Parallel Approaches to Shortest-Path Problems for Multilevel Heterogeneous Computing

PhD. Dissertation

Héctor Ortega Arranz

Advisors: Dr. Diego R. Llanos Ferraris
Dr. Arturo González Escribano

October 15th, 2015

Grupo Trasgo
Universidad de Valladolid

Departamento de Informática
Universidad de Valladolid

Universidad de Valladolid

Acknowledgements

This research has been partially supported by:

- the FPI-UVa and HiPEAC scholarships,
- the Ministerio de Economía y Competitividad: MOGECOPP project, CAPAP-H network, and HomProg-HetSys project;
- Junta de Castilla y León: ATLAS project; and
- ERDF program of the European Union: the COST Program Action IC1305: NESUS.

Outline

Introduction

The SSSP problem

Using GPUs to solve the SSSP problem

Values for GPU configuration parameters

The APSP problem

Using GPUs to solve the APSP problem

Using heterogeneous computing to solve the APSP problem

TuCCompi Programming Model

Conclusions

Introduction

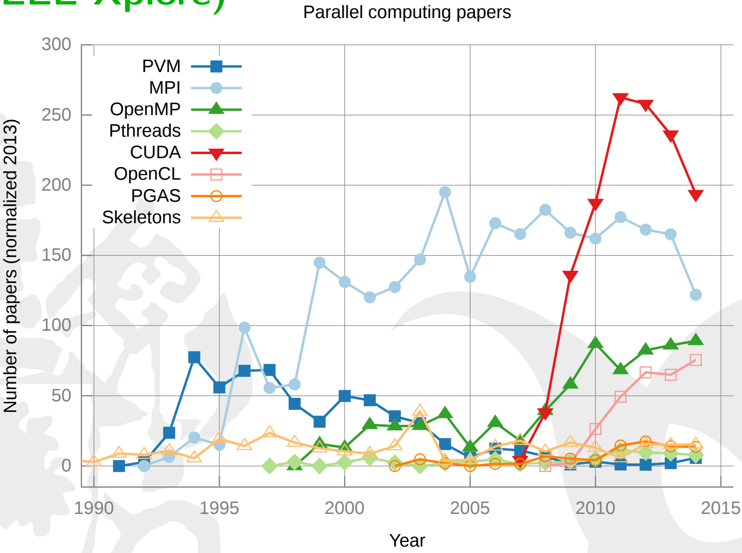
Motivation: The shortest-path problem

- **What is it?**
  The problem of finding the path between two or more locations with the lowest cost.
- **Where does it appear?**
  Many problems that arise in real-world networks imply its computation:
  - car navigation systems, traffic simulations.
  - spatial databases, web searching.
  - Internet route planners.
- **Why is it a matter of research?**
  Algorithms are still computationally costly.
- **Which is a possible solution?**
  To apply parallel computing mechanisms.

An overview to parallel computing

- Use two or more devices at the same time.
- Reduce the high temporal costs.
- **Multi-core systems:**
  CPUs that contain two or more all-purpose processing cores.
- **Many-core systems:**
  Devices with high number of processing units:
  - Supercomputers (↑#CPUs with ↑#Cores)
  - CPU coprocessors (XeonPhi)
  - Graphics Processing Units (GPUs)
- **Both** of them can be combined in **distributed environments**.

Parallel computing programming models (IEEE Xplore)

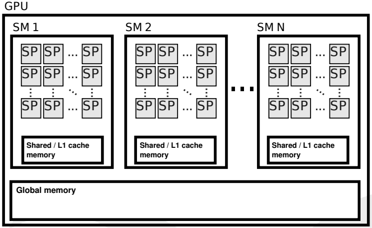


GPUs for parallel computing

8500GT	GTX 480	GTX 680	GTX TB	GTX TZ
16 cores	→ 480	→ 1536	→ 2880	→ 5760

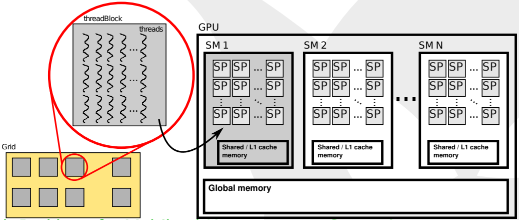
Hardware

- Single Processors (SP) or single cores.
- Multiprocessor (SM): Set of single processors.



Software

- GPU Thread: Executed in a SP.
- ThreadBlock: Set of GPU threads, executed in a SM.



Heterogeneous computing

Heterogeneous computing

- Computational units of different nature (e.g. CPUs & GPUs).
- Usually, different implementations for each unit type.
  ↪ Looking for maximum efficiency.
- Different execution times for each unit type.
  ↪ Creating system imbalances.

Load-balancing techniques

- Properly distribute the workload according to some criteria:
  - Computational capabilities, available resources, ...

Research question

Is it possible to develop techniques and tools to derive efficient parallel implementations to solve Shortest-Path problems using:

(1) *The new modern Graphics Processor Units (GPUs) and their corresponding tuning techniques, and*

(2) *Heterogeneous environments composed by such hardware accelerators together with traditional CPUs?*

The SSSP problem

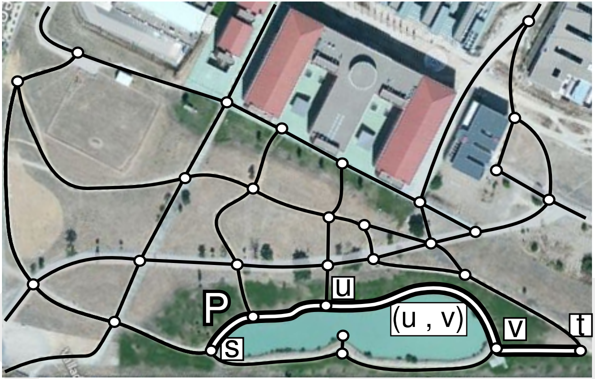
Brief introduction to graph theory I

NODES $\{v \in V\}$ $(n = |V|)$

- cities
- stations
- intersections

EDGES $\{e \in E\}$ $(m = |E|)$

- streets
- connections between nodes



PATH $\{P =< s, \ldots, u, v, \ldots, t >\}$

- sequence of nodes and edges between a source and a target.

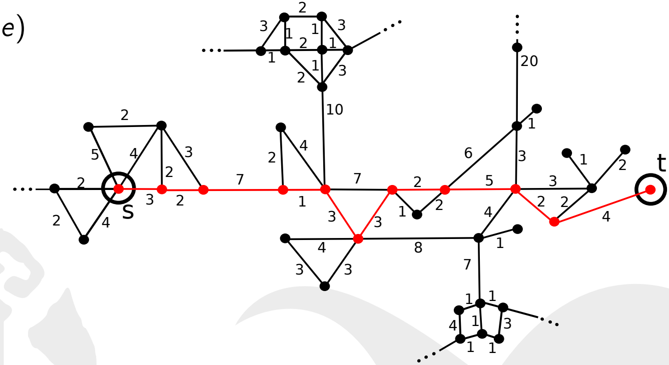
Brief introduction to graph theory II

WEIGHTS $w(e)$

- distance
- time
- fuel cost

WEIGHT of a PATH

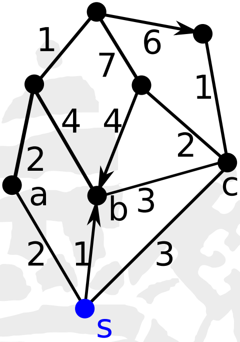
- $\sum_{e \in P} w(e)$



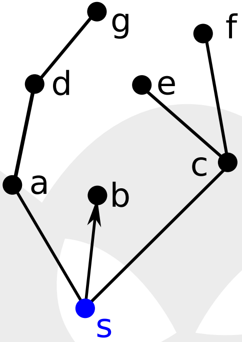
SHORTEST PATH between two nodes

- path with minimum weight among all possible paths.
- e.g. red path between node *s* and node *t*.

