GPU Farm 1.0
Maria Chiara Cecconi
My GitHub directory: https://github.com/MCC04/GPUfarm

Results Summary

The code has been designed to compare performances of different CUDA implementations:

1. **Future:**
   the host code spawns K thread, each one of them
   - Cuda memory copy H2D
   - Launch the kernel
   - Cuda memory copy D2H
2. **Stream:**
   this technique tries to exploit data transfer time to hide kernel computation times, in this version of stream the code works as follows
   For each new non-default stream:
   - Cuda memory copy H2D, only a “slice” of the whole input data are transferred to device
   - Launch the kernel, working only on the slice assigned to the current stream
   - Cuda memory copy D2H
3. **Stream with managed memory:**
   This is analogous to the previous implementation, except for the memory management.
   In this case we don't need to explicitly transfer data to/from device. The unified memory will do this for us, so we only need to:
   - Launch the kernel, working only on the slice assigned to the current stream
4. **One SM:**
   This is a classical implementation of a kernel launch, forcing the device to use only one Streaming Multiprocessor. This will be useful, in performances analysis, as lowest parallel degree.

In the table below we give average time measures, w.r.t. number of host iterations

Host iters (#streams)	4	8	16	**Kernel iterations**
Elem Number	7168	14336	28672	
Future	0.966545	1.15176875	1.565461875	500
	1.88885	2.21480	2.983875	1000
Stream	3.7891675	8.69859	17.9969875	500
	7.4715925	17.1455874999	33.2975375	1000
Managed stream	3.8915525	8.71605	16.68546875	500
	7.5729125	17.050137499	33.15710625	1000
One SM	5908.82	13770.85	29508.70625	500

From those measures we can see that Future seems to have the better performances, in term of device completion time (measured using Events).
Streams and Managed Streams seem to have almost the same behavior: they require more time to complete all device operations w.r.t. Future code; furthermore the streams time doubles as the element number doubles. Future code, instead, seems to grow by a small factor.

Is this behavior due to a bad time sampling (because of some anomaly in events)?
The attempt to understand that is done by comparing event performance, with NvProf performances. Below there's a summary of what we get from NvProf (GPU activities).

FUTURE PROFILING ON 500 KERNEL ITERS:

4 host executions | 7168 elements

Time(%)	*Time*	*Calls*	*Avg*	*Name*
99.30%	4.8866ms	4	1.2217ms	cosKernel(int, int, float*, int, int*)
0.35%	17.344us	8	2.1680us	[CUDA memcpy DtoH]
0.35%	17.248us	4	4.3120us	[CUDA memcpy HtoD]

8 host executions | 14336 elements

Time(%)	*Time*	*Calls*	*Avg*	*Name*
98.64%	9.7824ms	8	1.2228ms	cosKernel(int, int, float*, int, int*)
0.75%	73.952us	16	4.6220us	[CUDA memcpy DtoH]
0.62%	61.120us	8	7.6400us	[CUDA memcpy HtoD]

16 host executions | 28672 elements

Time(%)	*Time*	*Calls*	*Avg*	*Name*
98.56%	23.945ms	16	1.4966ms	cosKernel(int, int, float*, int, int*)
0.74%	179.93us	16	11.245us	[CUDA memcpy HtoD]
0.70%	169.89us	32	5.3090us	[CUDA memcpy DtoH]

The highlighted values are those to compare with event measures, and those are relative to the device time to complete a kernel that works on all N elements.
As we can see, event measures are coherent with those of Nvidia profiler.

