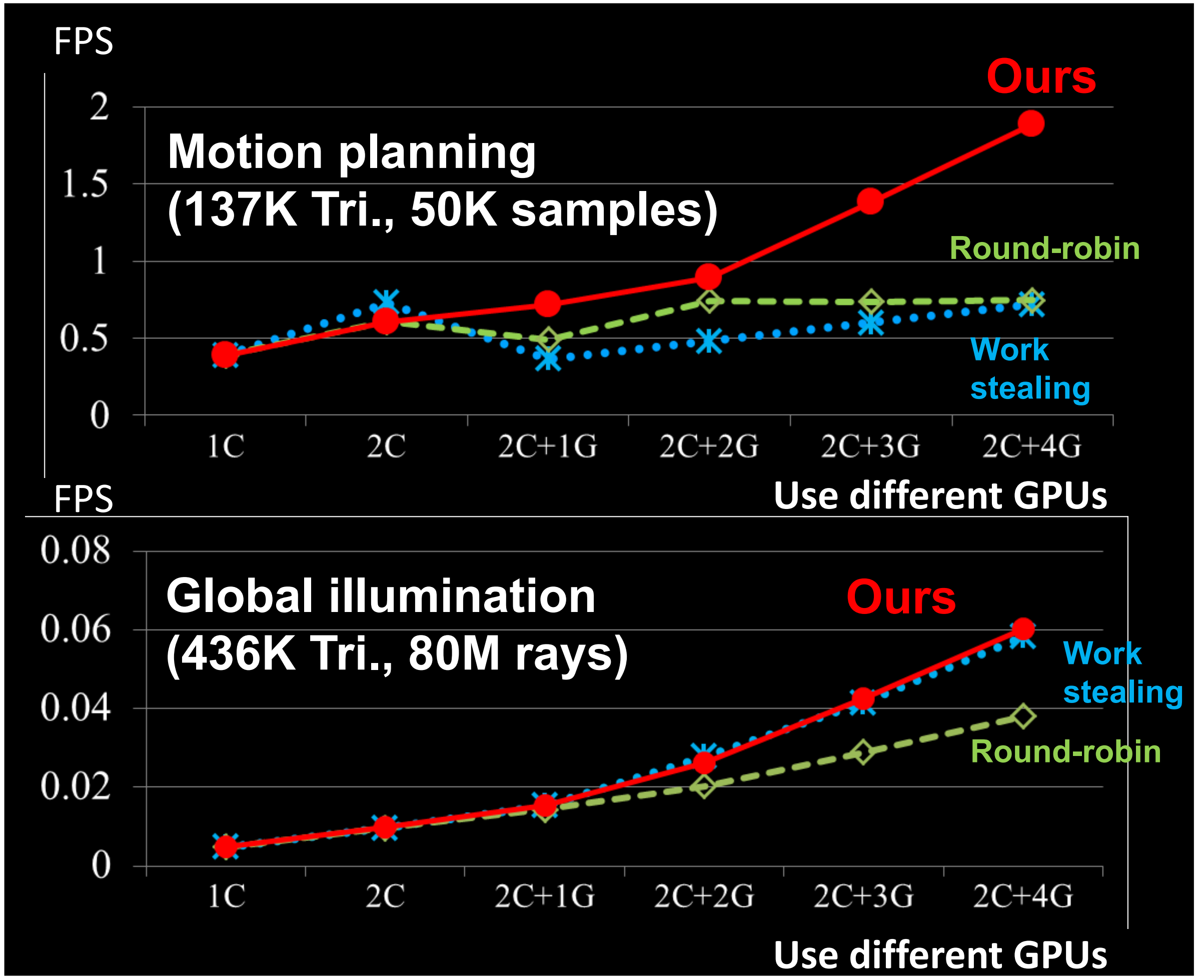
Fracturing simulation (252K Tri.)