Single-Source Shortest-Path Problem (SSSP)



- Graph $G = (V, E)$.
- Weight function: $w(e) : e \in E$.
- Goal: Shortest path distance $d(s, x)$ for every $x \in V$.
- **Result: shortest path tree and shortest path distances.**



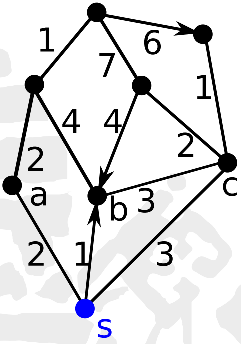
Best bounds for SSSP algorithms

Weight / type	Algorithm	Time complexity
Unweighted	BFS	$O(m+n)$
$\mathbb{R}_{\geq 0}$	Dijkstra	$O(m+n\log n)$
$\mathbb{R}_{\{1..C\}}$	Goldberg	$O(m+n\log C)$
$\mathbb{R}$ / Und	Pettie	$O(m+n\log\log n)$
$\mathbb{R}$	Bellman-Ford	$O(mn)$
$\mathbb{Z}_{\geq 0}$ / Und	Thorup	$O(m+n)$
$\mathbb{Z}_{\{0..C\}}$ / Dir	Thorup	$O(m+n\log\log min\{n,C\})$
$\mathbb{Z}_{\notin\{0,1\}}$	Goldberg	$O(m\sqrt{n}\log min\{w(e):e\in E\})$

Best Bounds for SSSP Algorithms

Weight / type	Algorithm	Time complexity
Unweighted	BFS	$O(m+n)$
$\mathbb{R}_{\geq 0}$	Dijkstra	$O(m+n\log n)$
$\mathbb{R}_{\{1..C\}}$	Goldberg	$O(m+n\log C)$
$\mathbb{R}$ / Und	Pettie	$O(m+n\log\log n)$
$\mathbb{R}$	Bellman-Ford	$O(mn)$
$\mathbb{Z}_{\geq 0}$ / Und	Thorup	$O(m+n)$
$\mathbb{Z}_{\{0..C\}}$ / Dir	Thorup	$O(m+n\log\log min\{n,C\})$
$\mathbb{Z}_{\notin\{0,1\}}$	Goldberg	$O(m\sqrt{n}\log min\{w(e): e\in E\})$

Dijkstra's algorithm: Initialization step

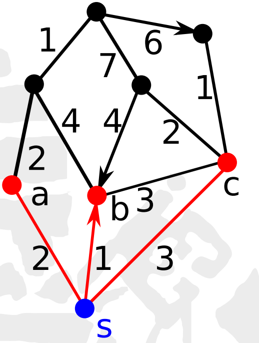


	s	a	b	c		
δ	**0**	∞	∞	∞	∞	∞

Dijkstra algorithm steps:

- 1. Initialization
  - $\boxed{\text{Frontier node} \leftarrow \mathbf{s}}$
  - $\boxed{\delta(\mathbf{s}) \leftarrow 0}$

Dijkstra's algorithm: Relaxation step

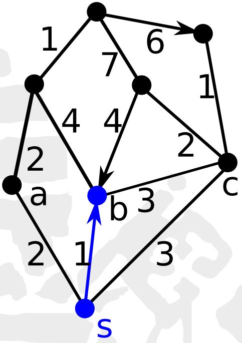


	s	a	b	c		
δ	0	2	1	3	∞	∞

Dijkstra algorithm steps:

- 1. Initialization
- 2. Edge relaxation
    - $$\boxed{\delta(a) = \min\{\delta(a), \delta(s) + w(s,a)\}}$$

Dijkstra's algorithm: Settlement step



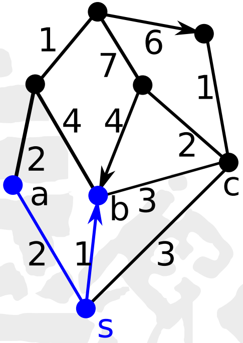
	s	a	b	c		
δ	**0**	2	**1**	3	∞	∞

Dijkstra algorithm steps:

- 1. Initialization
- 2. Edge relaxation
- 3. Settlement
  - 3.1 Minimum calculation
  - 3.2 Update frontier node

    Frontier node ← **b**
- 4. Termination criterion

Dijkstra's algorithm: Settlement step II



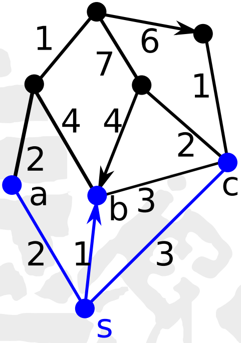
	s	a	b	c		
δ	**0**	**2**	**1**	3	∞	∞

Dijkstra algorithm steps:

- 1. Initialization
- 2. Edge relaxation
- 3. Settlement
  - 3.1 Minimum calculation
  - 3.2 Update frontier node

    Frontier node ← **a**
- 4. Termination criterion

Dijkstra's algorithm: Settlement step III



	s	a	b	c		
δ	**0**	**2**	**1**	**3**	∞	∞

Dijkstra algorithm steps:

- 1. Initialization
- 2. Edge relaxation
- 3. Settlement
  - 3.1 Minimum calculation
  - 3.2 Update frontier node

    Frontier node ← **c**
- 4. Termination criterion

Parallel SSSP approaches (Π-SSSP)

- Parallel deployment of sequential SSSP in disjoint subgraphs

- Parallelizing the internal operations:
  - Inner loop: Single-vertex edge-relaxation parallelism.
  - Outer loop: Multiple-vertex sequential-edge relaxation parallelism.

Algorithm	**Year**	**Parallelization**
Fine-Grain Parallel SSSP	1997	Inner
Crauser	1998	Outer
Δ-stepping	2003	Outer
GPU Label-Correcting	2007	Outer
GPU Parallel Dijkstra-Martín	2009	Outer
GPU Parallel Bellman-Ford	2011	Inner-Outer
Coarse-Grain Parallel SSSP	1997	Disjoint
Tang	2008	Disjoint

Parallel SSSP approaches (Π-SSSP)

- Parallel deployment of sequential SSSP in disjoint subgraphs
- Parallelizing the internal operations:
  - Inner loop: Single-vertex edge-relaxation parallelism.
  - Outer loop: Multiple-vertex sequential-edge relaxation parallelism.

Algorithm	**Year**	**Parallelization**
Fine-Grain Parallel SSSP	1997	Inner
Crauser	1998	Outer
Δ-stepping	2003	Outer
GPU Label-Correcting	2007	Outer
GPU Parallel Dijkstra-Martín	2009	Outer
GPU Parallel Bellman-Ford	2011	Inner-Outer
Coarse-Grain Parallel SSSP	1997	Disjoint
Tang	2008	Disjoint

Dijkstra outer loop parallelization

- Suitable for GPUs.

At each iteration i:

- Identify the unsettled nodes which $\boldsymbol{\delta(u)}$ **cannot be reduced.**
- **Frontier set instead** a single frontier node, in iteration $i+1$.
- **Perform the relaxation step** of next iteration from many frontier nodes **in parallel**.

Goal

- $\boldsymbol{\Delta_i}$ **threshold**: maximum value for a tentative distance to be considered as a shortest path, in each iteration.

Martín *et al.* solution [23]

- $\Delta_i = \min\{\delta(u) : u \in U_i\}$.

Using GPUs to solve the SSSP problem

Dijkstra's algorithm implementation for GPUs

Developed using **CUDA**

GPU pseudocode for main loop (Martín *et al.*)

```
while ($\Delta \neq \infty$) do
    gpu_kernel_relax(U, F, $\delta$);                          //Edge relaxation
    cudaDeviceSynchronize();
    vec_minimals = gpu_kernel_minimum(U, $\delta$);    //Settlement step_1
    $\Delta$ = min( vec_minimals )
    gpu_kernel_update(U, F, $\delta$, $\Delta$);                 //Settlement step_2
    cudaDeviceSynchronize();
end while
```

Martín *et al.*: Relax kernel

relax kernel

- One thread per node.
- Only outgoing edges of frontier nodes are relaxed.
- Race condition: atomic *min* operation needed.

GPU code for relax kernel

```
tid = thread.Id;
if (F[tid] == TRUE) then
    for all j successor of tid do
        if (U[j] == TRUE) then
            $\delta[j] = atomic\_min\{\delta[j], \delta[tid] + w(tid, j)\}$;
        end if
    end for
end if
```

Martín *et al.*: Minimum & update kernels

minimum kernel

- Min-reduction of the tentative distances ($\delta(u)$).
- CUDA SDK reduce kernel.
- Compare_min operation instead of the addition.

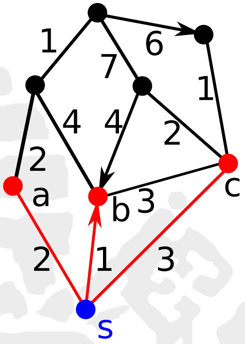
update kernel

- One thread per node.
- Frontier set erased.
- Still unsettled nodes inside the threshold:
    - Evicted from unsettled set.
    - Added to the frontier set.

GPU code for update kernel

```
tid = thread.Id;
F[tid]= FALSE;
if (U[tid]==TRUE and $\delta$[tid] == $\Delta$) then
    U[tid]= FALSE;
    F[tid]= TRUE;
end if
```

Improvement: A bigger threshold ...

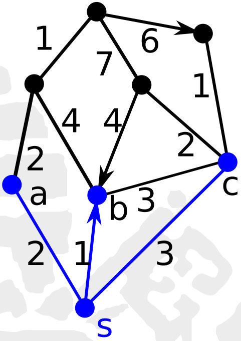


	s	a	b	c		
δ_0	**0**	**2**	**1**	**3**	∞	∞
ω	1	2	3	2	...	...

- Based on Crauser *et al.* [24]:
- $\boxed{\text{Vector } \omega\text{:}}$ Minimum weight of the outgoing edges for each node.
- $\boxed{\text{New minimum calculation:}}$

$$\Delta_i = \min\{\delta(u) + \boldsymbol{\omega(u)}\}$$

Improvement: ... implies more settled nodes



	s	a	b	c		
δ_0	**0**	**2**	**1**	**3**	∞	∞
ω	1	2	3	2	...	...

