Solving NP-Complete Problems using CUDA

CS538 Project

Haoran Wang

Dec 9, 2021

Overview

Motivation:

- We know that NP-Complete problems are very important. If we can solve one NP-Complete problem in polynomial time, then all problems in NP can be solved in polynomial time.

Overview

Motivation:

- We know that NP-Complete problems are very important. If we can solve one NP-Complete problem in polynomial time, then all problems in NP can be solved in polynomial time.
- Although we can't solve NP-Complete problems in polynomial time, can the solutions benefit from modern hardware, like GPU?

Overview

Motivation:

- We know that NP-Complete problems are very important. If we can solve one NP-Complete problem in polynomial time, then all problems in NP can be solved in polynomial time.
- Although we can't solve NP-Complete problems in polynomial time, can the solutions benefit from modern hardware, like GPU?
- How much performance boost can we get by using more powerful hardware?

Overview

- What is CUDA?
- What is GPU?
- Why is CUDA important?

Overview

- What is CUDA?
- What is GPU?
- Why is CUDA important?
- Using CUDA to solve SAT and Knapsack through Genetic Algorithm (GA) (2011)
- Using CUDA to brute force solving TSP, party, and knapsack (2014)
- Using CUDA to solve Vertex Cover through Fixed-Parameter Tractable Algorithm (FPT) (2015)

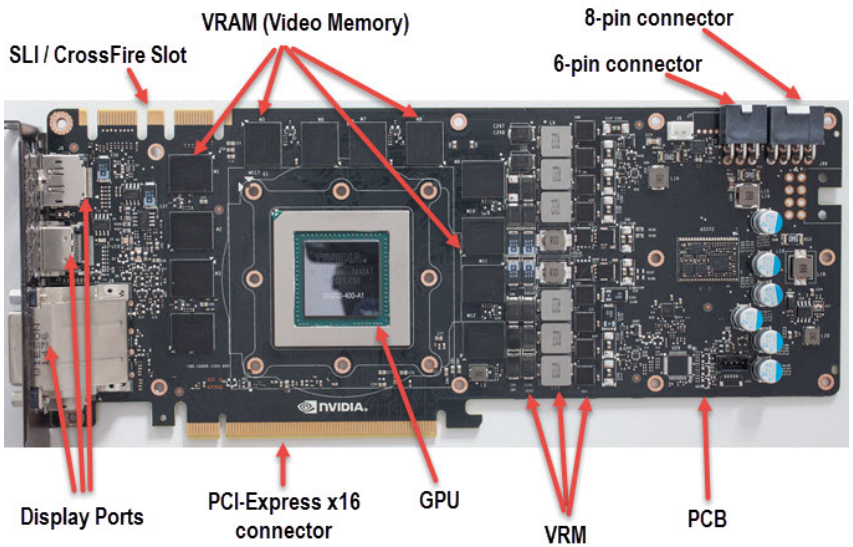
- CUDA (Compute Unified Device Architecture) is a parallel computing platform and API that allows software to use GPU (Graphic Processing Unit) for general purpose processing.
- CUDA was created by Nvidia in 2007.
- CUDA is designed to work with programming languages like C, C++ and Fortran.

- CUDA (Compute Unified Device Architecture) is a parallel computing platform and API that allows software to use GPU (Graphic Processing Unit) for general purpose processing.
- CUDA was created by Nvidia in 2007.
- CUDA is designed to work with programming languages like C, C++ and Fortran.
- While there have been other proposed APIs for GPUs, such as OpenCL, and other GPUs from other competitive companies, such as, AMD, the Combination of CUDA and NVIDIA GPUs dominates several application areas, such as, deep learning.

Introduction

GPU Overview

- GPU (Graphics Processing Unit) is a specialized electronic circuit designed for graphics and video rendering.
- Graphics cards are arguably as old as the PC. The first graphics card was introduced in 1999 (GeForce 256) by Nvidia. At the time, the principle reason for having a GPU was for gaming, it wasn't until later that people used GPUs for science and engineering.