Use different GPUs (high heterogeneity)

- For conservative comparison, we did manual tuning to get the best performance for tested methods except for ours

Results



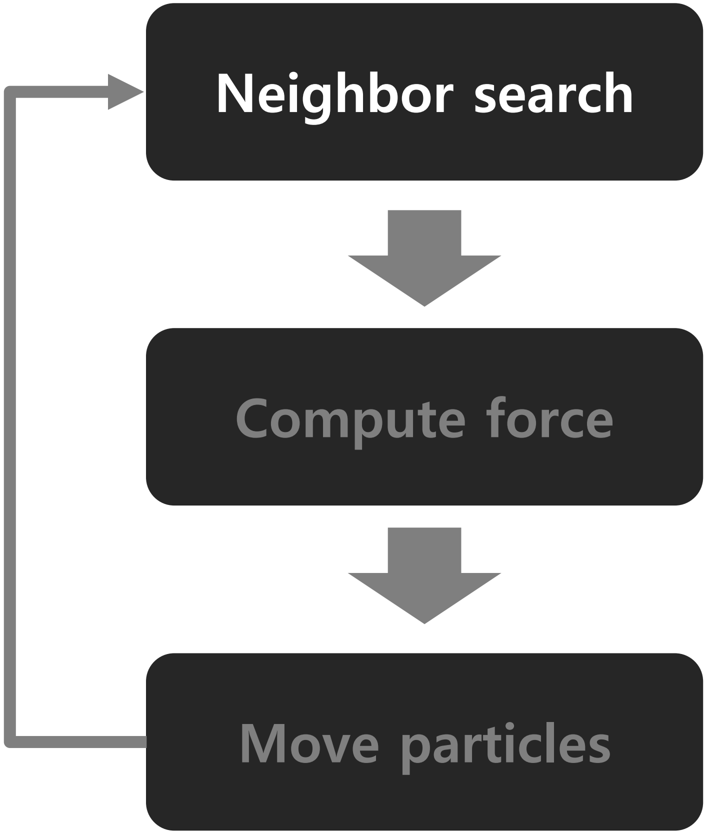
- For conservative comparison, we did manual tuning to get the best performance for tested methods except for ours