- Based on Crauser *et al.* [24]:
- Vector ω: Minimum weight of the outgoing edges for each node.
- New minimum calculation:

  $$\Delta_i = \min\{\delta(u) + \boldsymbol{\omega(u)}\}$$

- Update frontier set:

  for each node u with $\delta(u) \leq \Delta_i$

  Frontier set ← **a**, **b**, **c**

- **More nodes settled in each iteration**

Our improved implementation

Differences compared to Martín *et al.* (GPU Martín)

minimum: Reduction of $\boldsymbol{\delta(u) + \omega(u)}$: $u \in U$ instead of reduction of $\boldsymbol{\delta(u)}$: $u \in U$.

update: Due to bigger threshold Δ, settle unreached nodes with $\boldsymbol{\delta(u) \leq \Delta}$ instead of $\boldsymbol{\delta(u) = \Delta}$.

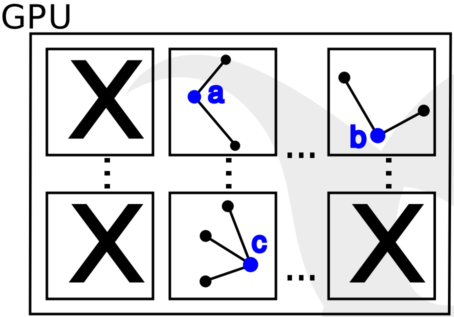
Our improved implementation

Differences compared to Martín *et al.* (GPU Martín)

minimum: Reduction of $\boldsymbol{\delta}(\boldsymbol{u}) + \boldsymbol{\omega}(\boldsymbol{u})$: $u \in U$ instead of reduction of $\boldsymbol{\delta}(\boldsymbol{u})$: $u \in U$.

update: Due to bigger threshold Δ, settle unreached nodes with $\boldsymbol{\delta}(\boldsymbol{u}) \leq \boldsymbol{\Delta}$ instead of $\boldsymbol{\delta}(\boldsymbol{u}) = \boldsymbol{\Delta}$.

No differences in **relax kernel** code but in behavior: **More parallelism**.



Experimental evaluation

Evaluate our improved solution (GPU Crauser)

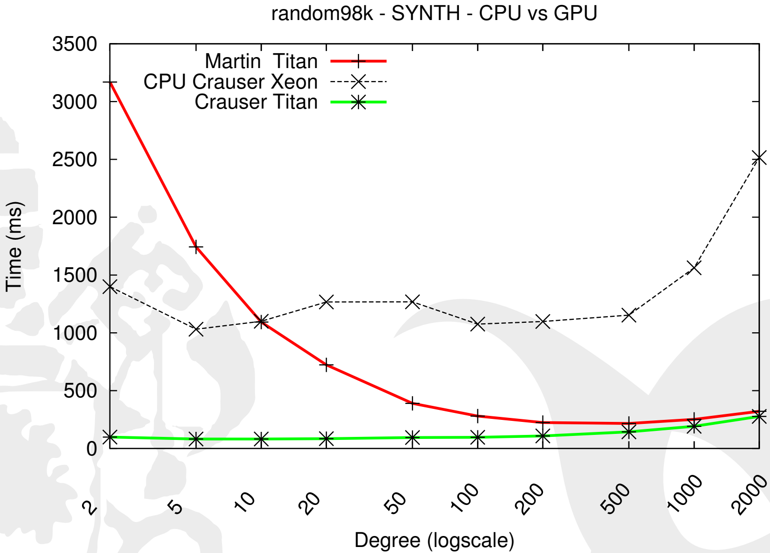
CPU: Intel(R) Xeon E5 2620 2.1GHz, memory of 32GB DDR3
GPU (Titan): a GeForce GTX Titan Black Kepler GK110B.

- Experiment I: **GPU Crauser vs. GPU Martín.**
- Experiment II: **GPU Crauser Opt. vs. BGL [25].**

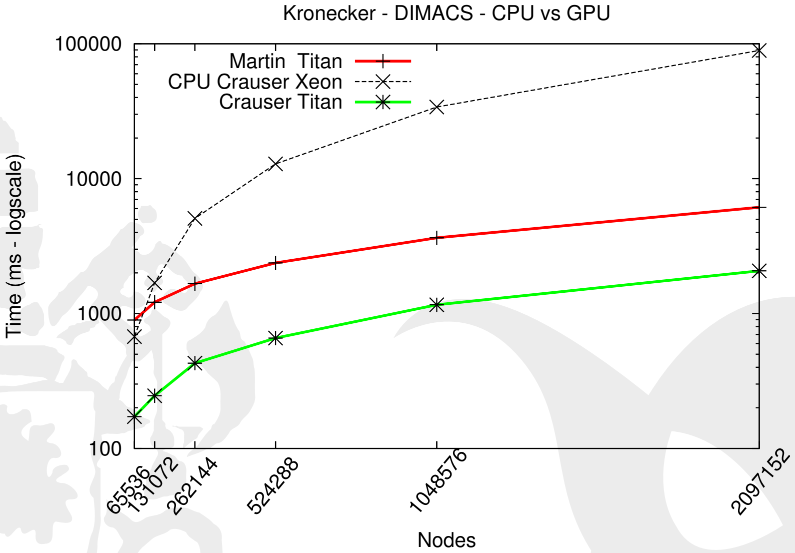
Input sets used:

- Synthetic graphs.
- Real-world and benchmarking graphs (RW&B).

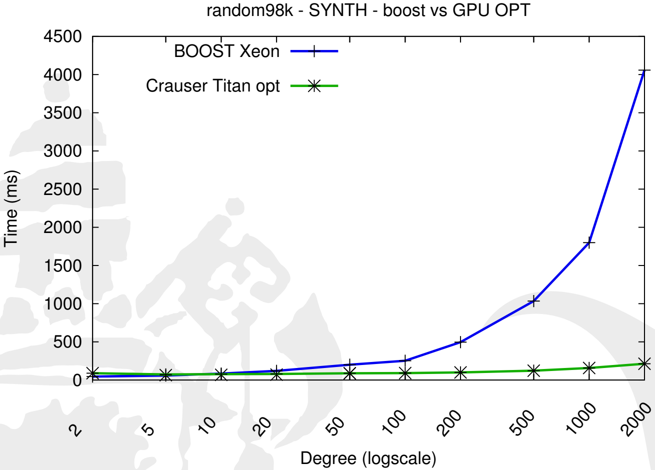
GPU Crauser vs. GPU Martín: Synthetic graphs



GPU Crauser vs. GPU Martín: RW&B graphs



GPU Crauser opt. vs. BGL: Execution times



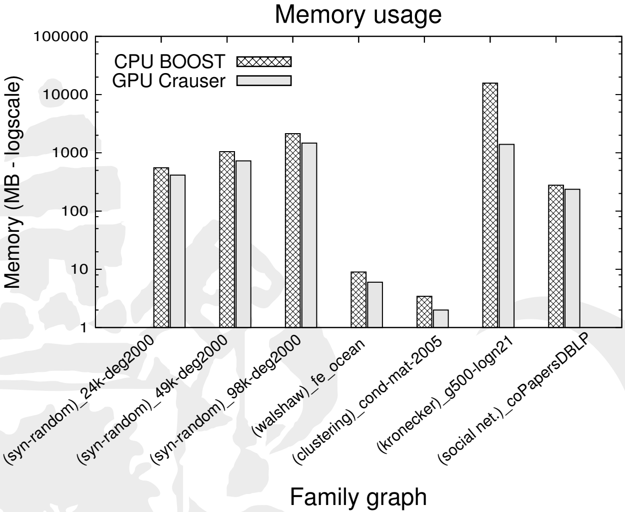
Synthetic graphs

- speedup up to 19×.

RW&B graphs

- speedup up to 4.76×.

GPU Crauser opt. vs. BGL: Memory space



Synthetic and RW&B graphs

- lower required memory space (up to 91%).

Values for GPU configuration parameters

Optimizing GPU applications

- Arriving to a GPU implementation is an affordable task.
- Optimizing its execution is the challenging activity.

- Tuning GPU configuration parameters for NVIDIA devices:
  - Threadblock size
  - L1 cache size

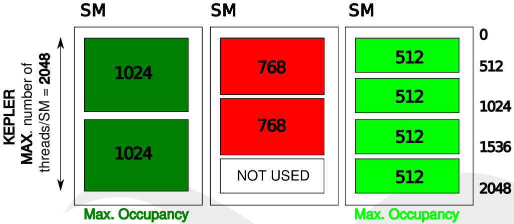
- Optimize without kernel code modifications.

Related work

- Lack of general studies.
- CUDA programming guidelines [13]:

→ Use maximum occupancy threadblock sizes:

$$\frac{Max.threads/SM}{Max.threadblocks/SM}$$



- CUDA recommendations **do not always lead to optimal performance**.
- uBench and kernel characterization model [16].