CPU

- Instruction-level parallelism (ILP)
  - few "brawny" cores
  - general-purpose computing
- Reduce latency
  - large, complex memory hierarchy
  - hardware prefetching

GPU

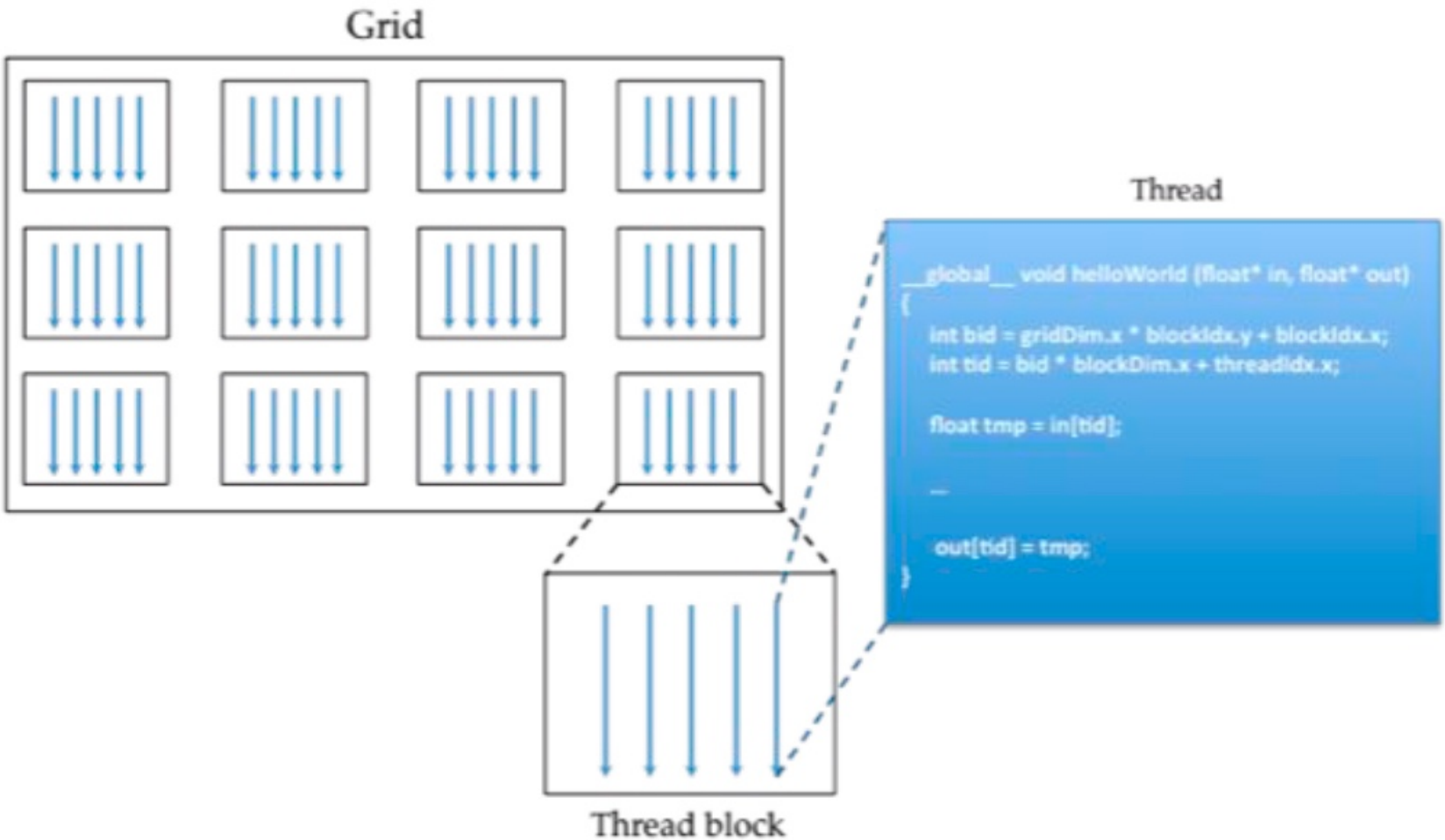
- Data-level parallelism (DLP)
  - many "wimpy" cores
  - high peak performance
- Increase throughput
  - large number of hardware threads
  - user-managed "scratchpad" cache

