Squeezing GPU performance

GPGPU 2015: High Performance Computing with CUDA

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

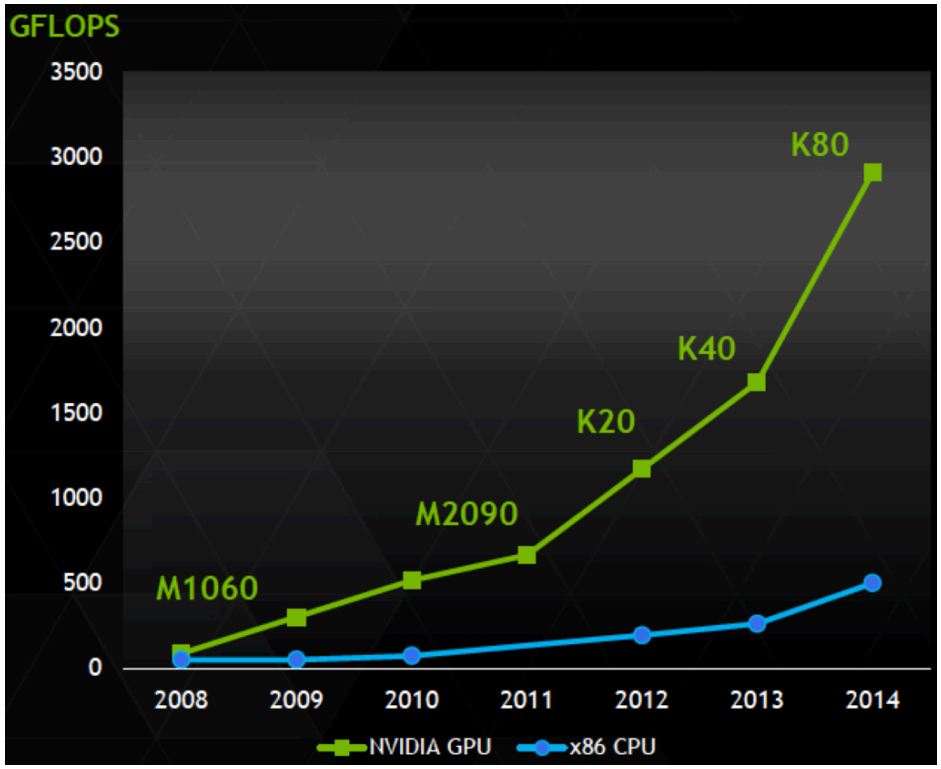
Manuel Ujaldón

Associate Professor @ Univ. of Malaga (Spain)
Conjoint Senior Lecturer @ Univ. of Newcastle (Australia)
CUDA Fellow @ Nvidia

NVIDIA

GPU peak performance vs. CPU

Peak Double Precision FLOPS



- GPU 6x faster on "`double`":
  - GPU: 3000 GFLOPS
  - CPU: 500 GFLOPS

Peak Memory Bandwidth

- GPU 6x more bandwidth:
  - 7 GHz x 48 bytes = 336 GB/s.
  - 2 GHz x 32 bytes = 64 GB/s.

