GPU Processing of Streaming Data: a CUDA Implementation of the FireHose Benchmark

Mauro Bisson*, Massimo Bernaschi† and Massimiliano Fatica*
*NVIDIA Corporation, †Italian Research Council

Identify valuable pieces of information spread across streaming data, is an interesting problem in several fields, from advertisement placement to cyber-security to name just a few. Modern communication systems have reached a scale and a capillarity such that it makes challenging to process the data they generate. The amount of data streamed through public/social networks per unit of time is so large that it is not feasible to use a store-and-process approach for their analysis. Instead, it is necessary to resort to a real-time, *on-the-fly*, processing of the continuously flowing stream of data. Given the (ever increasing) volume of streaming data, a timely processing imposes a number of constraints: the processing of each datum should require a limited processing effort in order to be ready to process the next one as soon as possible; in order to enable the concurrent tracking of the maximum number of datums, the state information kept for each one should be as limited as possible; storage systems bandwidth and capacity are usually too low to keep up with the data rate so each datum should be processed once, at receive time.

The main purpose of the FireHose benchmark [1] is to enable comparison of streaming software and hardware, both quantitatively vis-a-vis the rate at which they can process data, and qualitatively by judging the effort involved to implement and run the benchmarks. The benchmarks defined in the FireHose suite involve best-effort processing of UDP packets arriving at high rates, capturing the typical task of network monitoring.

In this paper we will discuss our implementation of the current FireHose benchmarks for GPU systems and report its results.

I. FireHose Benchmark

There are currently three benchmarks defined in FireHose, each one comprised of two parts: a front-end generator, that produce UDP packets containing datums, and a back-end analytic engine that receives and process the datums. Packets have a text-based format containing one datum per line. The contents of the datums depends on the benchmark. The generators are meant to be used as-is, without modifications, while the analytic engine must be implemented in code optimized for the target architectures.

The benchmark requires analytic engines to receive generators' packets through datagram sockets (since packets are sent according to the UDP protocol) but does not make any assumption about the communication network (loopback device, Internet, Ethernet, Infiniband, etc.). The usage of datagram sockets prevents the generator from being throttled by the analytic engine, just as in stream processing scenarios where data must be processed as it appears in real-time. Due to the UDP features, the analytic engine may not receive all data if its hosting system cannot keep up with the generated stream, which is one of the effects the benchmark wish to measure.

A. Power-law

The first benchmark is composed of a generator called `powerlaw` and of an analytic called `anomaly1`. The generator produces datums that contain three fields:

$$< key, value, truth >$$

The *key* is a 64-bit unsigned integer generated randomly from a static range according to a power-law distribution. Keys are divided in two categories, *biased* and *unbiased*, generated with a ratio of 1:255. The *value* is an integer assuming value either 0 or 1 at random and it is associated to the specific key occurrence in its datum. The *truth* is a binary flag which specifies whether the key is biased or not. For unbiased keys, value 0 *vs.* 1 is chosen with equal probability whereas, for biased keys, 0 is selected with probability $15/16$. The truth field is the same for every occurrence of the same key.

The goal of the analytic is to identify biased keys. When an individual key has been seen 24 times, a decision for the key is taken based on its values. If a 1 appeared four times or less then it is labeled as biased; otherwise it is labeled unbiased. The decision is checked against the truth value in order to count the identification event as one of the following:

- **true anomaly**: key correctly flagged as biased;
- **false positive**: key incorrectly flagged as biased;
- **false negative**: key incorrectly flagged as unbiased;
- **true negative**: key correctly flagged as unbiased.

B. Active

This benchmark is similar to the power-law anomaly detection with the exception that the `active` generator produces keys chosen in a continuously evolving "active set" (of size 128K keys) rather than in a fixed range. Each key is generated a limited number of times, then removed from the active set and replaced by a new key. In this way, the number of generated keys increases with the running time of the generator. Keys are emitted according to a "trending" effect where a key appears occasionally, then appears more frequently, then tapers off.