Out-of-Core Proximity Computation for Particle-based Fluid Simulations

Duksu Kim, Myung-Bae Son, Young J. Kim, Jeong-Mo Hong, Sung-Eui Yoon

High Performance Graphics, 2014

Particle-based Fluid Simulation



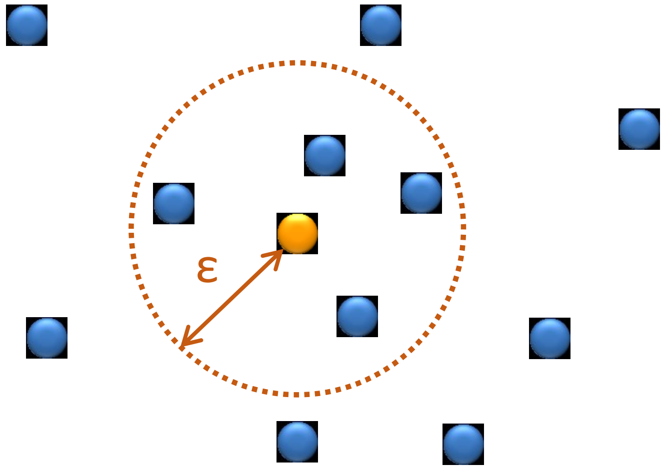
Neighbor search

Compute force

Move particles

Performance bottleneck

- Takes 60~80% of simulation computation time

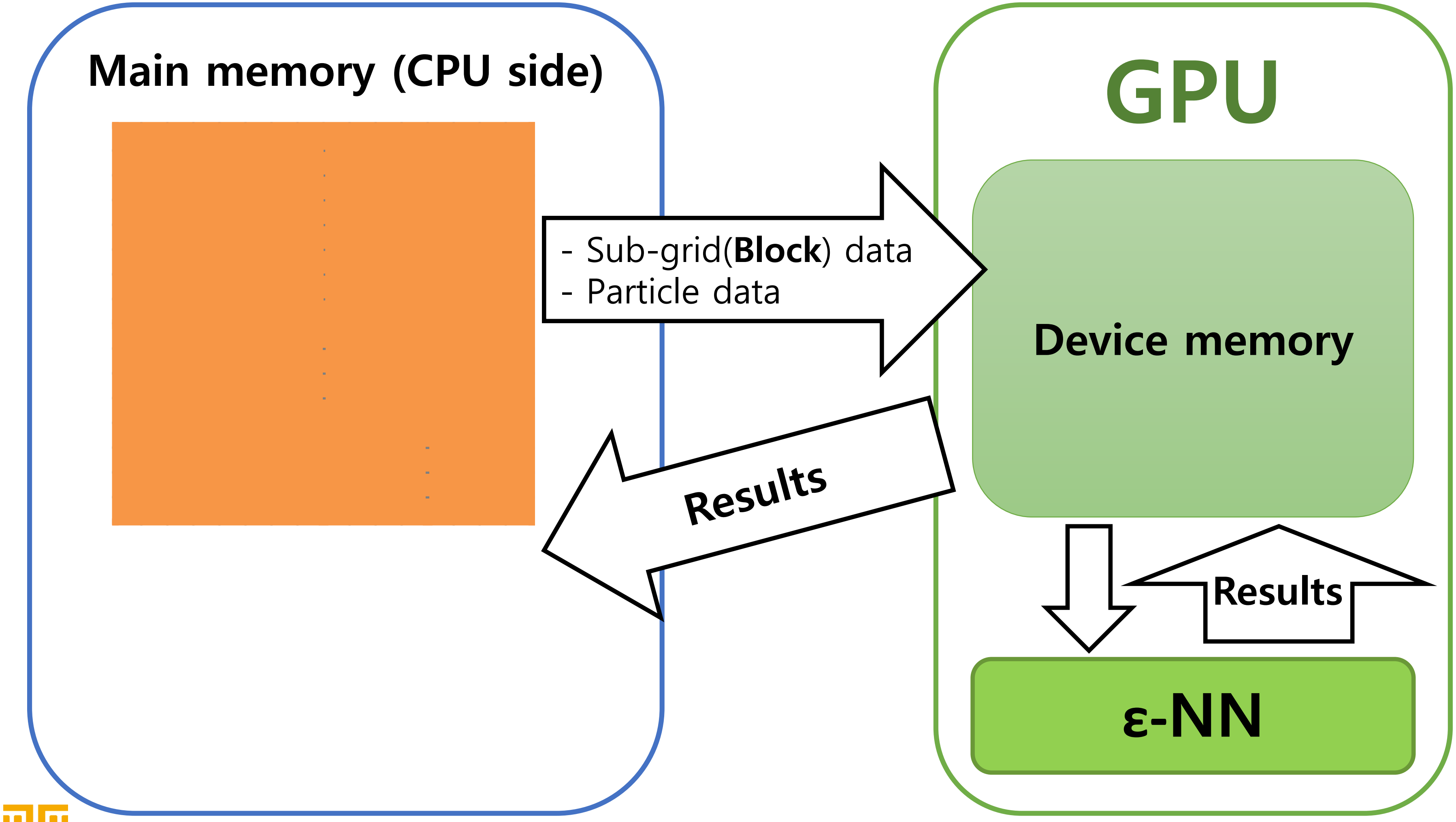


ε-Nearest Neighbor (ε-NN)

Observation

- **GPU shows much higher performance than CPU**
- **But, for a large scale simulation,**
  - **The device memory on a GPU is not enough** to load whole grid data and store lists of neighbors for all particles
- **CPU has relatively large memory space**
  - More than hundreds of GBs

Out-of-core Algorithm



Main memory (CPU side)