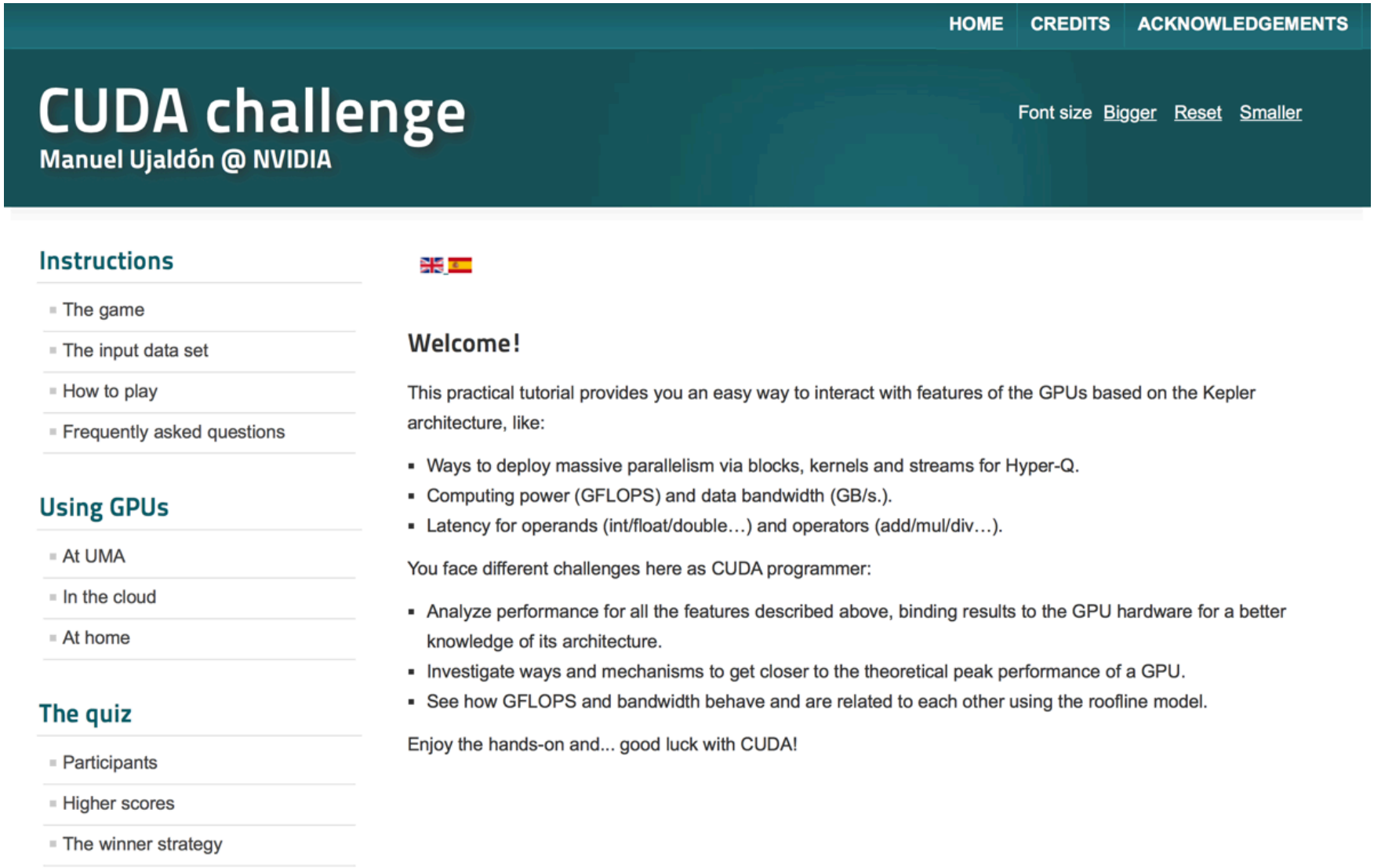
Let's make a Malaga - Madrid travel (500 km)

- 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?

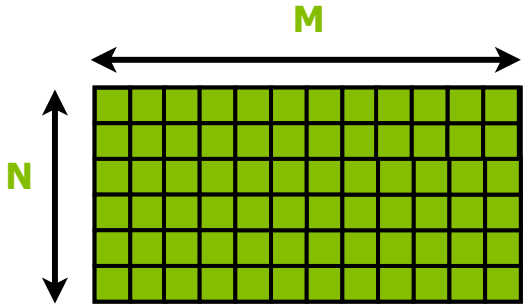
Instructions for the game available on the web site: http://cms.ac.uma.es/kepler



Forall loop execution versus data-dependent parallelism

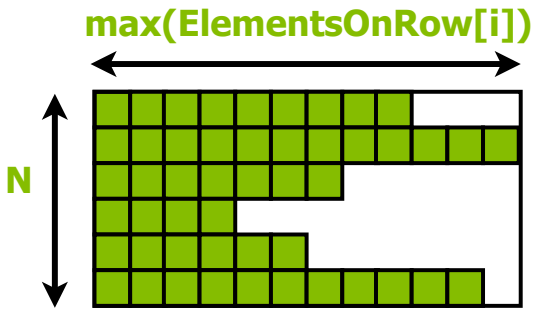
- The simplest possible parallel program:
  - Loops are parallelizable.
  - Workload is known at compile-time.

```
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);
```

