Pythia vs. GPU

Leif Lönnblad

Department of Physics
Lund University

Oxford, 2024-04-29

Preliminary GPU tests for PYTHIA8

Test case: Hadronic Rescattering

Complexity N^2

	without	with
pp 7 TeV	0.004 s/ev	0.006 s/ev

Preliminary GPU tests for PYTHIA8

Test case: Hadronic Rescattering

Complexity N^2

	without	with
pp 7 TeV	0.004 s/ev	0.006 s/ev
PbPb 2.76 TeV	0.246 s/ev	31.960 s/ev

Preliminary GPU tests for PYTHIA8

Test case: Hadronic Rescattering

Complexity N^2

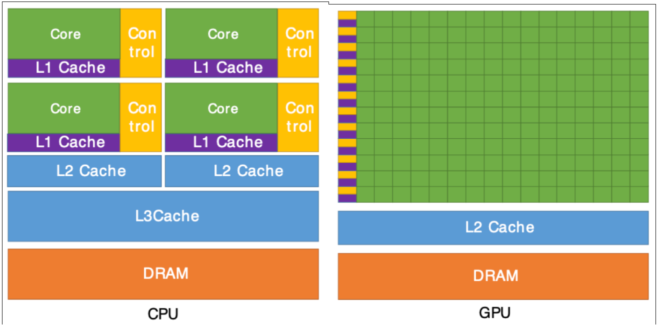
	without	with
pp 7 TeV	0.004 s/ev	0.006 s/ev
PbPb 2.76 TeV	0.246 s/ev	31.960 s/ev

	pp	PbPb
⟨ hadrons ⟩	60	3 900
⟨ pairs ⟩	3 400	15 000 000
⟨ tried ⟩	70	1 700 000
⟨ scattering ⟩	40	1 300

GPU vs. CPU

- GPU pros
  - Massively parallel
  - ?
- GPU cons
  - Can only do simple (numeric) tasks
  - Cannot do admin tasks
  - Only well localised data structures
  - Optimised for `float` not necessarily `double`
  - Communication CPU $\leftrightarrow$ GPU slow
  - A single GPU thread is slower than on the CPU

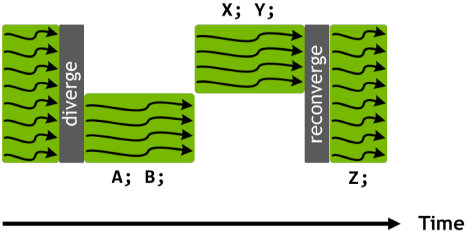
GPU vs. CPU

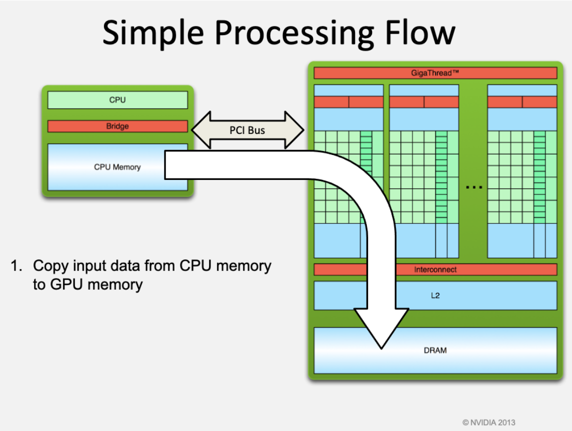


pictures stolen from Enrico Bothmann's tutotial

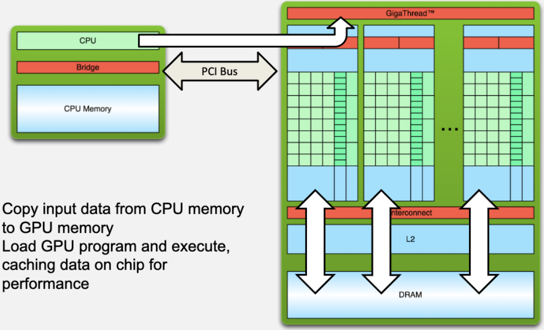
Branch Divergence

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



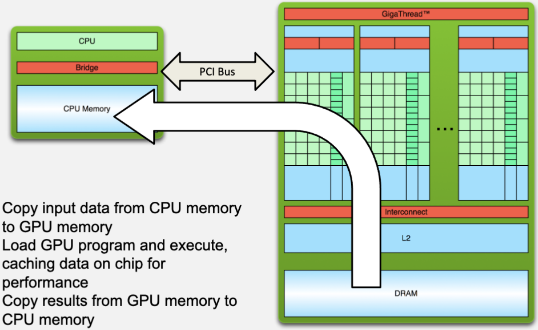

Simple Processing Flow
GigaThread™
CPU
Bridge
CPU Memory
PCI Bus
Interconnect
L2
DRAM
1. Copy input data from CPU memory to GPU memory
© NVIDIA 2013