Our kernel characterization model

Code-dependent parameters [16]

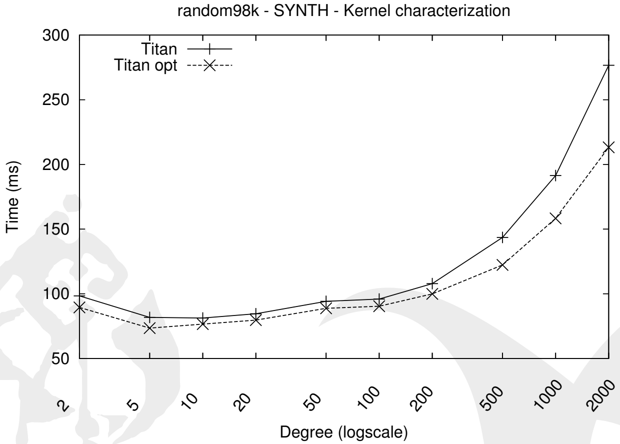
- Memory access pattern (MAp)
- Computational load ratio (CLr)
- Data sharing across blocks ratio (DSr)

Graph-dependent parameters *(new)*

- Size: number of vertices, n.
- Mean fan-out degree, $d(G)$.

- Intrincate dependences among these parameters.
- Prediction model [16] refined and extended.

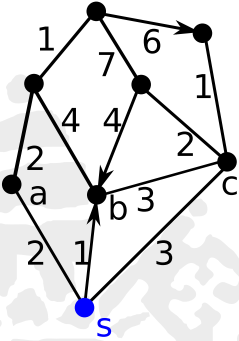
GPU Crauser vs. optimized: Synthetic graphs



- Predicted values leads to improvements up to 22.9%.

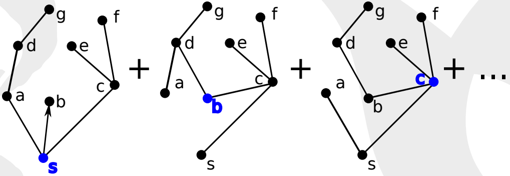
The APSP problem

All-Pair Shortest-Path problem (APSP)



- Graph $G = (V, E)$.
- Weight function: $w(e) : e \in E$.
- $\boxed{\text{Goal:}}$ Shortest path distances $d(u, v)$ for every $u, v \in V$.

- **$\boxed{\text{Result:}}$ *n* shortest-path trees and shortest path distances.**



APSP different sequential approaches

Dynamic Programming

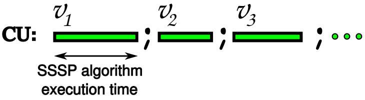
- Floyd-Warshall algorithm

- Temp. cost: $O(n^3)$
- Spatial cost: $O(n^2)$

- Dense graphs
  $m \in \Theta(n^2)$

Productivity Approach

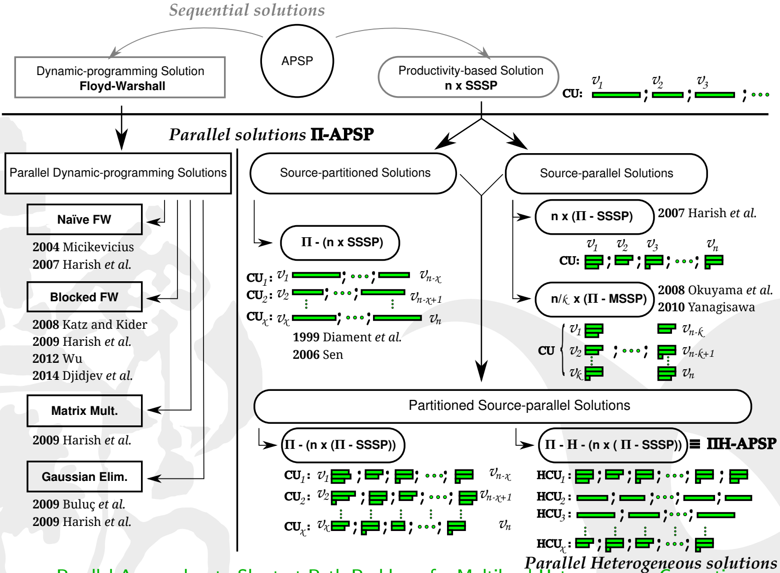
- n×SSSP algorithm



- Temp. cost: $O(mn + n^2 \log n)$
- Spatial cost: $O(n)$

- Sparse graphs
  $m \in O(n)$

Parallel APSP approaches (Π-APSP)



Sequential solutions

APSP

Dynamic-programming Solution
Floyd-Warshall

Productivity-based Solution
n x SSSP

CU: v_1 ; v_2 ; v_3 ; ···

Parallel solutions **Π-APSP**

Parallel Dynamic-programming Solutions

Naïve FW

2004 Micikevicius
2007 Harish *et al.*

Blocked FW

2008 Katz and Kider
2009 Harish *et al.*
2012 Wu
2014 Djidjev *et al.*

Matrix Mult.

2009 Harish *et al.*

Gaussian Elim.

2009 Buluç *et al.*
2009 Harish *et al.*

Source-partitioned Solutions

Π - (n x SSSP)

CU$_1$: v_1 ; ··· ; $v_{n-\chi}$
CU$_2$: v_2 ; ··· ; $v_{n-\chi+1}$
CU$_\chi$: v_χ ; ··· ; v_n

1999 Diament *et al.*
2006 Sen

Source-parallel Solutions

n x (Π - SSSP) **2007** Harish *et al.*

CU: v_1 ; v_2 ; v_3 ; ··· ; v_n

n/k x (Π - MSSP) **2008** Okuyama *et al.*
2010 Yanagisawa

CU v_1 v_2 v_k ; ··· ; v_{n-k} v_{n-k+1} v_n

Partitioned Source-parallel Solutions

Π - (n x (Π - SSSP))

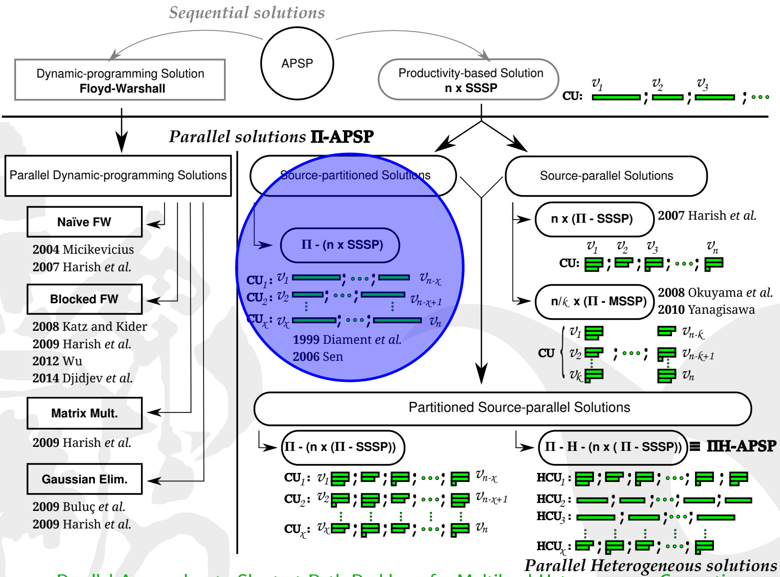
CU$_1$: v_1 ; ; ; ··· ; $v_{n-\chi}$
CU$_2$: v_2 ; ; ; ··· ; $v_{n-\chi+1}$
CU$_\chi$: v_χ ; ; ; ··· ; v_n

Π - H - (n x (Π - SSSP)) ≡ **ΠH-APSP**

HCU$_1$: ; ; ; ··· ; ;
HCU$_2$: ; ; ··· ; ;
HCU$_3$: ; ; ··· ;
HCU$_\chi$: ; ; ; ··· ;

Parallel Heterogeneous solutions

Π-APSP approaches



Sequential solutions

APSP

Dynamic-programming Solution
Floyd-Warshall

Productivity-based Solution
n x SSSP

CU: v_1 ; v_2 ; v_3 ; ···

Parallel solutions **Π-APSP**

Parallel Dynamic-programming Solutions

Naïve FW

2004 Micikevicius
2007 Harish *et al.*

Blocked FW

2008 Katz and Kider
2009 Harish *et al.*
2012 Wu
2014 Djidjev *et al.*

Matrix Mult.

2009 Harish *et al.*

Gaussian Elim.