C. Two-level

The last benchmark also involves anomaly detection but, in this case, the keys to be analyzed for bias are generated according to a two-level scheme working in the following way. The generator produce datums containing so-called *outer* keys the same way as keys produced by the active generator. In this case, however, instead of the value and truth fields, each datum contains a 16-bit unsigned integer as the second field, and a sequence number as the third. Each outer key is emitted at most five times and, combining the second fields from the complete sequence, an *inner* key with a value and a truth fields are devised. Each occurrence of an inner key is determined by five distinct outer keys. The analytics must keep track of the outer keys to devise the inner keys that will be processed for anomaly detection.

II. Results

Preliminary results of our CUDA [2] implementation of the FireHose suite are presented in the following table. For each anomaly, we report the result of the reference code with the highest number of generators as reported on the FireHose website (dark gray) on a desktop machine running RedHat Linux with dual hex-core 3.47 GHz Intel Xeon (X5690) CPUs, and a number of results on Tesla M40 GPUs. The generator(s) and the analytic were both run together on the same box. with at least that many generators (light gray). The score column is an absolute number counting the differences in each category of events with respect to the serial reference implementation run on the same data produced by the generators. Dividing the score by the total number of datums (column 4) gives an estimate of the actual percentage of misidentified events.

Version	Machine	Ngen	Ndata	Grate	Prate	Packets	Events	Score	Instant	LOC
					Anomaly1					
PH/C++ 142	Intel Xeon X5690	4	3.0 B	10 M/sec	—	100%	230 K	0	yes	525
CUDA	4x Tesla M40	4	4.6 B	12.8 M/sec	175 M/sec	100%	334 K	4	yes	1172
CUDA	4x Tesla M40	32	36.8 B	57.2 M/sec	576 M/sec	100%	2.7 M	18	yes	1172
					Anomaly2					
PH/C++ 142	Intel Xeon X5690	2	6.1 B	3.4 M/sec	—	100%	6.80 M	10	yes	665
CUDA	2x Tesla M40	2	12.2 B	6.1 M/sec	61 M/sec	100%	12.20 M	2431	yes	1361
CUDA	4x Tesla M40	8	49.0 B	23.4 M/sec	121 M/sec	100%	48.80 M	t.b.d.	yes	1361
CUDA	4x Tesla M40	32	194.6 B	44.2 M/sec	122 M/sec	100%	194 M	t.b.d.	yes	1361
					Anomaly3					
PH/C++ 142	Intel Xeon X5690	1	2.7 B	1.5 M/sec	—	100%	1.12 M	0	yes	495
CUDA	1x Tesla M40	1	3.6 B	2.5 M/sec	22 M/sec	100%	1.46 M	132	yes	1621
CUDA	2x Tesla M40	2	7.2 B	4.8 M/sec	43 M/sec	100%	2.92 M	t.b.d.	yes	1621
CUDA	4x Tesla M40	8	29.0 B	17.4 M/sec	86 M/sec	100%	11.65 M	t.b.d.	yes	1621
CUDA	4x Tesla M40	32	116.3 B	33.9 M/sec	85 M/sec	100%	43.07 M	t.b.d.	yes	1621

Version: which version of the analytic was run.
Machine: what machine the benchmark was run on.
Ngen: number of generators used.
Ndata: number of datums generated.
Grate: aggregate stream rate for all generators, in datums/sec.
Prate: aggregate processing rate for all GPUs, in datums/sec.
Packets: percentage of packets or datums processed (100.0 means no drops).
Events: total events for the analytic to detect.
Score: score for the analytic calculation (0 means perfect).
Instant: no/yes for whether the analytic produces its answers instantly when a datum triggers a result.
LOC: lines-of-code required to implement the analytic.

The final paper will present the details of the CUDA implementation and system tuning required to process this large amount of data.

References

[1] The FireHose Benchmark, http://firehose.sandia.gov/doc/README.html
[2] CUDA Toolkit, *http://developer.nvidia.com/cuda-toolkit*