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

What I did:

- Introduce `HadronRescattering` base class to plugin external models.
- Duplicated `Basics.h` to have `Vec4` and `RotBstMatrix` compiled for both CPU and GPU
- Reorganised the HR code to throw away possible scatterings early, checking only simple stuff, like closest approach. This we can do on the GPU.
- Put possible scatterings in a `priority_queue` (can also be done on the GPU).
- Go through the queue sequentially, do scatterings, add possible new scatterings to the queue.

The complexity is still formally N^2, but the initial step is parallelised, and we have no explicit loop that goes through the full $N(N-1)/2$ possible hadron pairs.

	without	with	reorg	+GPU
PbPb 2.76 TeV (s/ev)	0.3	32.0	14.8	8.2

	without	with	reorg	+GPU
PbPb 2.76 TeV (s/ev)	0.3	32.0	14.8	8.2
same but using float (s/ev)			13.2	5.3

	without	with	reorg	+GPU
PbPb 2.76 TeV (s/ev)	0.3	32.0	14.8	8.2
same but using float (s/ev)			13.2	5.3
A40 $\rightarrow$ A100				42.0
A100 using float				19.4

Caveats:

- Using the CUDA `thrust` library for ease of implementation, not necessarily optimal.
- Due to the cost of copying CPU $\leftrightarrow$ GPU, there is lower limit in size limit when GPU becomes useful. This has not been optimised.
- Running many jobs in parallel not benficial if there is only one GPU.

Tutorial

Simple main program based on `thrust`

- Generate a lot of random `Vec4` momenta
- Sum them
- Boost them to overall rest frame.
- Sum them again
- Calculate boosts to and from the rest frame of each pair.
- Measure time for all operations on CPU vs. GPU

thrust::

Based on vectors and algorithms with the look-and-feel of `std::`

```
thrust::host_vector<Vec4> hvec;
```

lives on the CPU

```
thrust::device_vector<Vec4> dvec;
```

lives on the CPU

Copy CPU to GPU:

```
dvec = hvec;
```

Copy GPU to CPU:

```
thrust::copy(dvec.begin(), dvec.end(), hvec.begin());
```

`thrust::` algorithms

We will use the following algorithms
(to avoid the rather awkward C-interface CUDA provides)

- `thrust::copy` to efficiently (?) copy CPU↔GPU
- `thrust::reduce` to sum momenta
- `thrust::transform` to boost momenta

There are others: `thrust::for_each`, `thrust::sort`, `thrust::remove_if`, ..., see `https://nvidia.github.io/cccl/thrust/`

There are no for-loops, but we can use `thrust::transform` (but nested loops difficult):

```
thrust::device_vector<Vec4> dmomenta;
// fill stuff ...
int N2 = dmomenta.size()*dmomenta.size();

thrust::device_vector<RotBstMatrix> dboosts(N2);
thrust::transform(thrust::make_counting_iterator((int)0),
                  thrust::make_counting_iterator(N2),
                  dboosts.begin(),
                  GetBoosts(dmomenta));
```

```
struct GetBoosts {

  long N;
  thrust::device_ptr<Vec4> dmomenta;

  GetBoosts(thrust::device_vector<Vec4> & dmomentaIn)
    : N(dmomentaIn.size()), dmomenta(dmomentaIn.data()) {}

  __device__
  RotBstMatrix operator()(int k) const {
    Vec4 & pi = *(dmomenta.get() + k%N);
    ...
  }

};
```

Let’s look at some code . . .

Logon to `sneezy` (or `sleepy`)

download the tutorial:

```
$ module purge
$ module load gcc/7.5/cuda/11.2
$ # (on sleepy use gcc/10.1.0 cuda/12.4)
$ wget http://home.thep.lu.se/~leif/misc/PythiaGPUTutorial.tgz
$ tar xzf PythiaGPUTutorial.tgz
$ cd PythiaGPUTutorial
$ make boost
$ ./boost
```