

# Squeezing GPU performance

GPGPU 2015: High Performance Computing with CUDA

University of Cape Town (South Africa), April, 20th-24th, 2015

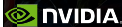


Manuel Ujaldón

Associate Professor @ Univ. of Malaga (Spain)

Conjoint Senior Lecturer @ Univ. of Newcastle (Australia)

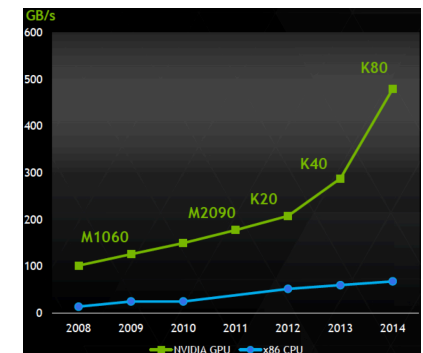
CUDA Fellow @ Nvidia



## GPU peak performance vs. CPU

Peak Double Precision FLOPS

Peak Memory Bandwidth



GPU 6x faster on "double":

GPU: 3000 GFLOPS

CPU: 500 GFLOPS

GPU 6x more bandwidth:

7 GHz x 48 bytes = 336 GB/s.

2 GHz x 32 bytes = 64 GB/s.



Manuel Ujaldon - Nvidia CUDA Fellow

19

## Let's make a Malaga - Madrid travel (500 km)

### Effective time using the train:

- Preliminaries: 3 minutes.
- Travel: 2 hours, 30 minutes.
- Closing: 2 minutes.
- TOTAL: 2 hours, 35 minutes.



### Effective time using the plane:

- Preliminaries: 90 minutes.
- Travel: 50 minutes.
- Closing: 30 minutes.
- TOTAL: 2 hours, 50 minutes (and you are away from downtown!)



## The real speed of my car

### Maximum:

- 250 km/h.

### Average on a 10 years use:

- 50 km/h.

So I regularly use my car at 20% of peak performance. Should I be disappointed?



Manuel Ujaldon - Nvidia CUDA Fellow

21



Manuel Ujaldon - Nvidia CUDA Fellow

20

# Instructions for the game available on the web site: <http://cms.ac.uma.es/kepler>

HOME CREDITS ACKNOWLEDGEMENTS

## CUDA challenge

Manuel Ujaldón @ NVIDIA

Font size [Bigger](#) [Reset](#) [Smaller](#)

### Instructions

- The game
- The input data set
- How to play
- Frequently asked questions

### Using GPUs

- At UMA
- In the cloud
- At home

### The quiz

- Participants
- Higher scores
- The winner strategy

### Welcome!

This practical tutorial provides you an easy way to interact with features of the GPUs based on the Kepler architecture, like:

- Ways to deploy massive parallelism via blocks, kernels and streams for Hyper-Q.
- Computing power (GFLOPS) and data bandwidth (GB/s).
- Latency for operands (int/float/double...) and operators (add/mul/div...).

You face different challenges here as CUDA programmer:

- Analyze performance for all the features described above, binding results to the GPU hardware for a better knowledge of its architecture.
- Investigate ways and mechanisms to get closer to the theoretical peak performance of a GPU.
- See how GFLOPS and bandwidth behave and are related to each other using the roofline model.

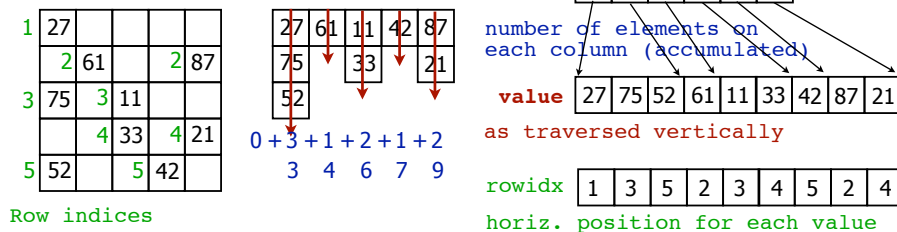
Enjoy the hands-on and... good luck with CUDA!

22

Manuel Ujaldón - Nvidia CUDA Fellow

# How you represent a sparse matrix in a Compressed Column Format

## Example for a 5x5 matrix:



## Given the data structure, this is how you traverse matrix:

```
for (i=0; i<N; i++)
  for (j=colptr[i]; j<colptr[i+1]; j++)
    value[j] += value[j];
```

24

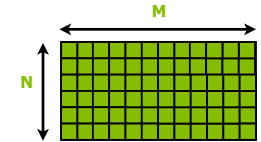
Manuel Ujaldón - Nvidia CUDA Fellow

# Forall loop execution versus data-dependent parallelism

## The simplest possible parallel program:

- Loops are parallelizable.
- Workload is known at compile-time.