STREAMS PROFILING ON 500 KERNEL ITERS:

4 host executions | 7168 elements

Time(%)	*Time*	*Calls*	*Avg*	*Name*
99.62%	17.482ms	16	1.0926ms	cosKernel(int, int, float*, int, int*)
0.22%	39.008us	32	1.2190us	[CUDA memcpy DtoH]
0.15%	27.072us	16	1.6920us	[CUDA memcpy HtoD]

8 host executions | 14336 elements

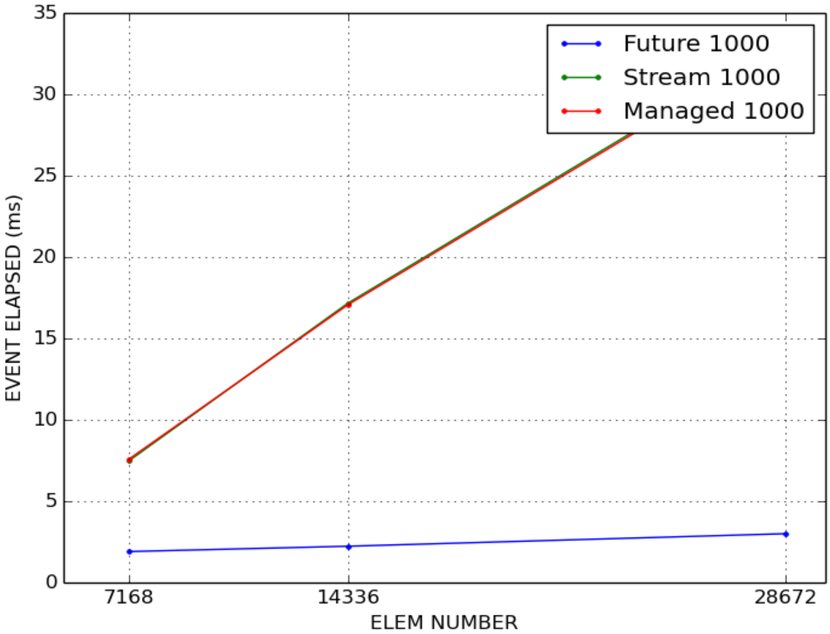
Time(%)	*Time*	*Calls*	*Avg*	*Name*
99.61%	69.914ms	64	1.0924ms	cosKernel(int, int, float*, int, int*)
0.24%	168.03us	128	1.3120us	[CUDA memcpy DtoH]
0.15%	107.52us	64	1.6800us	[CUDA memcpy HtoD]

16 host executions | 28672 elements

Time(%)	*Time*	*Calls*	*Avg*	*Name*
99.62%	279.72ms	256	1.0926ms	cosKernel(int, int, float*, int, int*)
0.23%	638.94us	512	1.2470us	[CUDA memcpy DtoH]
0.15%	432.44us	256	1.6890us	[CUDA memcpy HtoD]

The event times measures the time needed to complete the computation on all N elements by K streams.
So to compare the above highlighted times to event times, we should divide each by K (number of host executions).
Example: 69.91ms / 4 executions = 17,4775 ms that is the time to complete computation on all N elements by K streams.
So, as we can see from this comparison, event measures are almost good and precise.