2009 Buluç *et al.*
2009 Harish *et al.*

Source-partitioned Solutions

Π - (n x SSSP)

CU_1: v_1 ; ··· ; v_{n-x}
CU_2: v_2 ; ··· ; v_{n-x+1}
CU_x: v_x ; ··· ; v_n

1999 Diament *et al.*
2006 Sen

Source-parallel Solutions

n x (Π - SSSP) **2007** Harish *et al.*

CU: v_1 ; v_2 ; v_3 ; ··· ; v_n

n/k x (Π - MSSP) **2008** Okuyama *et al.*
2010 Yanagisawa

CU { v_1, v_2, v_k ; ··· ; v_{n-k}, v_{n-k+1}, v_n

Partitioned Source-parallel Solutions

Π - (n x (Π - SSSP))

CU_1: v_1 ; ; ; ··· ; v_{n-x}
CU_2: v_2 ; ; ; ··· ; v_{n-x+1}
CU_x: v_x ; ; ; ··· ; v_n

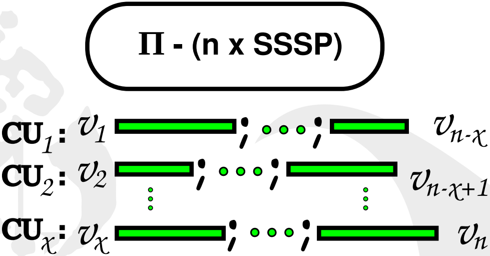
Π - H - (n x (Π - SSSP)) ≡ **ΠH-APSP**

HCU_1 : ; ; ; ··· ; ;
HCU_2 : ; ; ··· ;
HCU_3 : ; ; ··· ;
HCU_x : ; ; ; ··· ;

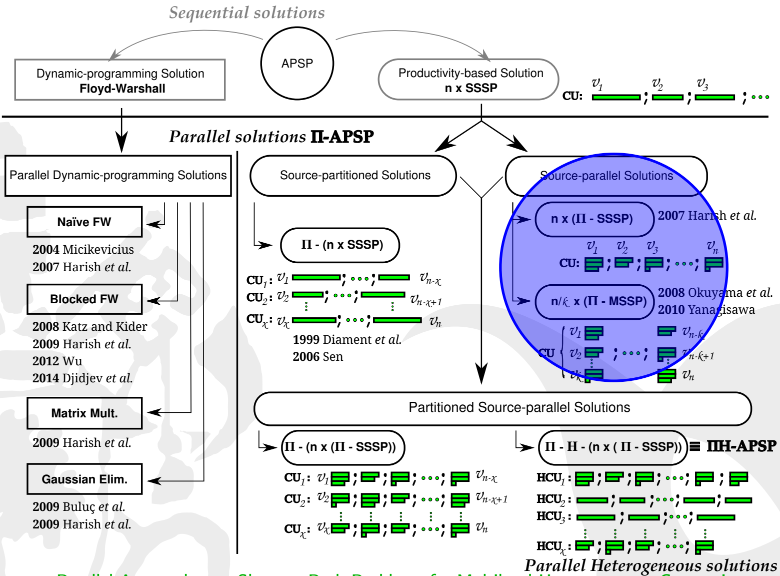
Parallel Heterogeneous solutions

Π-APSP: Productivity-based solutions I

- **Source-partitioned Solutions**,
  parallelize the serial n executions of a sequential SSSP.


Π - (n x SSSP)
CU$_1$: v_1 ; ... ; $v_{n-\chi}$
CU$_2$: v_2 ; ... ; $v_{n-\chi+1}$
CU$_\chi$: v_χ ; ... ; v_n

Π-APSP approaches



Sequential solutions

APSP

Dynamic-programming Solution
Floyd-Warshall

Productivity-based Solution
n x SSSP

CU: v_1 ; v_2 ; v_3 ; ···

Parallel solutions **Π-APSP**

Parallel Dynamic-programming Solutions

Naïve FW

2004 Micikevicius
2007 Harish *et al.*

Blocked FW

2008 Katz and Kider
2009 Harish *et al.*
2012 Wu
2014 Djidjev *et al.*

Matrix Mult.

2009 Harish *et al.*

Gaussian Elim.

2009 Buluç *et al.*
2009 Harish *et al.*

Source-partitioned Solutions

Π - (n x SSSP)

CU_1: v_1 ; ··· ; $v_{n-\chi}$
CU_2: v_2 ; ··· ; $v_{n-\chi+1}$
CU_χ: v_χ ; ··· ; v_n

1999 Diament *et al.*
2006 Sen

Source-parallel Solutions

n x (Π - SSSP) **2007** Harish *et al.*

CU: v_1 ; v_2 ; v_3 ; ··· ; v_n

n/k x (Π - MSSP) **2008** Okuyama *et al.*
2010 Yanagisawa

CU v_1 ; v_2 ; v_k ; ··· ; v_{n-k} ; v_{n-k+1} ; v_n

Partitioned Source-parallel Solutions

Π - (n x (Π - SSSP))

CU_1: v_1 ; ; ; ··· ; $v_{n-\chi}$
CU_2: v_2 ; ; ; ··· ; $v_{n-\chi+1}$
CU_χ: v_χ ; ; ; ··· ; v_n

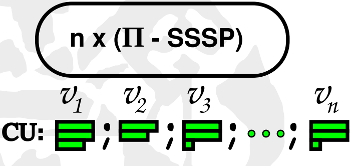
Π - H - (n x (Π - SSSP)) ≡ **ΠH-APSP**

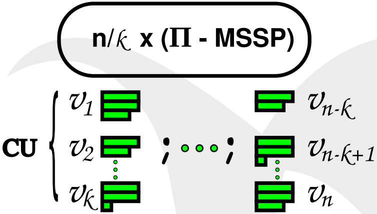
HCU_1 : ; ; ; ··· ; ;
HCU_2 : ; ; ··· ; ;
HCU_3 : ; ; ··· ;
HCU_χ : ; ; ; ··· ;

Parallel Heterogeneous solutions

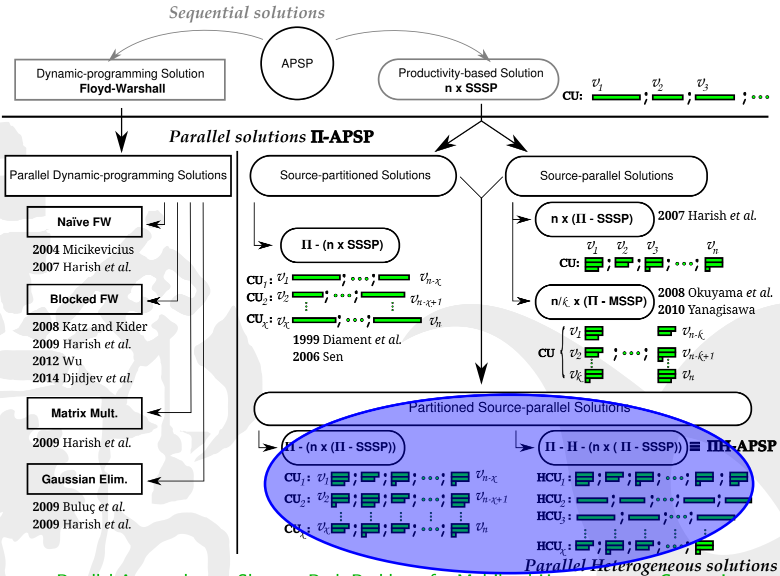
Π-APSP: Productivity-based Solutions II

- **Source-parallel Solutions**,
  involve the execution of a parallel SSSP algorithm.
  - (1) **Sequential Source-parallel Solutions**,
    sequential *n* executions of parallel SSSPs





Π-APSP Approaches



Sequential solutions

APSP

Dynamic-programming Solution
Floyd-Warshall

Productivity-based Solution
n x SSSP

CU: v_1 ; v_2 ; v_3 ; ···

Parallel solutions **Π-APSP**

Parallel Dynamic-programming Solutions

Naïve FW

2004 Micikevicius
2007 Harish *et al.*

Blocked FW

2008 Katz and Kider
2009 Harish *et al.*
2012 Wu
2014 Djidjev *et al.*

Matrix Mult.

2009 Harish *et al.*

Gaussian Elim.

2009 Buluç *et al.*
2009 Harish *et al.*

Source-partitioned Solutions

Π - (n x SSSP)

CU$_1$: v_1 ; ··· ; $v_{n-\chi}$
CU$_2$: v_2 ; ··· ; $v_{n-\chi+1}$
CU$_\chi$: v_χ ; ··· ; v_n

1999 Diament *et al.*
2006 Sen

Source-parallel Solutions

n x (Π - SSSP) **2007** Harish *et al.*

CU: v_1 ; v_2 ; v_3 ; ··· ; v_n

n/k x (Π - MSSP) **2008** Okuyama *et al.*
2010 Yanagisawa

CU v_1, v_2, v_k ; ··· ; v_{n-k}, v_{n-k+1}, v_n

Partitioned Source-parallel Solutions

Π - (n x (Π - SSSP))

CU$_1$: v_1 ; ; ; ··· ; $v_{n-\chi}$
CU$_2$: v_2 ; ; ; ··· ; $v_{n-\chi+1}$
CU$_\chi$: v_χ ; ; ; ··· ; v_n

Π - H - (n x (Π - SSSP)) ≡ **ΠH-APSP**

HCU$_1$: ; ; ; ··· ; ;
HCU$_2$: ; ; ··· ; ;
HCU$_3$: ; ; ··· ;
HCU$_\chi$: ; ; ; ··· ;