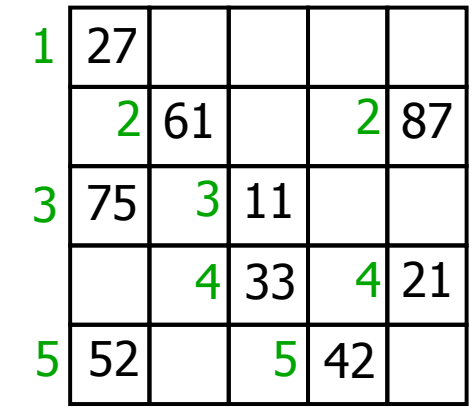


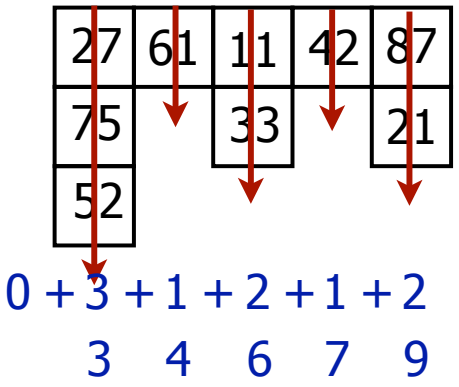
Poor solution #1: Oversubscription.

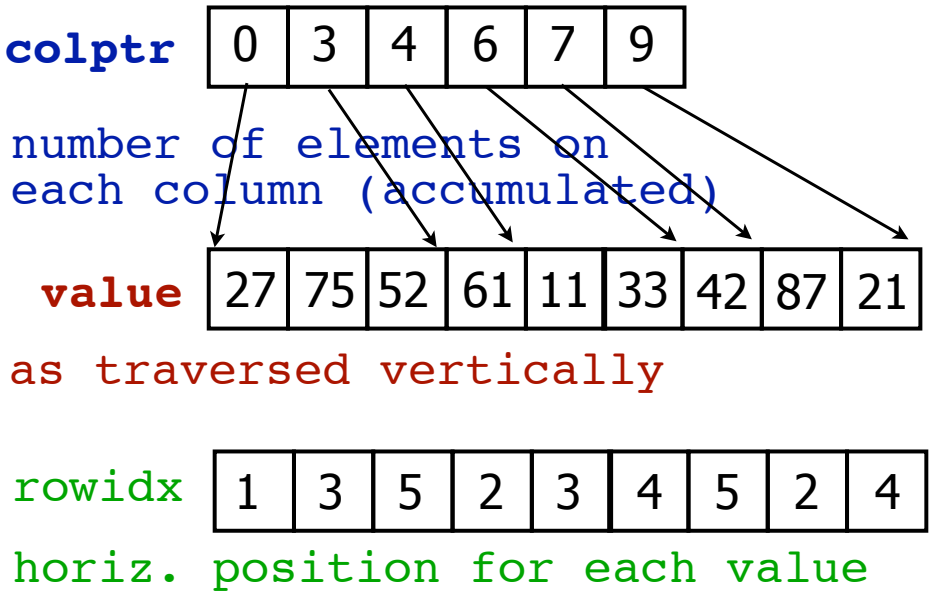
Poor solution #2: Serialization.

How you represent a sparse matrix in a Compressed Column Format

- Example for a 5x5 matrix:







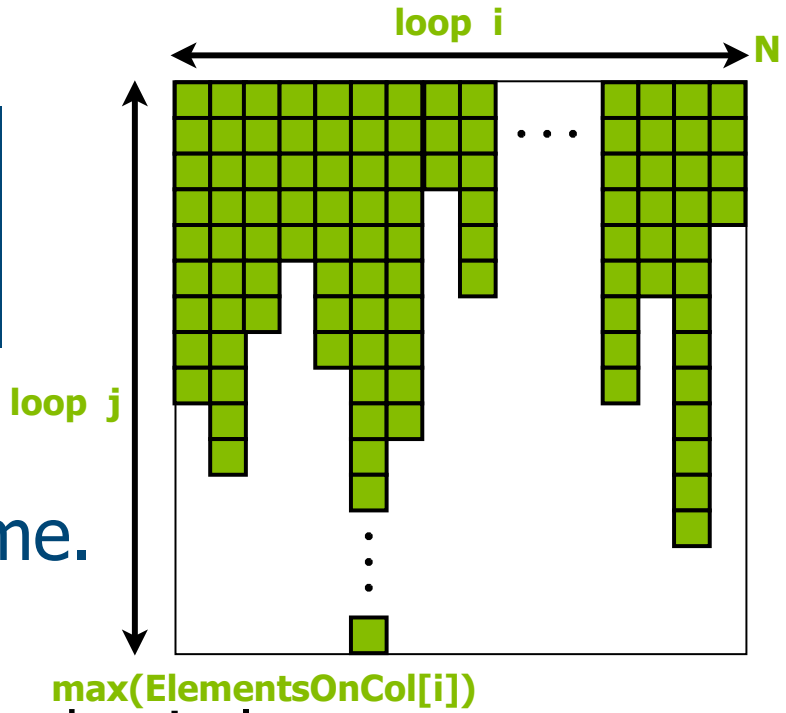
- Given the data structure, this is how you traverse matrix:

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

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];
```

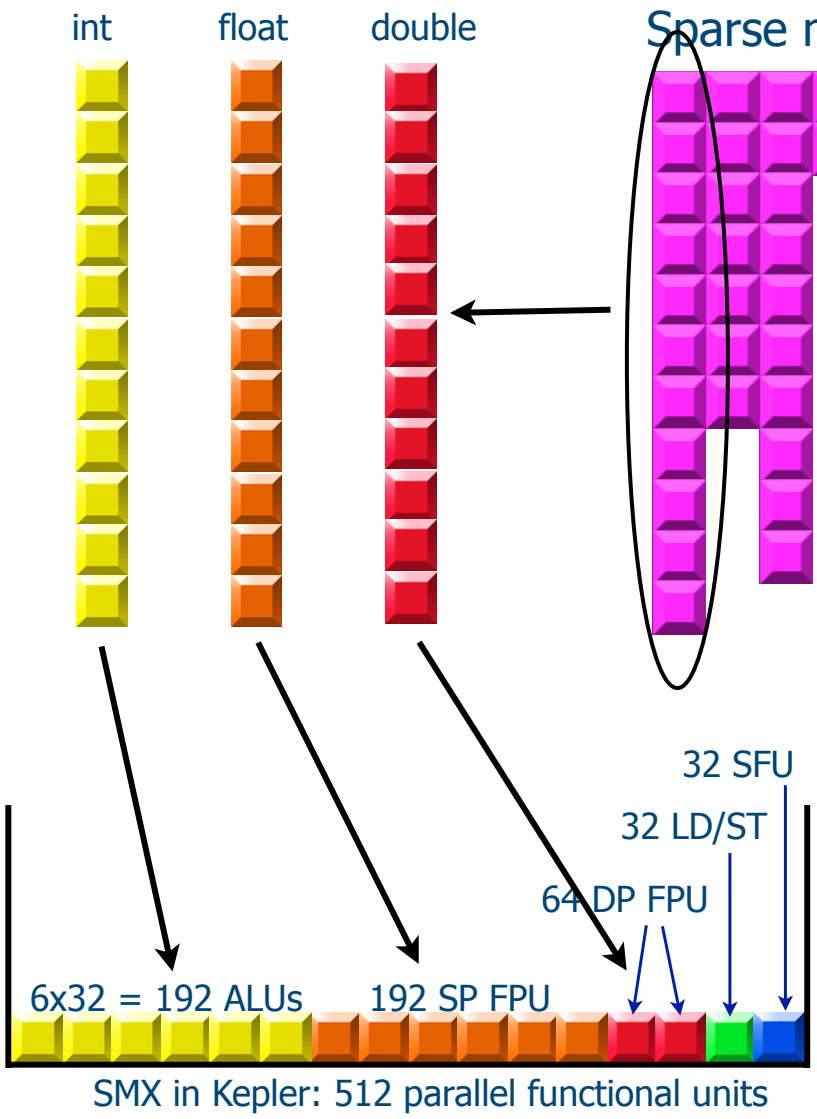


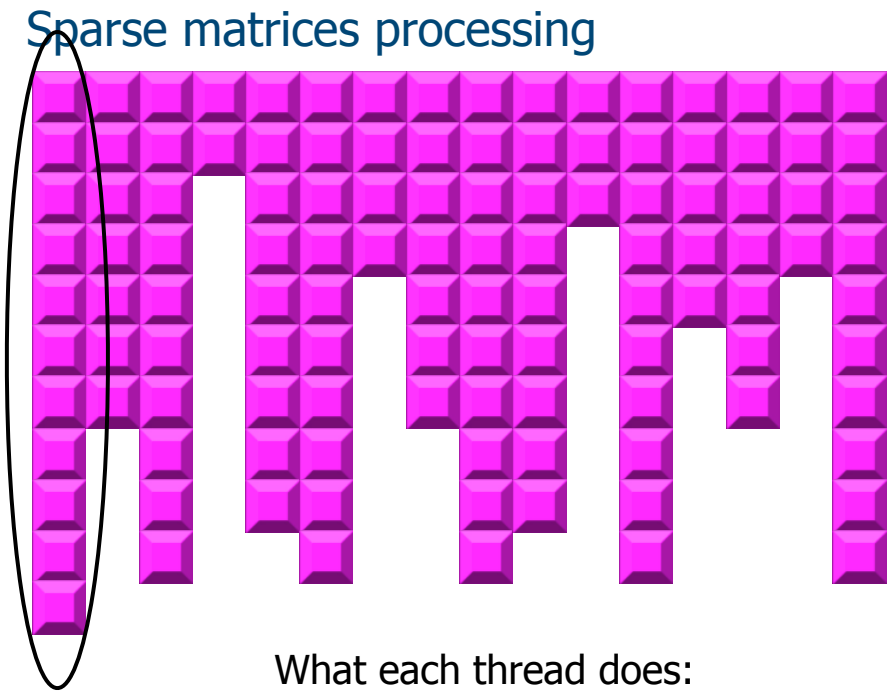
- All loops are fully parallel.
- Workload is unknown at compile-time.
- The challenge is data partitioning:
  - Deploy streams, kernels, blocks and threads wisely.

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

You can try different operands and operators



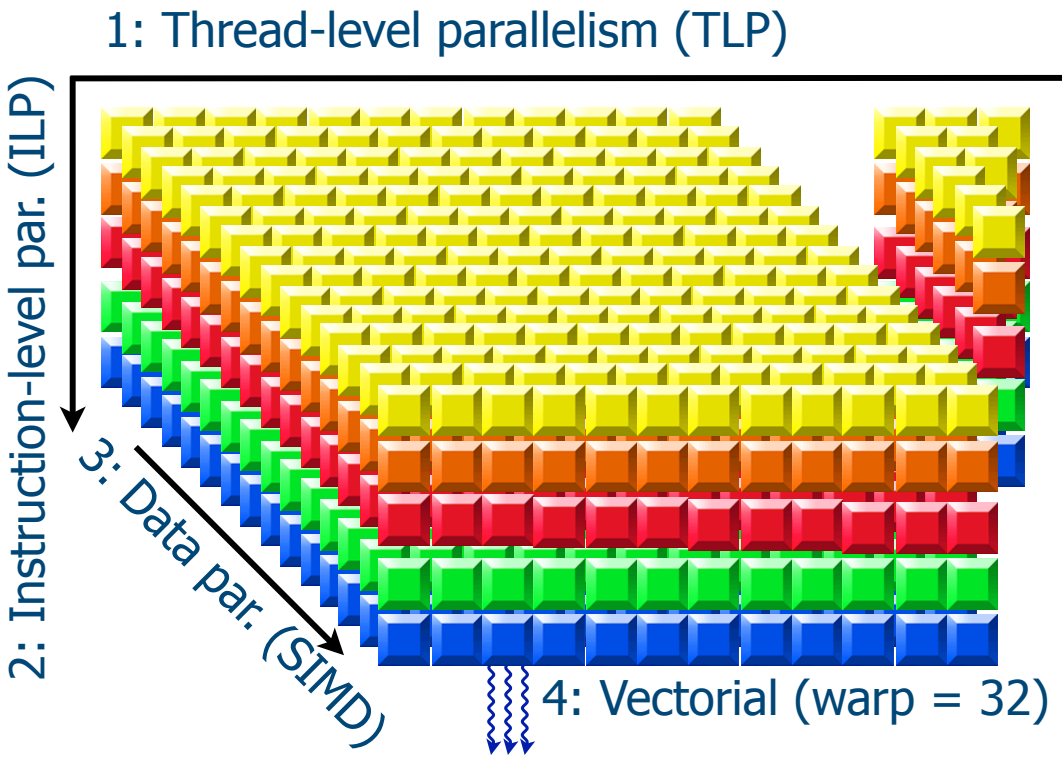


What each thread does:

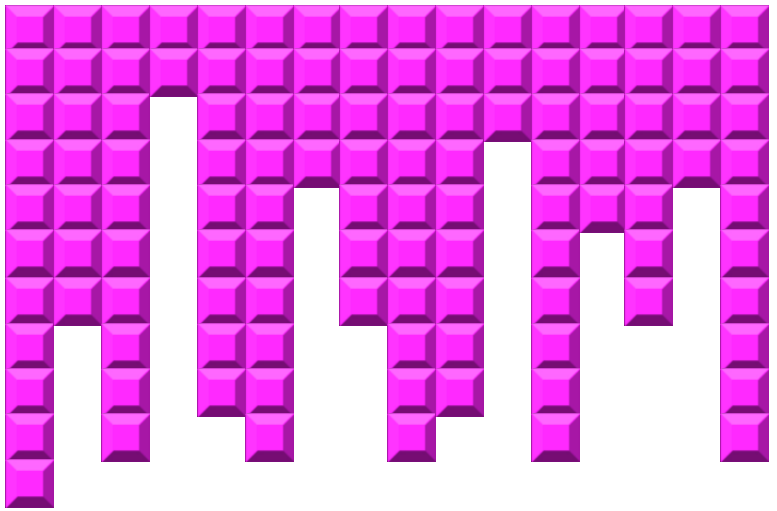
```
int float double value[numelements];
for all elements assigned to each thread:
  for numops. to be done on each element
    value[i] *= value[i];
```

Changing the operator to lighter (addition) or heavier (division) will also have an impact depending on the time spent to perform each operation (its latency).

And you have to choose the winner parallelization strategy



Sparse matrices processing



Our code traverses the whole matrix, performing operations independently on each element.

- 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.)

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