Introduction

GPU Overview

	Skylake Platinum 8180	Volta V100	Difference
# Cores/SM	28	80	2.86x
Clock (max)	3.8 GHz	1.455 GHz	0.4x
SIMD width	512 Bits	N/A	
CUDA cores	N/A	5120	
Performance (single-precision)	6.8 TFLOPS*	14.9 TFLOPS	2.2x
Performance (double-precision)	3.4 TFLOPS*	7.45 TFLOPS	2.2x
Bandwidth	128 GB/s	900 GB/s	7x
TDP	205 Watts	300 Watts	1.46x

- GPU has dramatically accelerate workloads in HPC (High Performance Computing) and Deep Learning.
- Popular Deep Learning frameworks that uses CUDA including PyTorch and TensorFlow. Specifically, they use cuDNN library for the deep neural network computations.


Grid
Thread
__global__ void helloWorld (float* in, float* out)
{
int bid = gridDim.x * blockIdx.y + blockIdx.x;
int tid = bid * blockDim.x + threadIdx.x;
float tmp = in[tid];
out[tid] = tmp;
}
Thread block

Naïve

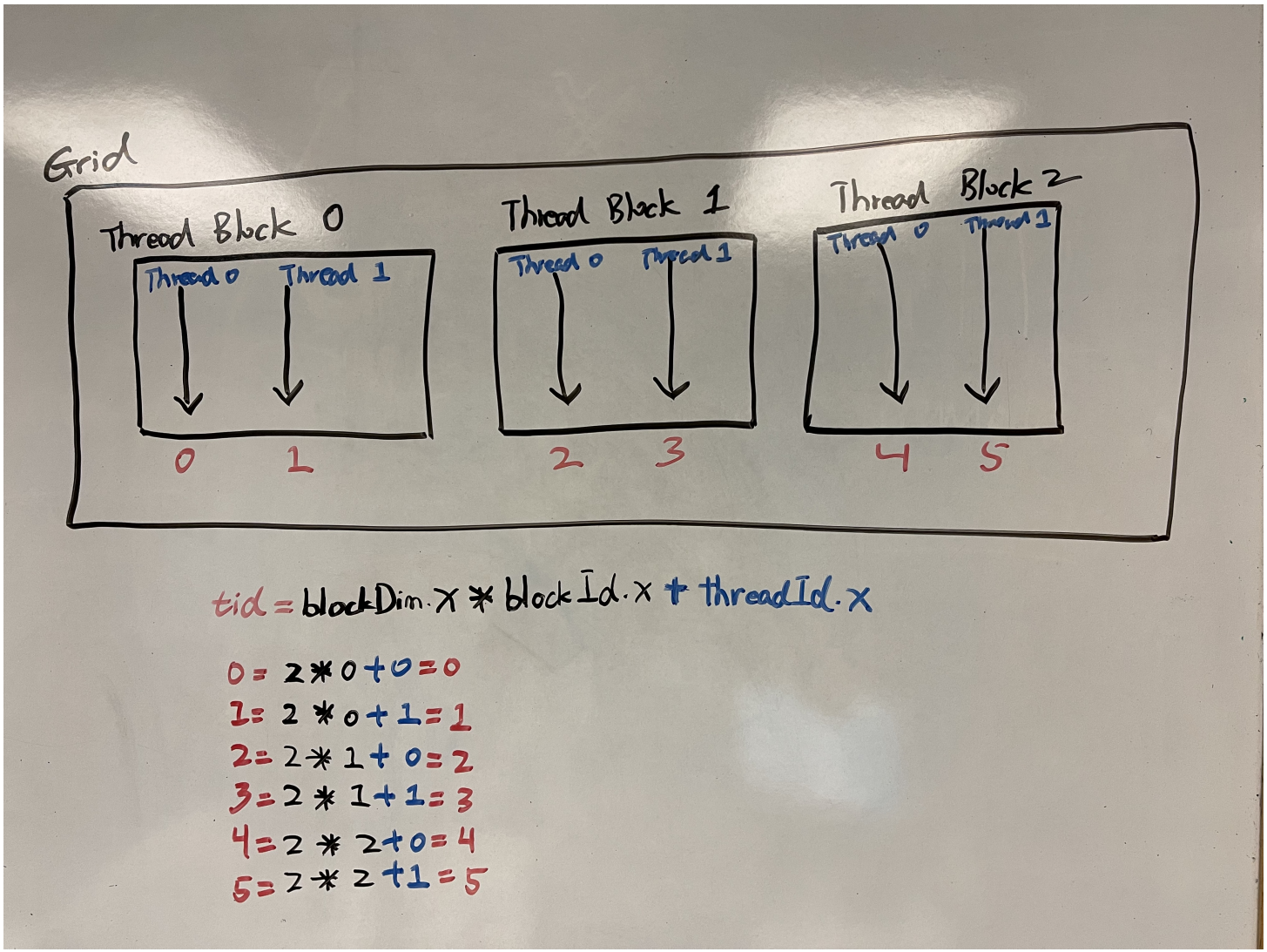
```
for(i = 0; i < N;
i++) {
    A[i] += 2;
}
```