<u>Graphic view of measured times</u>



Speedup Estimation

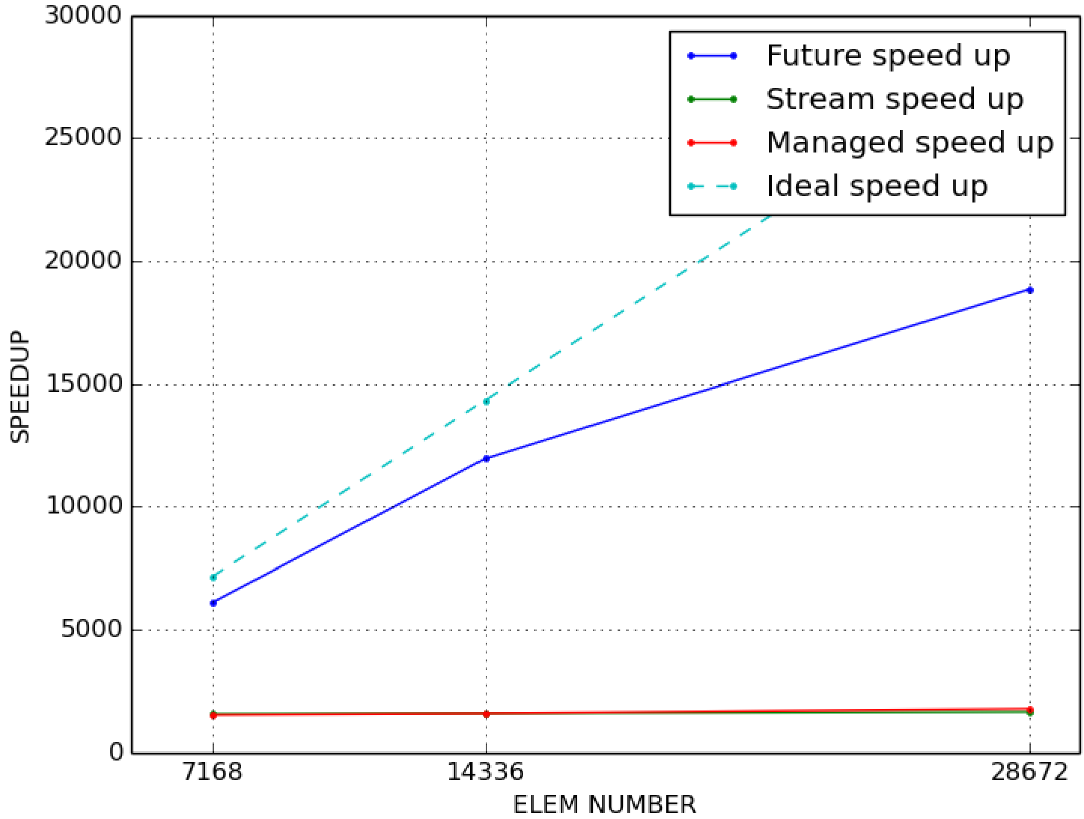
A speedup has been estimated for the 500 kernel iters version.
This has been estimated as follows:

$$Sp(n) = \frac{T_{oneSM}}{T_{n\,SM}}$$

- $T_{one\,SM}$ is the time collected using one SM, in other words we used as <<<GRID, BLOCK>>> dimensions <<<1, 32>>>.
  So we considered this as "sequential time", more precisely time for *parallelism degree = 1*.
  Note that effective degree is $deg_{effective}(1) = 32\,threads$

- $T_{n\,SM}$ is the time collected using more SMs, the number of SMs depends on GRID values, that are different for Stream or Future.
  This is the time using *parallelism degree = n* where
  - $deg_{effective}(n) = GRID * BLOCK$ for Future
  - $deg_{effective}(n) = GRID * BLOCK * num\,of\,Streams$ for Streams

Par. Degree	7168	14336	28672
Future Speedup	6113.34185165	11956.2629217	18849.84024284
Stream Speedup	1559.39794163	1583.11289531	1639.64698259
Managed Speedup	518.37088154	1579.94160199	1768.52725519



It's clear to see that the achieved speedup by Future is quite good, the speedup by Streams seems to be very bad (as expected from measured times).

Conclusions

It's clear that, currently, streams are not behaving good.
Watching at profiler output streams require to spend more time on memCpy (we copy little slices many times, in future we copy all in one time), this doesn't helps.
But the very increment in device time, between Future and Streams, is in kernel spent in Kernel execution.
I'm trying to discover whether it depends on a bad GRID/BLOCK tuning, or in the increment to kernel launch calls, or a mix of these two reasons.