- Sub-grid(**Block**) data
- Particle data

Results

GPU

Device memory

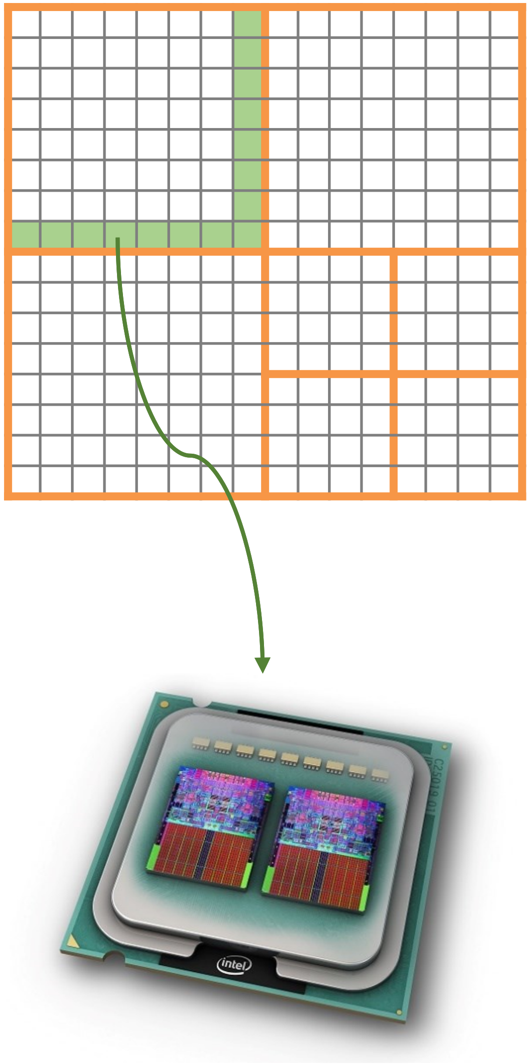
Results

ε-NN

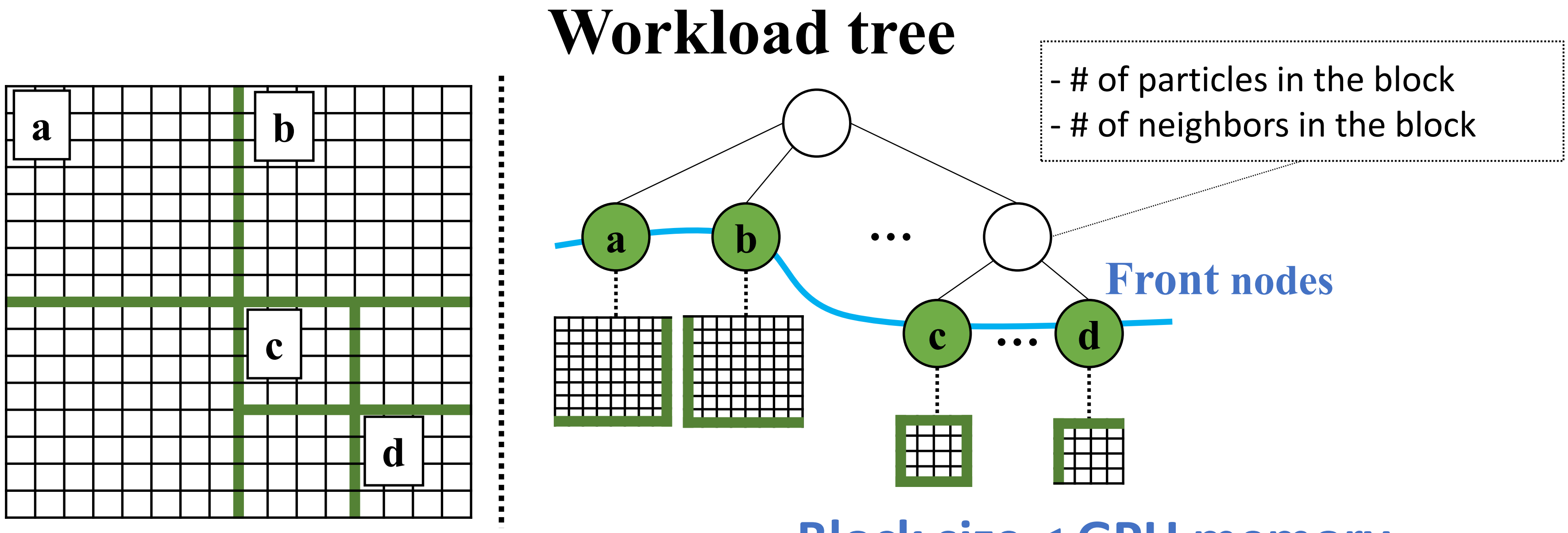
Boundary Region

- **Required data in adjacent blocks**
- **Inefficient to handle in out-of-core manner**

- **Multi-core CPUs handles the boundary region**
  - CPU (main) memory contain all required data
  - Ratio of boundary region is usually much smaller than inner region