OpenMP

```
#pragma omp parallel for
for(i = 0; i < N; i++) {
    A[i] += 2;
}
```

CUDA

```
int threadID = blockIdx.x * blockDim.x + threadIdx.x
A[threadID] += 2;
```


Grid
Thread Block 0
Thread 0
Thread 1
0
1
Thread Block 1
Thread 0
Thread 1
2
3
Thread Block 2
Thread 0
Thread 1
4
5
tid = blockDim.x * blockId.x + threadId.x
0 = 2*0+0=0
1= 2*0+1=1
2= 2*1+0=2
3=2*1+1=3
4=2*2+0=4
5=2*2+1=5

Genetic Algorithm (GA):

- GA refers to a heuristic search process that mimics the process of natural evolution in order to find approximate solutions for a specified problem.

Genetic Algorithm (GA):

- GA refers to a heuristic search process that mimics the process of natural evolution in order to find approximate solutions for a specified problem.
- They operate with techniques such as selection, crossover, and mutation in order to generate candidate solutions, called individuals.

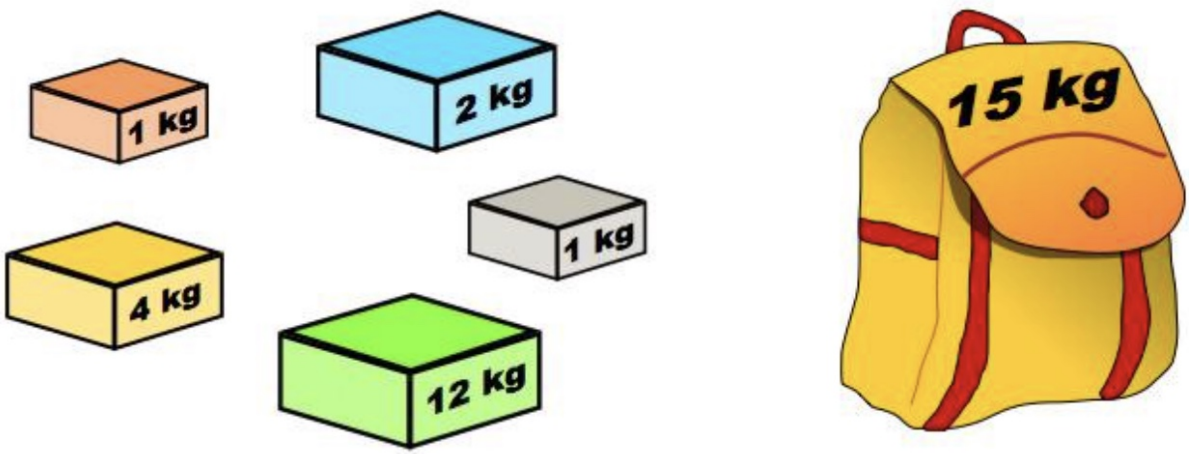
Genetic Algorithm (GA):

- GA refers to a heuristic search process that mimics the process of natural evolution in order to find approximate solutions for a specified problem.
- They operate with techniques such as selection, crossover, and mutation in order to generate candidate solutions, called individuals.
- GAs are generally employed in complex optimization and search problems - usually NP.