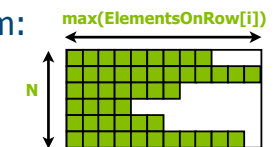
```
for (i=0; i<N; i++)
  for (j=0; j<M; j++)
    convolution (i, j);
```



## The simplest impossible parallel program:

- Workload is unknown at compile-time.
- The challenge is data partitioning.

```
for (i=0; i<N; i++)
  for (j=0; j<ElementsOnRow[i]; j++)
    convolution (i, j);
```



Poor solution #1: Oversubscription.  
Poor solution #2: Serialization.

23

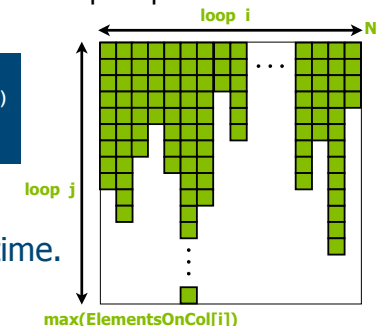
Manuel Ujaldón - Nvidia CUDA Fellow

# A challenge for CUDA programmers around the world: Performed on 8 countries so far

## What the program does: Iterate in parallel on each element of a sparse matrix compressed by columns.

- The sparse matrix may have N=100 or N=200 columns, each with a different number of nonzero elements. "numops" operations are performed on each element:

```
for (i=0; i<N; i++)
  for (j=colptr[i]; j<colptr[i+1]; j++)
    for (k=0; k<numops; k++)
      value[j] += value[j];
```



- All loops are fully parallel.
- Workload is unknown at compile-time.
- The challenge is data partitioning:
- Deploy streams, kernels, blocks and threads wisely.

25

Manuel Ujaldón - Nvidia CUDA Fellow

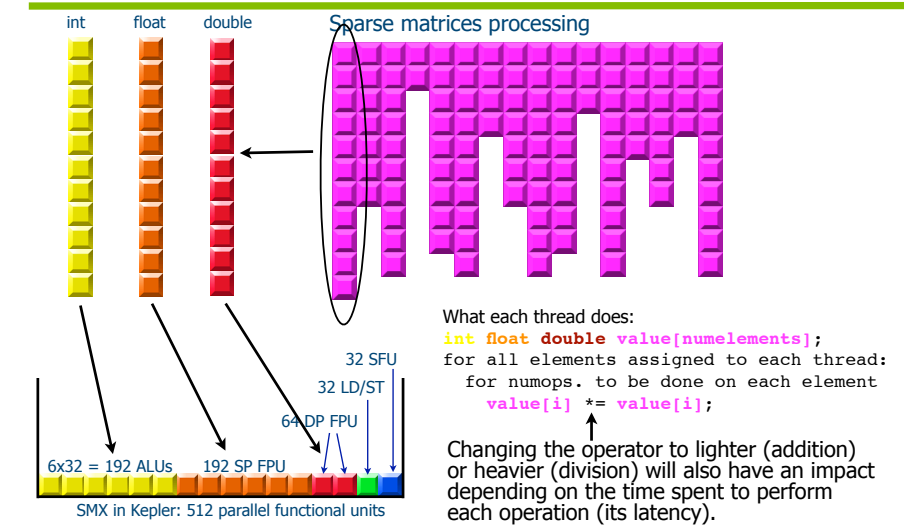
## Input sparse matrices (taken from the Matrix Market collection)

Application area	Matrix rows	Matrix columns	Nozeros	Workload
Economics	300	100	22.000	Base
Demography	6.000	100	440.000	20 x Base
Oceanography	24.000	100	1.760.000	160 x Base
Quantum physics	96.000	100	7.040.000	2560 x Base
Linear algebra	200	200	27.000	Base
Image processing	4.000	200	540.000	20 x Base
Astrophysics	32.000	200	4.320.000	160 x Base
Biochemistry	512.000	200	69.120.000	2560 x Base

26

Manuel Ujaldon - Nvidia CUDA Fellow

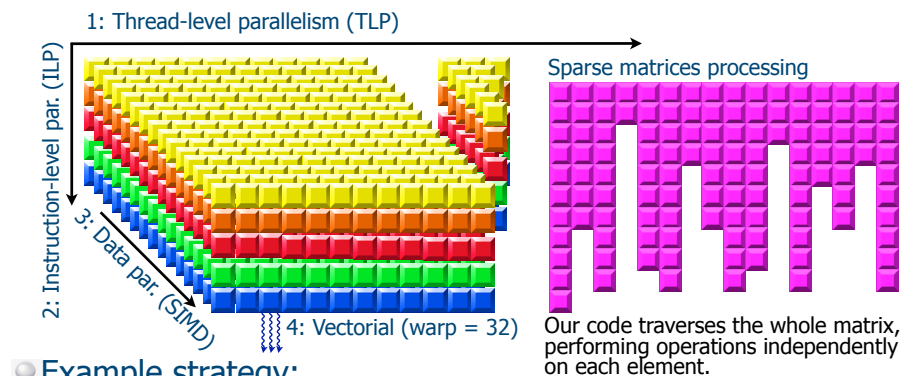
## You can try different operands and operators