Hierarchical Work Distribution



Workload tree

- # of particles in the block
- # of neighbors in the block

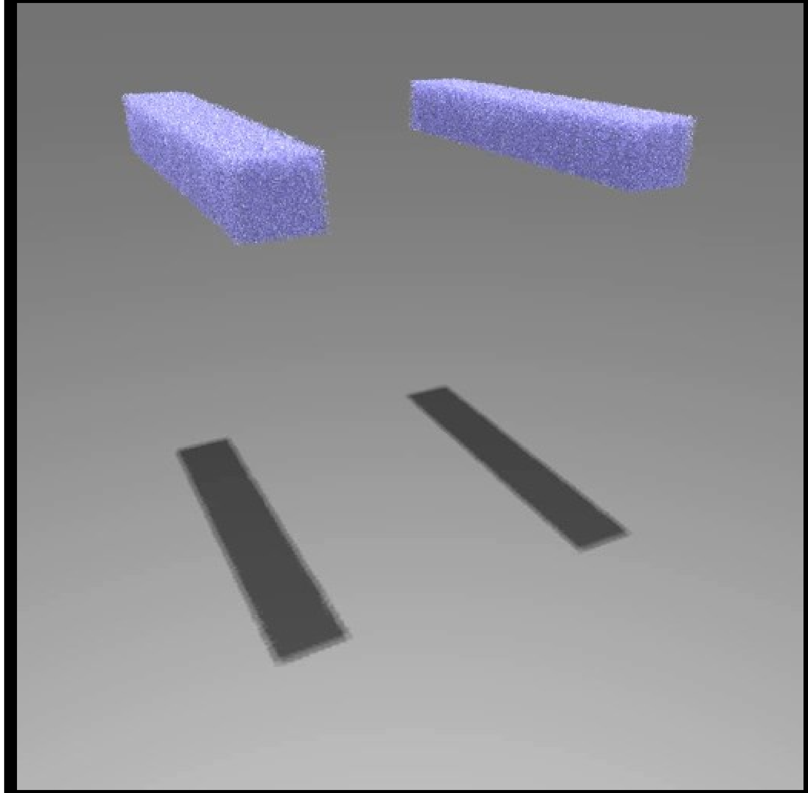
Front nodes

Block size < GPU memory

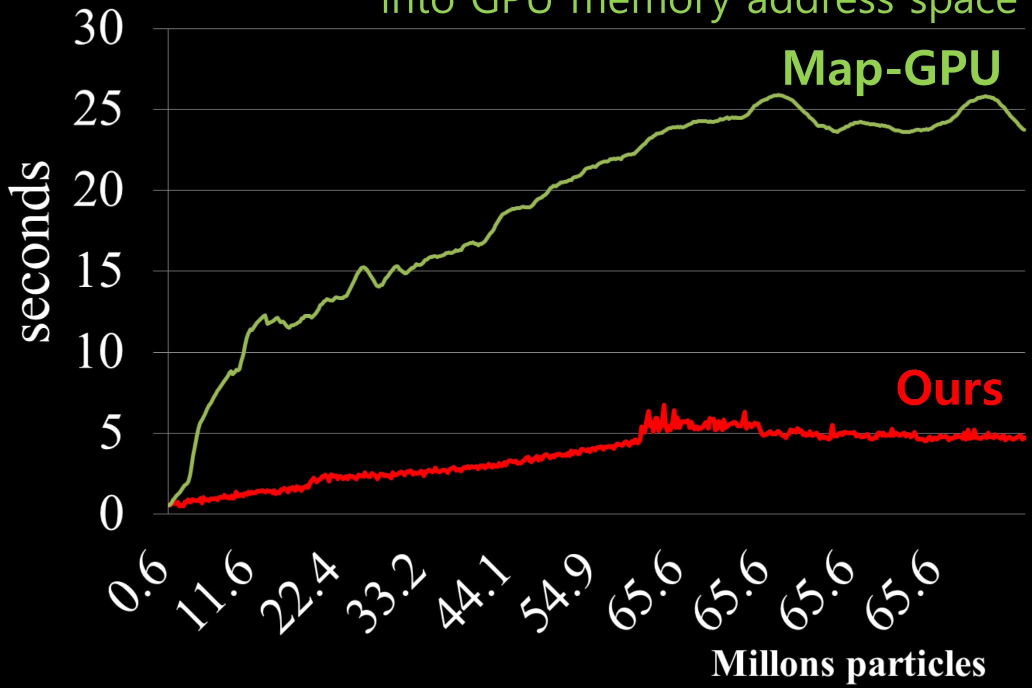
How to determine the block size?