More Information

Knapsack (0-1 Knapsack):

Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.



Knapsack problem has a simple representation in genetic algorithms.

- Each chromosome is a binary string of bits, with each value encoding whether object *i* has been considered as part of the solution or not.
- For fitness function, the sum of the values for each object in the solution can be considered and scaled.

- The nature of a genetic algorithm is very well suited for a massive parallel architecture like CUDA, because each individual (encoding a candidate solution in the search space) can be independently computed and analyzed.
- In order to make better use of CUDA, Island based implementation for GA is used to isolate populations within a block and minimize inter-block communication.

This approach benefits both the algorithm itself as well as for GPU implementation.

- For algorithm, subsets of individuals locally search for an optimum.
- For GPU implementation, no inter-block synchronization is required, each island evolving in a separate block.

Mapping:

- Mapping each population on a block.
- Mapping each genome in a population on a thread.

SAT:

- The SAT problem are from the DIMACS benchmark instances. These are random problems with hard generator, no single clauses and hard density.
- Each test is formed of roughly 800 CNF statements of 100 distinct variables.
- 15 different tests performed, each being run an average of 5 times.

Knapsack:

- The tests were 100 different knapsack instances having the number of objects set to 40, with weights and values being integers less than 100.

Hardware:

CPU: Intel Q6600 quad core running at 2.4GHz
GPU: Nvidia GeForce GTX260 (192 CUDA cores, 896 MB GDDR3 memory, released in 2008)

Hardware:

CPU: Intel Q6600 quad core running at 2.4GHz
GPU: Nvidia GeForce GTX260 (192 CUDA cores, 896 MB GDDR3 memory, released in 2008)

Table 1 – Time performance of the sequential and parallel versions, with different GA settings

Islands	1		16		32		128	
Pop size	32	128	32	128	32	128	32	128
	SAT problem							
$T_{seq}(ms)$	125	143	1255	1717	2548	3091	10240	12117
$T_{par}(ms)$	267	360	298	387	317	421	340	1014
Speedup	0.47	0.39	4.21	4.43	8.04	7.34	**30.12**	11.95
	Knapsack problem							
$T_{seq}(ms)$	17	25	131	281	255	551	1008	2237
$T_{par}(ms)$	10	17	11	19	12	23	15	65
Speedup	1.7	1.47	11.9	14.8	21.3	23.9	**67.2**	34.4

Hardware:

- Two quad-core Intel Xeon CPUs
- Nvidia Tesla K20 GPU (2496 CUDA cores, 5 GB GDDR5 memory, released in 2012)

Just for reference, here are the specs of the state-of-the-art GPU for deep learning from Nvidia: **Tesla A100** (6912 CUDA cores, 40GB/80GB HBM2 memory, released in 2020)

Moore's Law: the number of transistors on a microchip doubles every two years, still valid?

Brute Force Solving NP-Complete problems using CUDA

Test Results

Table 1 - TSP Times and Speedups

Device	Time (min)	Speedup
CPU: 1 core	112.5	-----
CPU: 2 cores	56.2	2.0
CPU: 4 cores	28.1	4.0
CPU: 8 cores	14.1	8.0
GPU	5.7	19.8
Coprocessor	24.9	4.5

Table 2 - Party Problem Times and Speedups

Device	Time (min)	Speedup
CPU: 1 core	138.0	-----
CPU: 2 cores	69.1	2.0
CPU: 4 cores	34.6	4.0
CPU: 8 cores	17.3	8.0
GPU	1.7	79.4
Coprocessor	19.8	6.9