27

Manuel Ujaldon - Nvidia CUDA Fellow

## And you have to choose the winner parallelization strategy



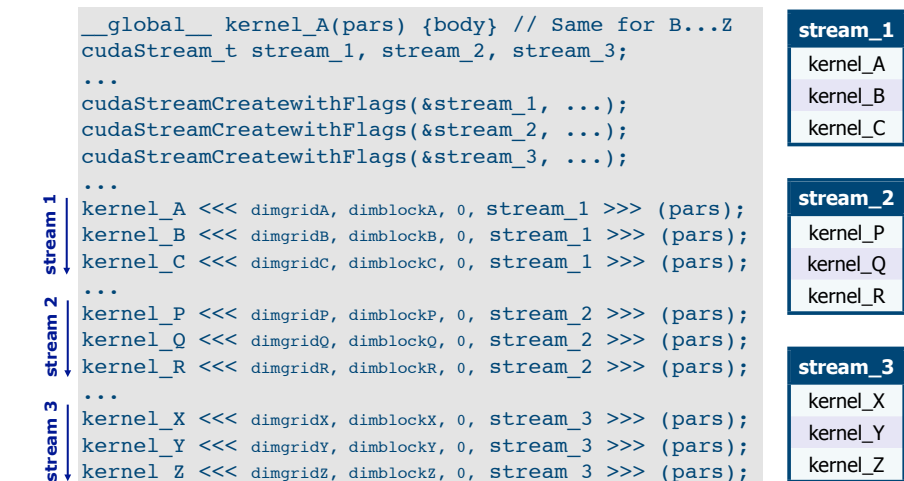
### Example strategy:

- We launch a CUDA kernel for each matrix column.
- Each kernel will have the lowest number of blocks.
- Each kernel will have the largest number of warps.
- Each thread will be as thin as possible (computes on a single elem.)

28

Manuel Ujaldon - Nvidia CUDA Fellow

## The way we create streams. An example of 3 streams, each composed of 3 kernels



29

Manuel Ujaldon - Nvidia CUDA Fellow

## Top 10 optimizations performed by students

1. Increase the number of operations per element (1024).
2. Increase the sparse matrix size (up to 69M nonzeros).
3. Change the operator (add/sub/mul/div).
4. Change the operand (int/float/double).
5. Tune the CUDA block size (384 threads per block).
6. Group blocks in kernels and those in streams to express more parallelism.
7. Optimize memory access using shared memory and regs.
8. Guide the compiler via `#pragma unroll` directives.
9. Enable the fused multiply-add operator.
10. Use vector instructions to exploit (x,y,z,w) and (r,g,b,a).

30

## Performance attained on a GeForce GTX480 (peak performance 1330 GFLOPS on 32-bit)

Optimization	Acceler.	Performance
Departure point		0.0008 GFLOPS
1. Increase the number of operations per element (up to 1024)	250.00 x	0.20 GFLOPS
2. Use a bigger sparse matrix (up to 69.120.000 nonzeros)	116.35 x	23.27 GFLOPS
3. Choose the sum operator (add)	1.00 x	23.27 GFLOPS
4. Replace the double operand (64-bits) by float (32-bit)	1.89 x	44.00 GFLOPS
5. Tune the block size (384 threads)	1.00 x	44.00 GFLOPS
6. Group kernels in streams	1.00 x	44.00 GFLOPS
7. Optimize memory accesses using shared memory and registers	3.19 x	140.75 GFLOPS
8. Unroll the loop via a <code>#pragma unroll</code> compiler directive	4.07 x	573.95 GFLOPS
9. Enable the FMADD (fused multiply-add) operator	2.15 x	1236.58 GFLOPS
10. Enable vector processing on computational sentences (4 in 1)	1.00 x	1236.58 GFLOPS
1.2 Saturate the number of operations (up to 1M)	1.02 x	1260.00 GFLOPS
8.2 Saturate the loop unroll factor (until 4096)	1.01 x	1280.00 GFLOPS
2.2 Generate a huge matrix to exploit GPU scalability	1.02 x	1310.00 GFLOPS
2.3 Tune the matrix to match the structure of CUDA parallelism	1.01 x	1330.00 GFLOPS

31