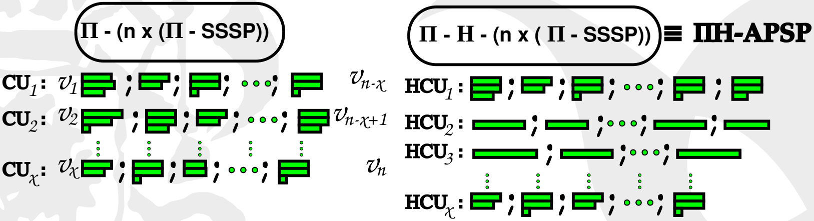
Parallel Heterogeneous solutions

Π-APSP: Productivity-based Solutions III

- **Source-parallel Solutions**,
  involve the execution of a parallel SSSP algorithm.
  - (2) **Partitioned Source-parallel Solutions**,
    parallel *n* executions of parallel SSSPs using:

homogeneous
computational units (CUs)

heterogeneous
computational units (HCUs)

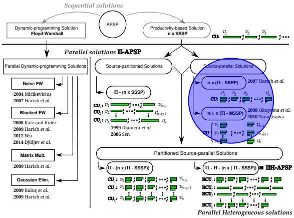

Π - (n x (Π - SSSP))
CU$_1$: v_1 ; ; ; ⋯ ; $v_{n-\chi}$
CU$_2$: v_2 ; ; ; ⋯ ; $v_{n-\chi+1}$
CU$_\chi$: v_χ ; ; ; ⋯ ; v_n
Π - H - (n x (Π - SSSP))
≡ ΠH-APSP
HCU$_1$: ; ; ; ⋯ ; ;
HCU$_2$: ; ; ⋯ ; ;
HCU$_3$: ; ; ⋯ ;
HCU$_\chi$: ; ; ; ⋯ ;

Using GPUs to solve the APSP problem

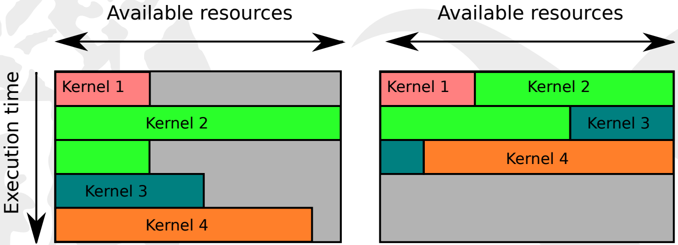
Goals

Solve the APSP problem by:

Serial executions of
a parallel algorithm
using a single device.


Sequential solutions
APSP
Dynamic-programming Solution
Floyd-Warshall
Productivity-based Solution
n x SSSP
CU: v_1 ; v_2 ; v_3 ; •••
Parallel solutions Π-APSP
Parallel Dynamic-programming Solutions
Naïve FW
2004 Micikevicius
2007 Harish et al.
Blocked FW
2008 Katz and Kider
2009 Harish et al.
2012 Wu
2014 Djidjev et al.
Matrix Mult.
2009 Harish et al.
Gaussian Elim.
2009 Buluç et al.
2009 Harish et al.
Source-partitioned Solutions
Π - (n x SSSP)
CU_1: v_1 ; ••• ; $v_{n-\chi}$
CU_2: v_2 ; ••• ; $v_{n-\chi+1}$
CU_χ: v_χ ; ••• ; v_n
1999 Diament et al.
2006 Sen
Source-parallel Solutions
n x (Π - SSSP)
2007 Harish et al.
CU: v_1 ; v_2 ; v_3 ; ••• ; v_n
n/k x (Π - MSSP)
2008 Okuyama et al.
2010 Yanagisawa
CU v_1 v_2 v_k ; ••• ; v_{n-k} v_{n-k+1} v_n
Partitioned Source-parallel Solutions
Π - (n x (Π - SSSP))
CU_1: v_1 ; ; ; ••• ; $v_{n-\chi}$
CU_2: v_2 ; ; ; ••• ; $v_{n-\chi+1}$
CU_χ: v_χ ; ; ; ••• ; v_n
Π - H - (n x (Π - SSSP)) ≡ ΠH-APSP
HCU_1 : ; ; ; ••• ; ;
HCU_2 : ; ; ••• ; ;
HCU_3 : ; ; ••• ;
HCU_χ : ; ; ; ••• ;
Parallel Heterogeneous solutions

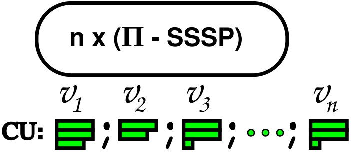
Improve the approach with the Concurrent Kernel execution (CK).



Serially-executed, source-parallel solutions

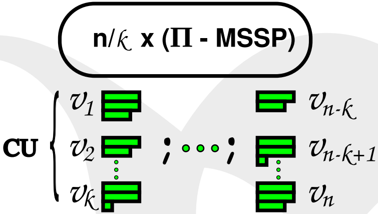
Single kernel solution:

- ***n*** serial executions of our GPU Crauser solution.



CK improvement:

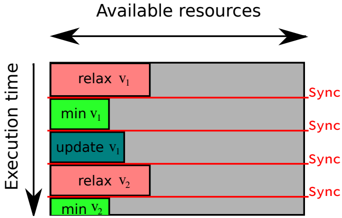
- ***k*** concurrent kernels.
- Each CK → different SSSP task.
- n/k serial executions of our GPU Crauser solution.



Serially-executed, source-parallel solutions

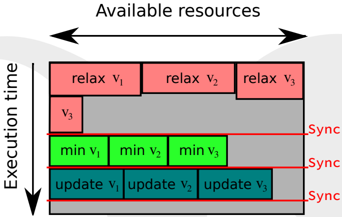
Single kernel solution:

- ***n*** serial executions of our GPU Crauser solution.



CK improvement:

- ***k*** concurrent kernels.
- Each CK → different SSSP task.
- n/k serial executions of our GPU Crauser solution.



Experimental evaluation

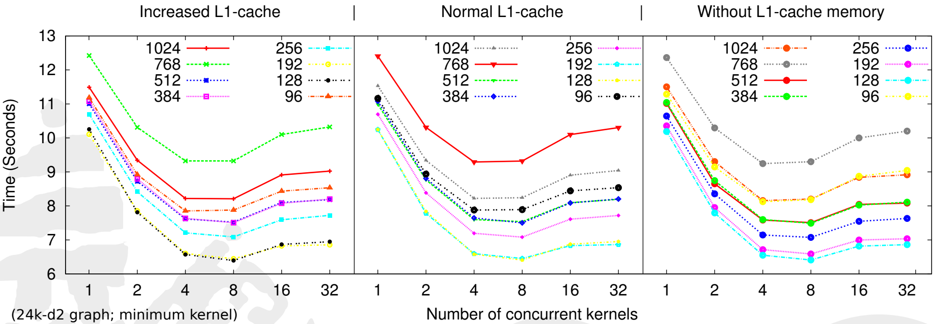
Evaluate the Concurrent kernel approach vs. single kernel

2 GPUs (Kepler GK104, Fermi GF100)

Experiment:

- Exhaustive combination of all key values for:
  - Number of concurrent kernels
  - Threadblock size
  - L1 Cache state
- **Study I:** CK performance impact.
- **Study II:** CK influence on GPU conf. parameters prediction.
- **Study III:** Validity of the prediction model, and usefulness for APSP.

CK performance impact and influence

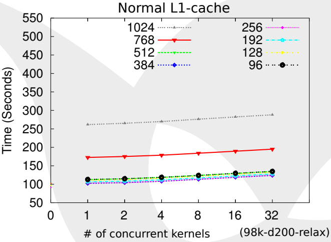


Study I:

- Performance gain of up to 52.8%.
- Little performance degradation.

Study II:

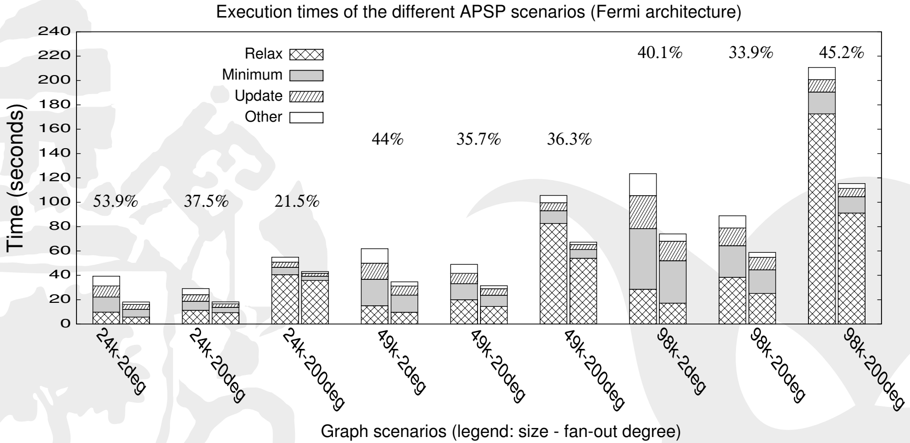
- Predictions are not affected.