Things I'm working on:

- Understand why Stream performances are bad, and so if it's possible to get better performances from Streams
- Better understand CUDA clock() to find kernel bottlenecks (useful for problem said above)
- Do tests on more different values of Kernel iterations (M) and number of elements (N)
- Get a more precise estimation on speedup and compute the maximum achievable speedup (Amdahl law, Gustafson's law)
- Understand how and if I can reduce page faults on Stream with unified Memory. Try to improve memory performances (those articles can be the key https://devblogs.nvidia.com/unified-memory-cuda-beginners/ https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels/)
- Better understand how to estimate other performance parameters as Bandwidth and GFLOPs (not sure if my effective BW is correctly computed)
- Will be useful to compare device code to pure host code? In other words, compare CPU time to GPU time (I already prepared a host sequential version of the code)

Most important articles I based performance and stream study:

https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/
https://devblogs.nvidia.com/how-implement-performance-metrics-cuda-cc/

Below all times measures (collected in milliseconds) are reported

Stream 500 Times:

- 4 iterations: 5.09091, 5.0513, 5.01446, 5.01296,
- 8 iterations:10.0012, 9.95654, 9.92013, 9.9456, 9.92099, 9.9192, 9.92506, 9.93642,
- 16 iterations: 20.0428, 19.957, 19.8752, 19.8451, 19.847, 19.869, 20.0177, 19.8944, 18.4479, 18.4646, 18.434, 18.4447, 18.4468, 18.2517, 18.1139, 18.1361]

Stream 1000 Times:
- 4 iterations: 9.96909, 9.95946, 9.95782, 9.95507,
- 8 iterations: 19.7634, 19.582, 19.5239, 19.5173, 19.625, 19.5273, 19.6258, 19.6776,
- 16 iterations: 35.6837, 35.6826, 35.6914, 35.6872, 35.6525, 35.541, 35.413, 35.3651, 35.3946, 35.4159, 35.404, 35.3932, 35.4774, 35.4826, 35.4764, 35.4771]

Future 500 Times:
- 4 iterations:1.30346, 1.28454, 1.27818, 1.27613,
- 8 iterations: 1.32042, 1.30128, 1.29888, 1.30099, 1.32947, 1.33325, 1.32986, 1.33034,
- 16 iterations: 1.6375, 1.60406, 1.69434, 1.67328, 1.6768, 1.6824, 1.68803, 1.69325, 1.6761, 1.66931, 1.67392, 1.66758, 1.67104, 1.67104, 1.66874, 1.67901]

Future 1000 Times:
- 4 iterations: 2.5313, 2.51344, 2.51066, 2.51162,
- 8 iterations: 2.52944, 2.51882, 2.51718, 2.52042, 2.53814, 2.54128, 2.55312, 2.54019,
- 16 iterations: 3.14013, 3.10979, 3.17635, 3.17446, 3.20419, 3.20592, 3.17392, 3.19014, 3.19654, 3.19123, 3.19891, 3.18266, 3.18298, 3.2047, 3.21008, 3.21306]

Managed 500 Times:
- 4 iterations: 5.44896, 5.07568, 5.04157, 5.07971,
- 8 iterations: 10.3638, 9.93866, 9.92342, 9.92614, 9.93341, 9.7967, 9.84627, 9.83379,4
- 16 iterations: 18.2522, 17.7132, 17.77, 17.75, 17.7564, 17.7571, 17.7874, 17.7399, 17.7931, 17.7683, 17.7725, 17.7855, 17.7534, 17.7645, 17.804, 17.7973]

Managed 1000 Times:
- 4 iterations:10.2927, 9.99565, 10.0033, 9.93853,
- 8 iterations: 19.7336, 19.4227, 19.4311, 19.4292, 19.4121, 19.4925, 19.4799, 19.4788,
- 16 iterations: 35.7487, 35.3643, 35.4599, 35.4684, 35.4714, 35.4619, 35.4569, 35.4437, 35.4469, 35.2812, 35.1972, 35.2068, 35.2336, 35.1409, 35.1319, 35.1337]

One SM 1000 Times:
- 4 iterations: 7893.6, 7870.84, 7870.84, 7870.83,
- 8 iterations: 15738.0, 15738.1, 15738.2, 15738.1, 15738.2, 15738.2, 15738.0, 15737.9,
- 16 iterations: 31476.2, 31476.1, 31476.2, 31476.1, 31475.9, 31476.0, 31476.1, 31475.8, 31475.8, 31475.9, 31475.9, 31475.8, 31475.8, 31475.8, 31475.9, 31475.9]