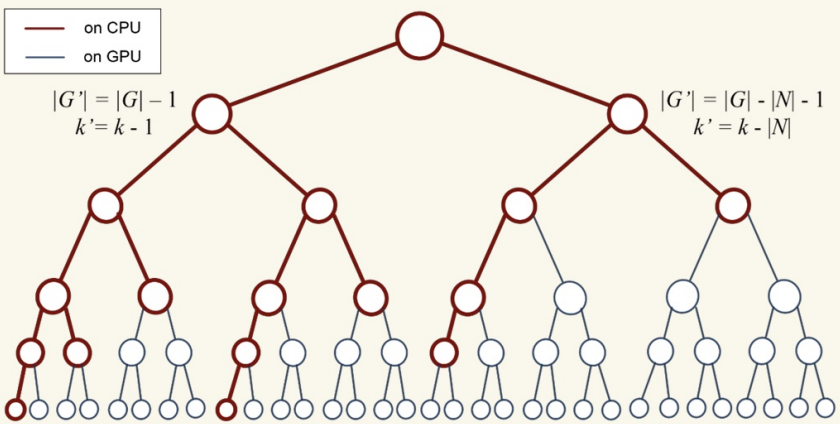
Table 3 - Knapsack Problem Times and Speedups

Device	Time (min)	Speedup
CPU: 1 core	419.5	-----
CPU: 2 cores	212.0	2.0
CPU: 4 cores	106.6	3.9
CPU: 8 cores	53.8	7.8
GPU	30.4	13.8
Coprocessor	68.0	6.2

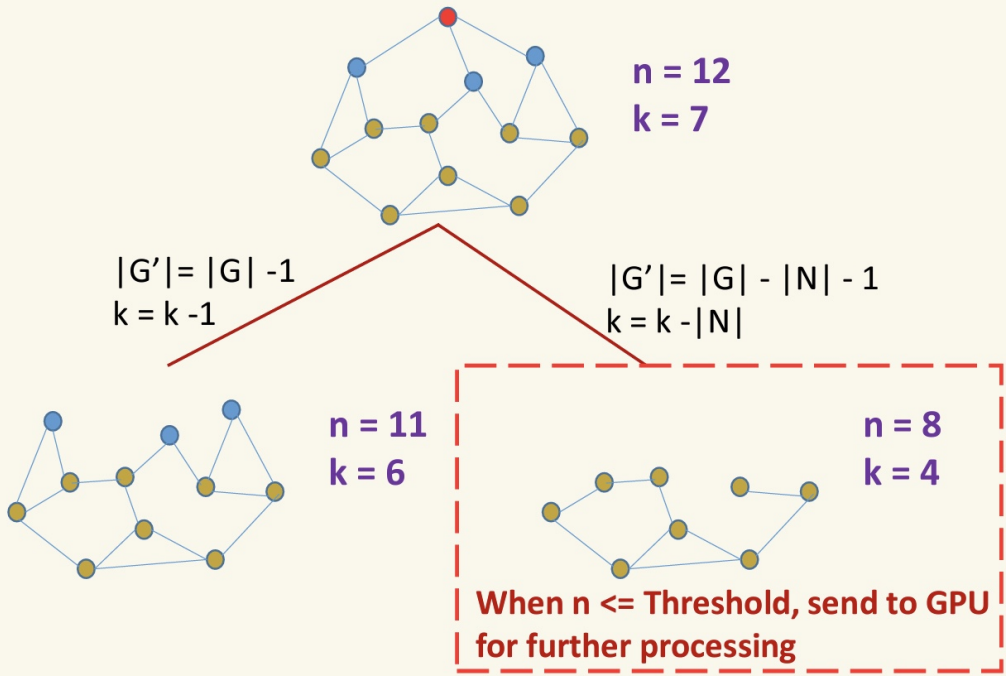
Solving FPT on Vertex Cover using CUDA

Branching Process

- Find max degree vertex v
- Two branch sets: v is in vertex cover or v's neighbors are in vertex cover