Validation and usefulness

Study III:

- Correct predictions.
- Global performance gains in the range [33.7% – 58.5%] for Kepler.
- Global performance gains in the range [21.5% – 53.9%] for Fermi.

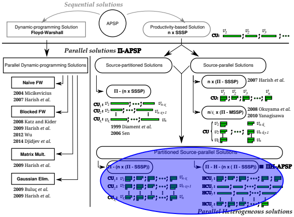


Using heterogeneous computing to solve the APSP problem

Goals

Solve the APSP problem by:

Parallel execution of
parallel SSSP algorithms
using heterogeneous systems.


Sequential solutions
APSP
Dynamic-programming Solution
Floyd-Warshall
Productivity-based Solution
n x SSSP
CU: v_1 ; v_2 ; v_3 ; …
Parallel solutions Π-APSP
Parallel Dynamic-programming Solutions
Naïve FW
2004 Micikevicius
2007 Harish et al.
Blocked FW
2008 Katz and Kider
2009 Harish et al.
2012 Wu
2014 Djidjev et al.
Matrix Mult.
2009 Harish et al.
Gaussian Elim.
2009 Buluç et al.
2009 Harish et al.
Source-partitioned Solutions
Π - (n x SSSP)
CU_1: v_1 ; … ; $v_{n-\chi}$
CU_2: v_2 ; … ; $v_{n-\chi+1}$
CU_χ: v_χ ; … ; v_n
1999 Diament et al.
2006 Sen
Source-parallel Solutions
n x (Π - SSSP)
2007 Harish et al.
CU: v_1 v_2 v_3 … v_n
n/k x (Π - MSSP)
2008 Okuyama et al.
2010 Yanagisawa
CU v_1 v_2 v_k ; … ; v_{n-k} v_{n-k+1} v_n
Partitioned Source-parallel Solutions
Π - (n x (Π - SSSP))
CU_1: v_1 … $v_{n-\chi}$
CU_2: v_2 … $v_{n-\chi+1}$
CU_χ: v_χ … v_n
Π - H - (n x (Π - SSSP)) ≡ ΠH-APSP
HCU_1
HCU_2
HCU_3
HCU_χ
Parallel Heterogeneous solutions

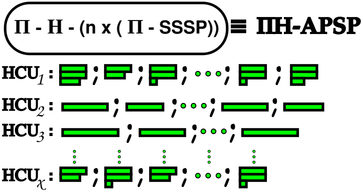
Improve by using load-balancing techniques.

- Properly distribute the workload according to some criteria:
    - Computational capabilities, available resources, . . .
- Depending on when the distribution is done:
    - **Static**: previously to the computing
    - **Dynamic**: during the computing

Our parallel-executed, source-parallel solution

Heterogeneous solution:

- parallel executions of our GPU Crauser impl. in **GPUs**, and the sequential version in **CPU-cores**.



Load-balancing policies:

- Equitable Scheduling.
  - Static.
  - Equal workspace distribution.
  - Ignores computational capabilities.
- Work-retrieving-queue Sched.
  - Dynamic.
  - Centralized queue.
  - Each unit retrieves a task when idle.

Experimental Evaluation

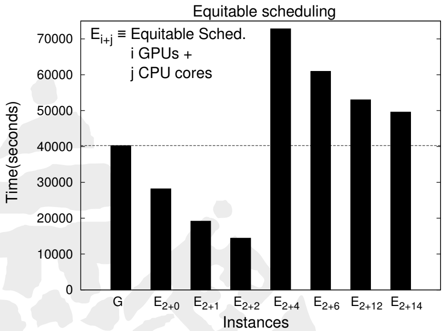
Evaluate the Heterogeneous system vs. One GPU

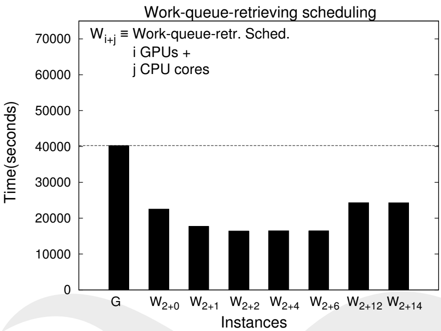
4 i7-CPUcores@3.2 GHz with hyperthreading, and
2 GPUs (Kepler GK104, Fermi GF100)

- **Experiment I: complete APSP.**
    - Workspace: *n* SSSP-tasks.
    - Tricky graphs 1M nodes due to Martín *et al.*:
        - Leads to 2 kind of shortest path trees.
    - 2 kind of tasks: heavy and light load.
    - Known load distribution.

- **Experiment II: random sources.**
    - Workspace: 512 tasks randomly selected.
    - Graphs ranging from 1 million to 11 millions of nodes.
    - Same features as in Experiment I.
    - Unknown load distribution due to random selection.

Experimental Results I: Complete APSP

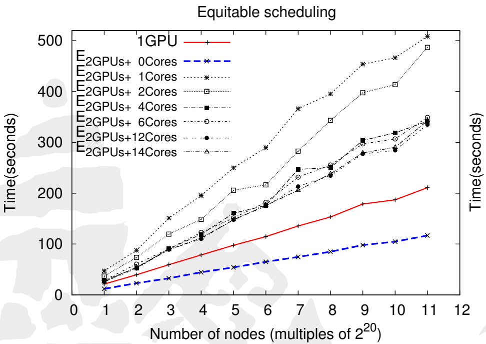


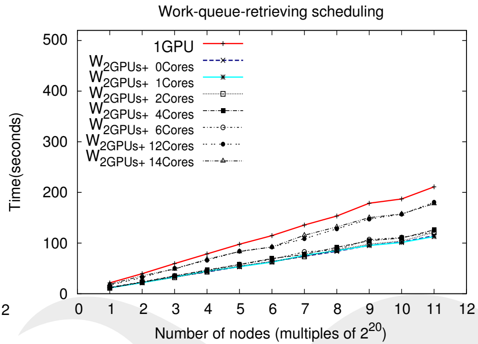


- $E_{2GPUs+0Cores}$ → 30% (1.43×).
- $E_{2GPUs+2Cores}$ → 65% (2.86×).
- $E_{2GPUs+(j>2Cores)}$ → Performance drops.

- $W_{2GPUs+0Cores}$ → 44% (1.78×).
- $W_{2GPUs+2Cores}$ → 60% (2.50×).
- $W_{2GPUs+(j>4Cores)}$ → Similar performance.

Experimental Results II: Random Sources





- $\mathrm{E}_{2GPUs+0Cores}$ → 45% (1.81×).
- $\mathrm{E}_{2GPUs+(j>0Cores)}$ → Performance drops.

- $\mathrm{W}_{2GPUs+0Cores}$ → 46% (1.85×).
- $\mathrm{W}_{2GPUs+1Cores}$ → 47% (1.89×).
- All → Better than reference.

Summary

- Using heterogeneous systems is useful, obtaining up to:
  - 2.86× for our graphs with a known load distribution.
  - 1.89× for our graphs with an unknown load distribution.

- A previous study of the graph features is important.
  - Equitable scheduling delivered the best performance when correctly tuned accordingly with the load distribution.
  - Work-queue retrieving scheduling is a safer choice when the load distribution is unknown.

TuCCompi Programming Model

Programming heterogeneous systems

- Much more difficult than programming homogeneous systems.
- Aim to exploit all computational resources.
- Need to combine different kind of mainstream programming models (CUDA, MPI, OpenMP, OpenCL, ...).

Facilitating the heterogeneous programming

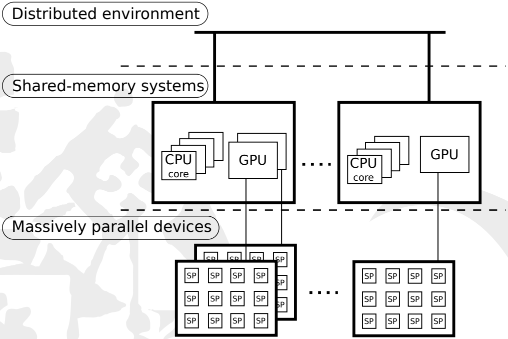
- **TuCCompi** programming model.
  Tuned, **C**oncurrent, **C**UDA, **O**pen**MP**, **MPI**.
- **Embarrassingly-parallel** problems:
  - workspace divided in independent tasks,
  - can be executed in parallel,
  - with no communication required among them.
- llCoMP [189], OMPICUDA [190], . . .
  - do not include automatic exploitation of GPU special features.

TuCCompi novelties

- **Tuning layer (T layer)**: Mechanisms to automatically choose optimal values for GPU configuration parameters.
    - based on provided programmer kernel characterization.
- **Concurrent kernel layer**: Automatic exploitation of modern GPU features as the *concurrent-kernel execution*.
- TuCCompi: programing model that combines the use of:
    - these two novel layers, and
    - the traditional ones (e.g., MPI, OpenMP, CUDA, ...).