```
__global__ kernel_A(pars) {body} // Same for B...Z
cudaStream_t stream_1, stream_2, stream_3;
...
cudaStreamCreatewithFlags(&stream_1, ...);
cudaStreamCreatewithFlags(&stream_2, ...);
cudaStreamCreatewithFlags(&stream_3, ...);
...
kernel_A <<< dimgridA, dimblockA, 0, stream_1 >>> (pars);
kernel_B <<< dimgridB, dimblockB, 0, stream_1 >>> (pars);
kernel_C <<< dimgridC, dimblockC, 0, stream_1 >>> (pars);
...
kernel_P <<< dimgridP, dimblockP, 0, stream_2 >>> (pars);
kernel_Q <<< dimgridQ, dimblockQ, 0, stream_2 >>> (pars);
kernel_R <<< dimgridR, dimblockR, 0, stream_2 >>> (pars);
...
kernel_X <<< dimgridX, dimblockX, 0, stream_3 >>> (pars);
kernel_Y <<< dimgridY, dimblockY, 0, stream_3 >>> (pars);
kernel_Z <<< dimgridZ, dimblockZ, 0, stream_3 >>> (pars);
```

stream 1 ↓

stream 2 ↓

stream 3 ↓

stream_1
kernel_A
kernel_B
kernel_C

stream_2
kernel_P
kernel_Q
kernel_R

stream_3
kernel_X
kernel_Y
kernel_Z

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).

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 #pragma unroll 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