Please see the paper for the details

Results

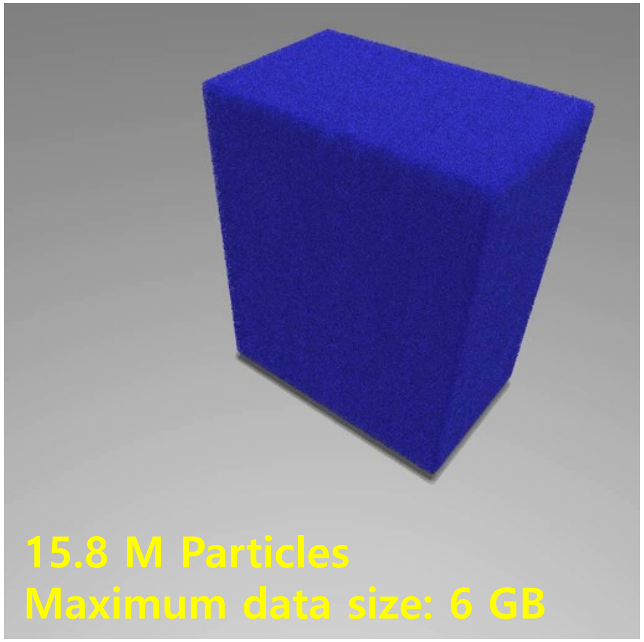


NVIDIA mapped memory Tech
- Map CPU memory space into GPU memory address space

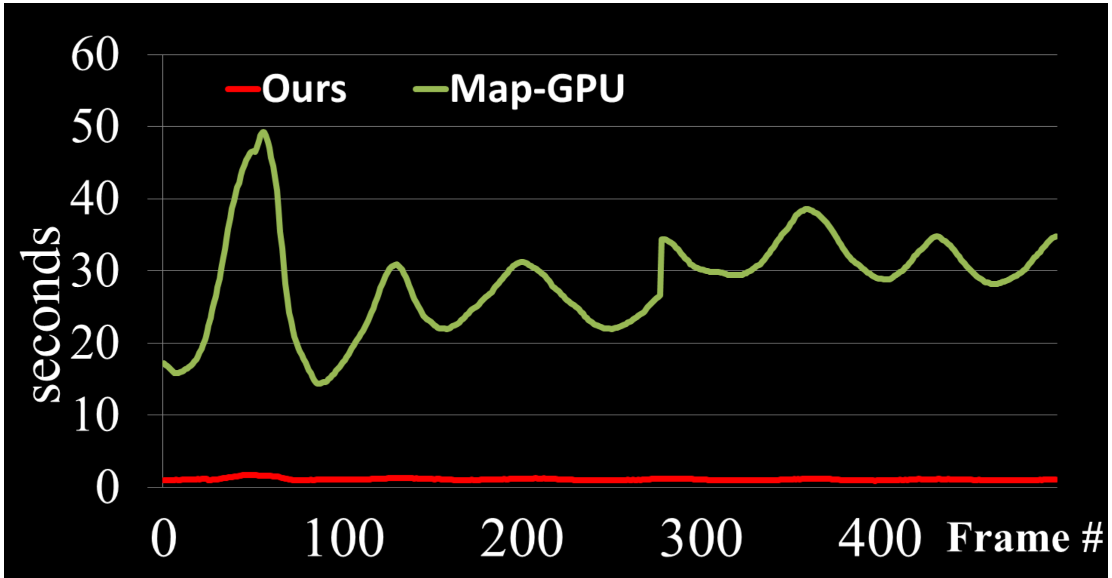


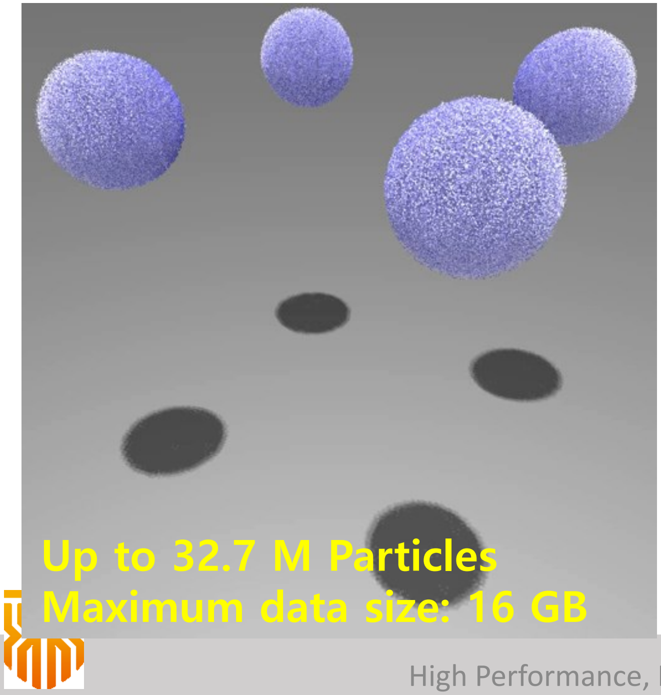
Up to 65.6 M Particles
Maximum data size: 13 GB