Branch Searching Process

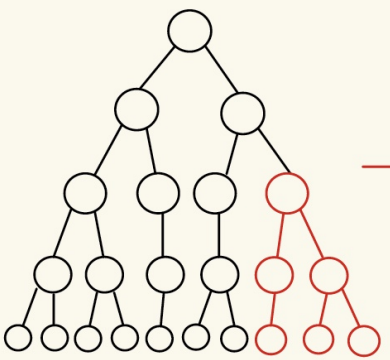


Solving FPT on Vertex Cover using CUDA

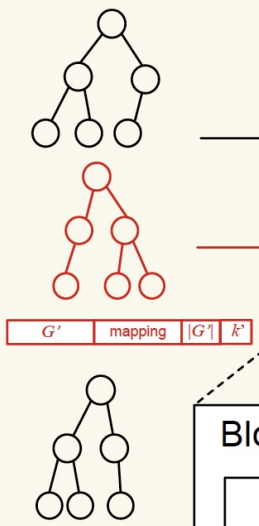
Reduction Rules

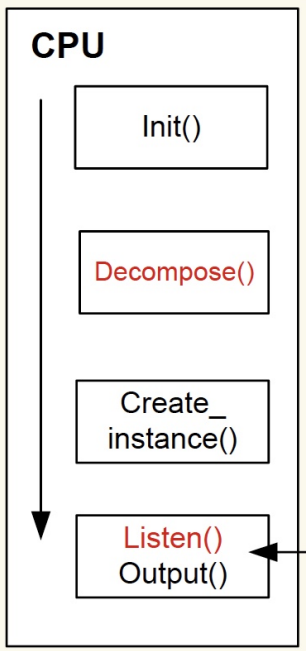
- No branching for vertices with:
  - degree greater than k
  - degree one
  - degree two if max degree is two

The original graph's algorithm tree



The reduced algorithm tree of an instance (subgraph with $|G'| <= Threshold$)