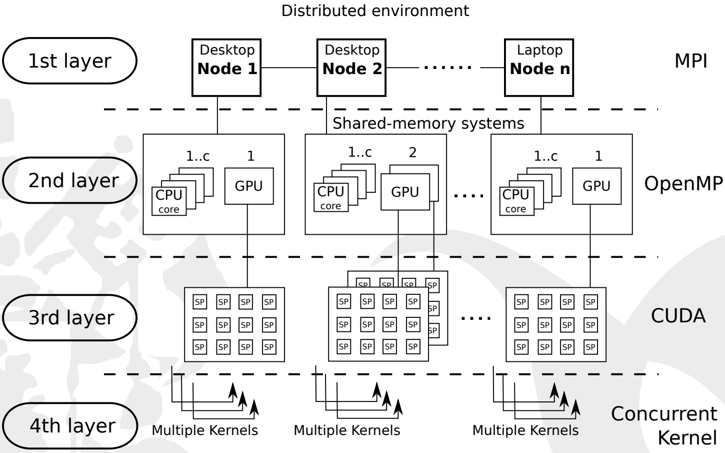
Heterogeneous distributed environment

Layer division of a heterogeneous cluster.

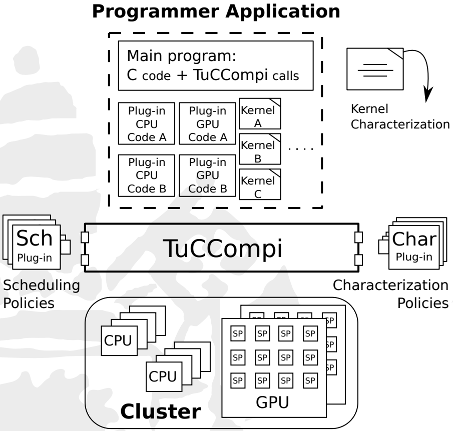


TuCCompi architecture

Layer deployment of TuCCompi model in a heterogeneous cluster.



TuCCompi Model Usage



1. Main program impl.
2. User-code Plugins.
   - plugin_Cpu
   - plugin_Gpu
3. Kernel characterization.

Custom usage:

- Workload scheduling.
- Characterization plugin.

The APSP as case study

- Embarrassingly parallel problem with *n* tasks ($n \times SSSP$).

CUDA SSSP algorithm

```
...
while(Δ ≠ ∞){
  <<<relax>>> (...)
  cudaDeviceSynchronize()
  Δ =<<<minimum>>> (...)
  cudaDeviceSynchronize()
  <<<update>>> (...)
  cudaDeviceSynchronize()
}
...
```

→

TuCCompi GPU plug-in

```
...
while(Δ ≠ ∞){
  TuCCompi_GPULAUNCH(relax, ...)
  TuCCompi_GPUSYN( )
  Δ = TuCCompi_GPULAUNCH(min, ...)
  TuCCompi_GPUSYN( )
  TuCCompi_GPULAUNCH(update, ...)
  TuCCompi_GPUSYN( )
}
...
```

- Kernel characterization:

```
TuCCompi_KERNELCHAR(relax, 1, scatter, low, high, low);
```

```
TuCCompi_KERNELCHAR(minimum, 1, coalesced, low, low, medium);
```

```
TuCCompi_KERNELCHAR(update, 1, coalesced, low, low, low);
```

Experimental scenarios

Experiment: TuCCompi's layer scalability solving the APSP problem.

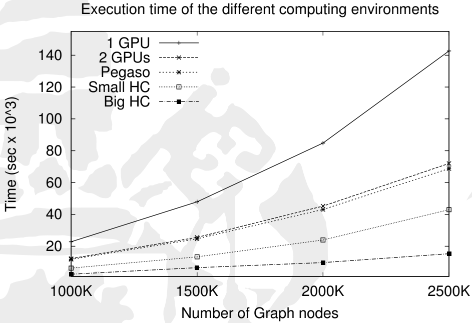
- **A single GPU** (3rd, 4th and T layers).
- **Two GPUs**, (2nd, 3rd, 4th and T layers).
- **Heterogeneous Node** (2nd, 3rd, 4th and T layers).
    - *Pegaso*: with 2 GPUs and 8 CPU-cores
- **Heterogeneous Clusters** (all layers).
    - ***Small HC*** 10 nodes: **4** GPUs and **48** asymmetric CPU-cores.
    - ***Big HC*** 19 nodes: **4** GPUs and **180** asymmetric CPU-cores.

Workload scheduling used: A `master-slave` policy.

Concurrent kernel execution set to 4.

Experimental Results

GPU vs. the heterogeneous environment:



Functionality:

- The addition of many less-powerful computational units enhances the total performance gain similarly as the addition of a GPU device.

Scalability:

- The execution times are reduced as more computational resources are used.
- The use of TuCCompi has a lower communication overhead less than the 1 %.

Summary

TuCCompi: A multilayer deployment model that helps the programmer to easily obtain flexible and portable programs for heterogeneous systems. It automatically detects at run-time the available computational resources and exploits hybrid clusters.

Offers to the programmer easy mechanisms to:

- Select proper values for GPU configuration parameters just characterizing the nature of the kernels.
- Exploit a concurrent-kernel execution.
- Deploy the solution using the traditional layers.
- Change the:
    - Algorithm that solves the problem.
    - Scheduling policy.
    - Characterization values.

Conclusions

Research question

- This PhD. Thesis answers the research question affirmatively.

It is possible to develop techniques and tools to derive efficient parallel implementations to solve Shortest-Path problems using:

(1) *The new modern Graphics Processor Units (GPUs) and their corresponding tuning techniques, and*

(2) *Heterogeneous environments composed by such hardware accelerators together with traditional CPUs.*

Thesis conclusions

- This Ph.D. Thesis gives an answer to these problems by providing:
  - a new improved GPU-based solution for the SSSP problem.
  - tuning heuristics to optimize GPU executions.
  - implementations and studies of productivity-based APSP approaches.
  - a multilayer programming model to ease the implementation and deployment of this kind of problems.

Contributions I

Study of state-of-the-art of both sequential and parallel shortest-path approaches for both SSSP and APSP problems:

1. **[Book]** H. Ortega-Arranz, D. R. Llanos, and A. Gonzalez-Escribano, "The Shortest Path Problem: Analysis and Comparison of Methods," Morgan & Claypool, 2014.
2. **[Survey article]** —. "Parallel Approaches to the Shortest Path Problem - A Survey," *In preparation*.

Development of a new GPU-based solution for the SSSP problem:

3. **[Journal article]** H. Ortega-Arranz, Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, "Comprehensive Evaluation of a New GPU-based Approach to the Shortest Path Problem," *International Journal of Parallel Programming*, 2015.
4. **[Conference article]** —. "A New GPU-based Approach to the Shortest Path Problem," in *Proc. of IEEE 11th International Conference HPCS*, 2013.

Contributions II

Kernel characterization model extension, and the development of parallel productivity approaches for the APSP problem:

5. **[Journal article]** H. Ortega-Arranz, Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, "Optimizing an APSP Implementation for NVIDIA GPUs Using Kernel Characterization Criteria", *The Journal of Supercomputing,* 2014.
6. **[Conference article]** —. "A Tuned, Concurrent-Kernel Approach to Speed Up the APSP Problem," in *Proc. of the 13th International Conf. CMMSE*, 2013.

Studies of heterogeneous productivity-based approaches for the APSP problem:

7. **[Book chapter]** —. "The All-Pair Shortest-Path Problem in Shared-Memory Heterogeneous Systems," in book *High-Performance Computing on Complex Environments,* John Wiley & Sons, Inc., 2014.

Contributions III

Development of a multilayer programming model for heterogeneous systems:

8. **[Journal article]** H. Ortega-Arranz, Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, "TuCCompi: A Multi-Layer Model for Distributed Heterogeneous Computing with Tuning Capabilities," *International Journal of Parallel Programming*, 2015.

9. **[Workshop article]** —. "TuCCompi: A Multi-Layer Programing Model for Heterogeneous Systems with Auto-Tuning Capabilities," in *Proc. of HLPGPU Workshop, HiPEAC*, 2014.

Future directions I

Shortest Path context

- Algorithmic modifications to take advantage of new GPU parallel capabilities, or for the emerging XeonPhi devices.

GPU Tuning context

- More model extensions using other graph characteristics.
- Adaptation of code analyzers to automatically obtain the kernel characterizations.

Future directions II

TuCCompi context

- Optional auto-tuning behavior for the concurrent kernel execution.
- Comparison against other libraries/frameworks specifically designed for particular problems or input sets.
- Addition of other functionalities provided by tools developed inside our research group, such as the data partition.

Thank you for your attention!

Parallel Approaches to Shortest-Path Problems for Multilevel Heterogeneous Computing

PhD. Dissertation

Héctor Ortega Arranz

Advisors: Dr. Diego R. Llanos Ferraris
Dr. Arturo González Escribano

October 15th, 2015

Grupo Trasgo
Universidad de Valladolid

Departamento de Informática
Universidad de Valladolid

Universidad de Valladolid