OOCNNS

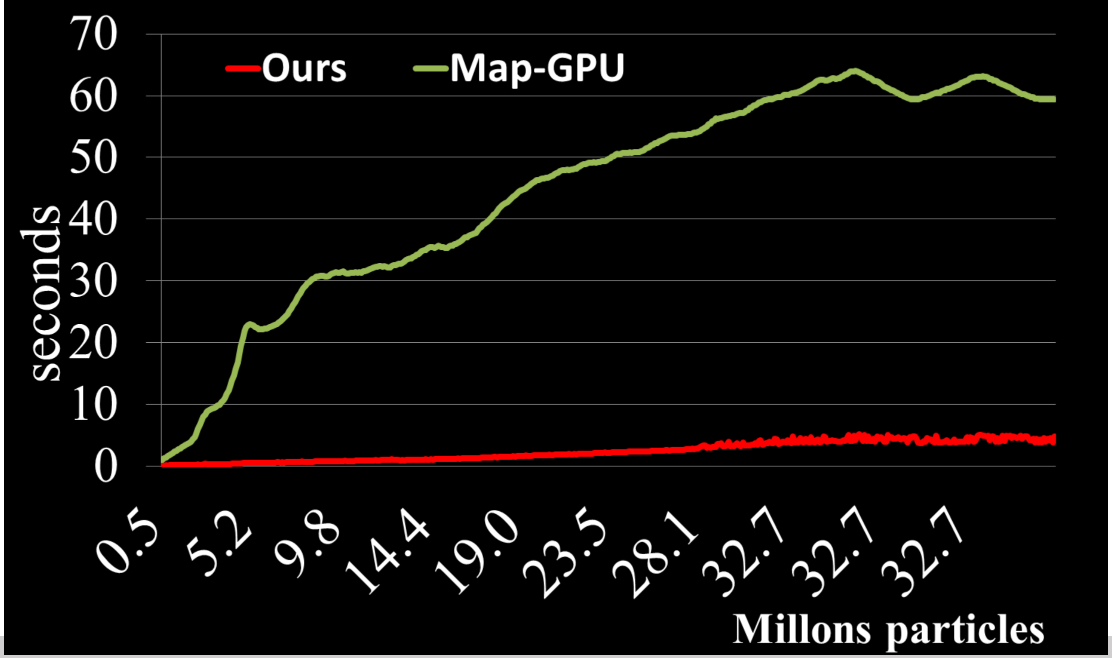


15.8 M Particles
Maximum data size: 6 GB

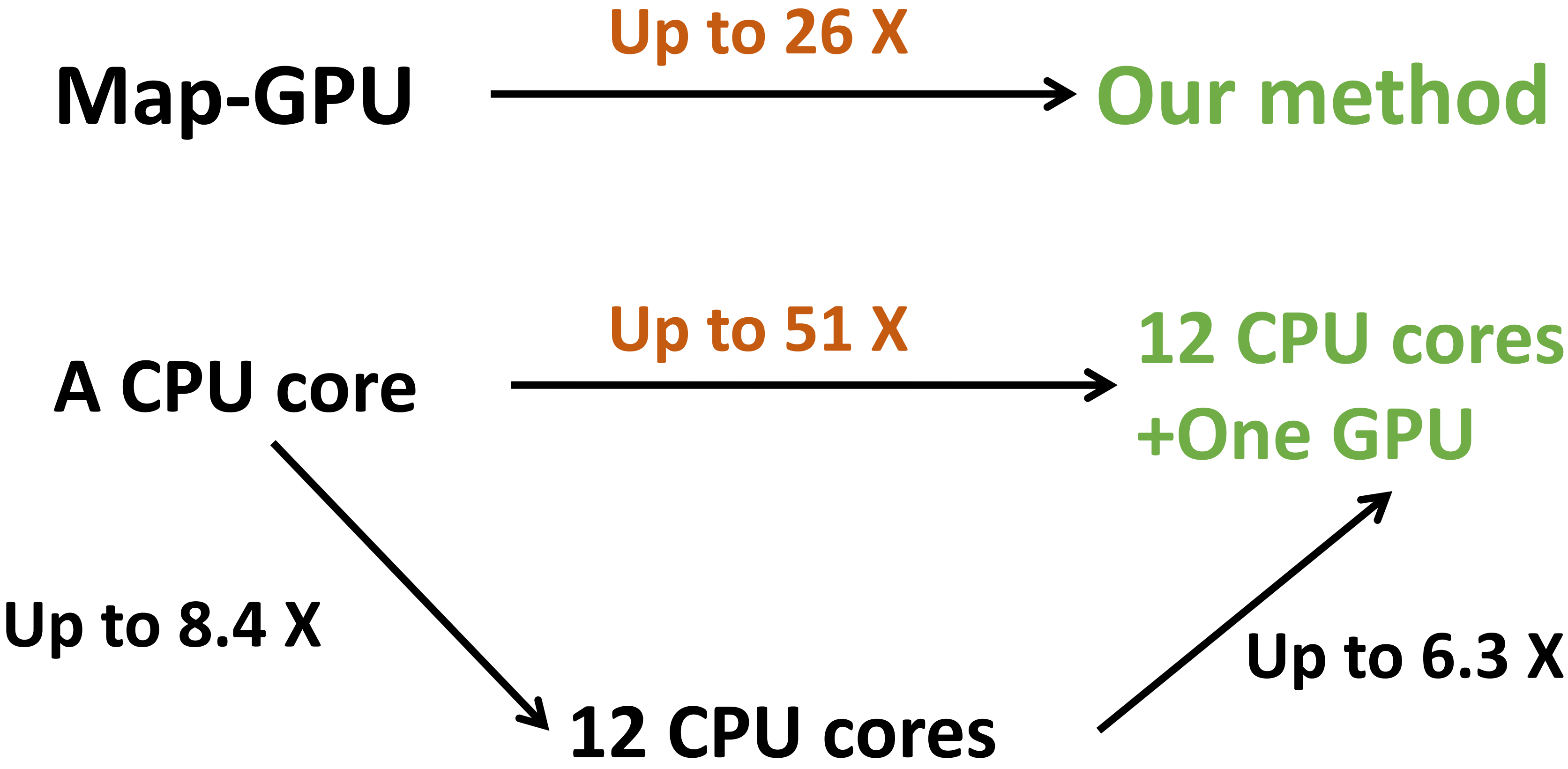




Up to 32.7 M Particles
Maximum data size: 16 GB



Results



Up to 26 X

Map-GPU → Our method

Up to 51 X

A CPU core → 12 CPU cores +One GPU

Up to 8.4 X

A CPU core → 12 CPU cores

Up to 6.3 X

12 CPU cores → 12 CPU cores +One GPU

Summary

- **We have learned how prior work improve the performance of the proximity computation with heterogeneous parallel algorithms**

- **Hints for designing a heterogeneous parallel Algo.**
  - Understand characteristics of tasks and resources
    - Computational and Spatial perspectives
  - Generally,
    - Hierarchical work maps to CPU-like architectures
    - Compute-intensive work maps to GPU-like architectures
  - To achieve an optimal performance,
    - Formulate performance model
    - Design a dynamic work distribution algorithm

Any Questions?

- bluekdct@gmail.com
- http://hpc.koreatech.ac.kr