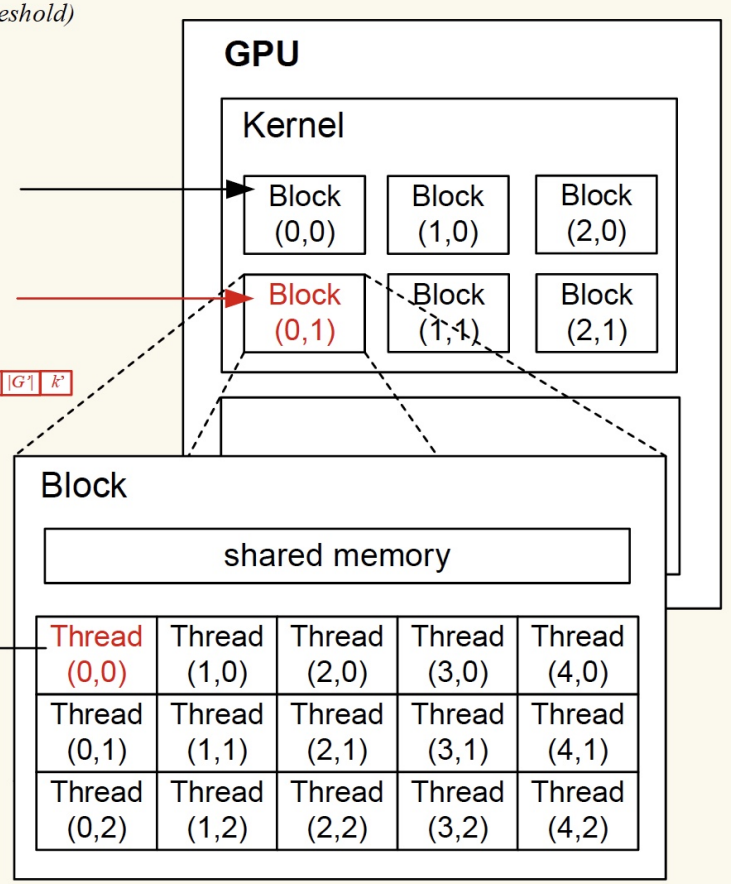
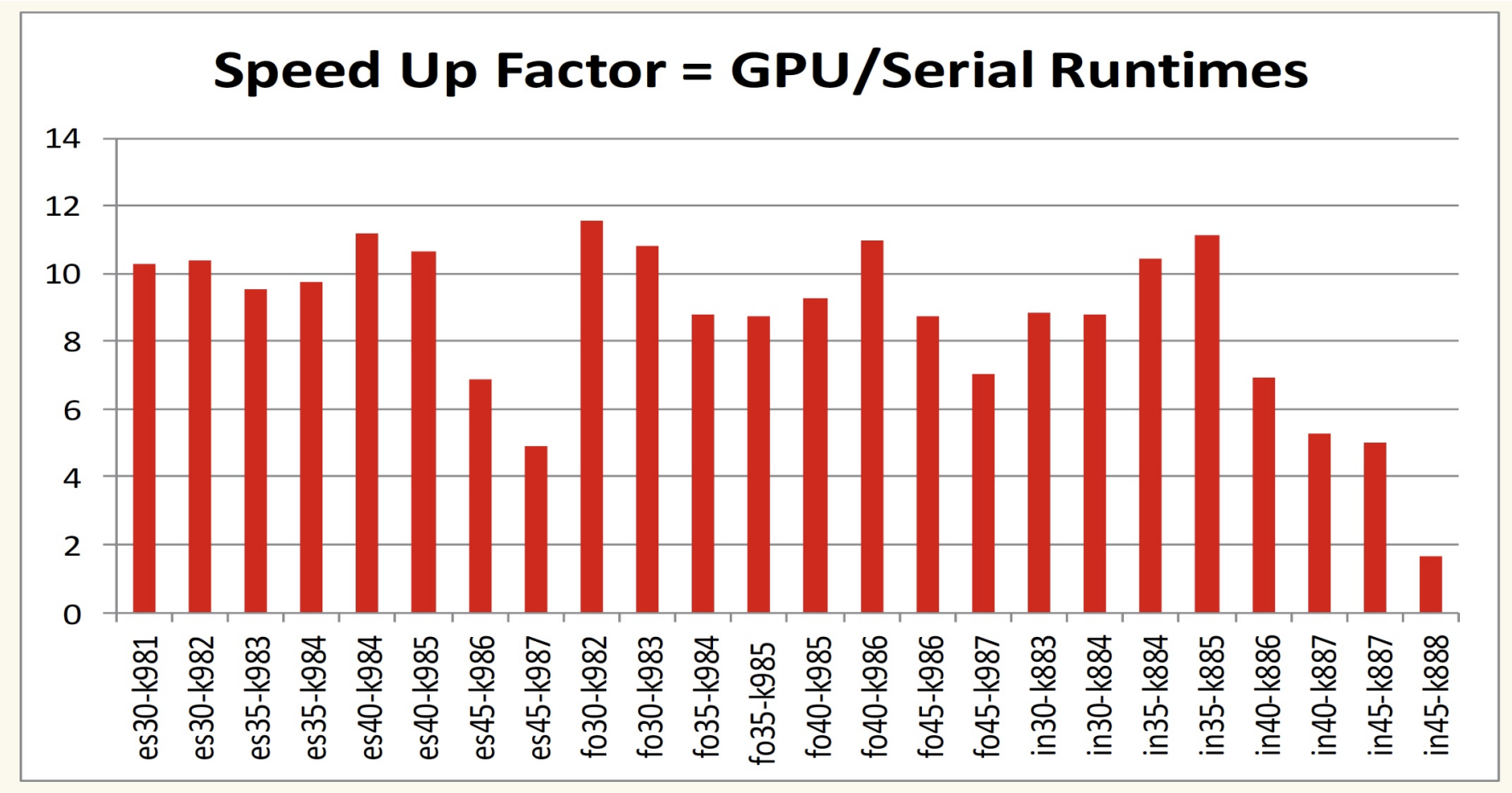


Figure 4 — Dis tribution and Synchronization of Computation

Solving FPT on Vertex Cover using CUDA

Hardware: Intel 10-core CPU and Nvidia Tesla K20



Q&A

Any Questions?

Reference Papers

- M. C. Feier, C. Lemnaru and R. Potolea, "Solving NP-Complete Problems on the CUDA Architecture Using Genetic Algorithms," 2011 10th International Symposium on Parallel and Distributed Computing, 2011, pp. 278-281, doi: 10.1109/ISPDC.2011.50.
- Toth, D.M., Goodwyn, Z., & Mueller, J. (2014). Using NP-Complete Problems to Compare a CPU, GPU, and the Intel® Xeon Phi™ Coprocessor,
- Y. Liu and J. Ju, "Finding Vertex Cover: Acceleration via CUDA," GPU Technology Conference, 2015.