Plotting data in a GPU with Histogrammar

Jim Pivarski

Princeton – DIANA

September 30, 2016

Problem:

You're developing a numerical algorithm for GPUs (tracking).

If you were doing it on a CPU, you could visualize the distribution of any intermediate quantity to debug development.

But a GPU is like a ship in a bottle: hard to get data in and out; can't use standard tools (ROOT TH1) in `__global__` or `__device__` functions.

Solution:

Use my library (next page).

How to plot a `__device__` quantity (call it "`residual`"):

1. Define a histogram or suite of histograms (Python).

```
h = Bin(100, -5.0, 5.0, "residual")
```

2. Generate CUDA code and include it in your project.

```
open("histogram.cu", "w").write(h.cuda())
```

3. Allocate histograms in your code (CUDA).

```
const int numHists = 1024;
Aggregator* hists;
cudaMalloc((void**)&hists, numHists * sizeof(Aggregator));
initialize<<<1, numThreadsPerBlock>>>(hists, numHists);
```

4. Fill them in your `__global__` or `__device__` function.

```
__global__
void yourFunc(Aggregator* hists, int numHists, ...) {
  float residual = ...;
  fill(hists, numHists, residual);
}
```

5. When you're done, collapse `numHists` histograms into one histogram and bring it back to the CPU.

```
Aggregator* resultGPU;
cudaMalloc((void**)&resultGPU, sizeof(Aggregator));
extract<<<1, numThreadsPerBlock>>>(hists, numHists, resultGPU);

// (or use pinned memory to avoid explicit copy).
Aggregator resultCPU;
cudaMemcpy(&resultCPU, resultGPU, sizeof(Aggregator),
           cudaMemcpyDeviceToHost);
```

6. Print out the histogram data.

```
toJson(&resultCPU, stdout);   // or a FILE* to "h.json"
```

7. Plot it in PyROOT (Python).

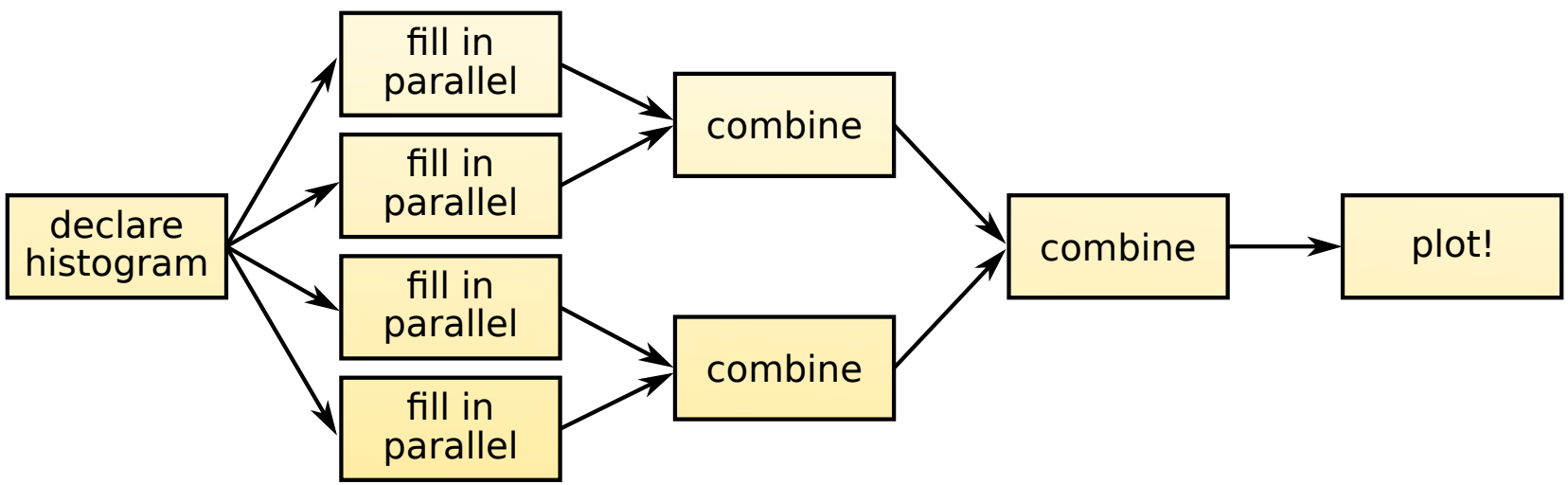
```
h = Factory.fromJson(open("h.json").read())
roothist = h.plot.root(name="resid")
roothist.Draw()
```

What just happened?

Generates C structs and fill functions specific to your histogram, making no assumptions about number of threads or blocks.

Whenever one of your kernels calls `fill`, a thread-local histogram is filled. They may be allocated in global memory (this example) or shared memory (more complex).

The `extract` kernel recursively adds the thread-local histograms into one final histogram (e.g. 1024 histograms in 10 steps).



Why not roll your own?

- "Mappers" (embarrassingly parallel) are easy on GPUs, "reducers" are hard because of coordination among threads. Good to have a library.
- Number of histograms might be different from number of threads. Generated code uses `%` and `atomicAdd`.

  (Perhaps that should be optional?)
- Contiguous in memory for efficient cache usage.
- Many types of histograms: regular, irregular, N-dimensional, profiles, stacked, efficiency ratio, directories of histograms, etc.

  Actually, a generic grammar for creating new aggregator types.
- Part of an ecosystem involving much more than GPUs.

histo·*grammar*

/histō¦ˈgɹæm.ər/

Standard histograms:

```
Bin(num, low, high, fillRule, Count())
```

Two-dimensional histograms:

```
Bin(xnum, xlow, xhigh, "x",
  Bin(ynum, ylow, yhigh, "y",
    Count()))
```

Profile plots:

```
Bin(xnum, xlow, xhigh, "y",
  Deviate("y"))

Bin(xnum, xlow, xhigh, "x",
  Bin(ynum, ylow, yhigh, "y",
    Average("z")))
```

Complex trees of aggregations:

```
Bin(xnum, xlow, xhigh, "x",
  Branch(Minimize("y"),
         Maximize("y"),
         Average("y"),
         Sum("weight"),
         Sum("weight * weight")))
```

Mix and match binning methods:

```
IrregularlyBin([-2.4, -2.1, -1.5,
    0.0, 1.5, 2.1, 2.4], "eta",
  Bin(314, -3.14, 3.14, "phi",
    Count()))

SparselyBin(0.01, "eta",
  Bin(314, -3.14, 3.14, "phi",
    Count()))

Categorize(fillByName,
  Bin(314, -3.14, 3.14, "phi",
    Count()))
```

Directories of histograms:

```
Label(hx = Bin(num, low, high, "x"),
      hy = Bin(num, low, high, "y"),
      hz = Bin(num, low, high, "z"))
```

High-level interface to common patterns:

```
Fraction("numHits > 5",
  Bin(numBins, lowEdge, highEdge, "pt"))

Stack([5.0, 10.0, 30.0, 50.0, 100.0], "pt",
  Bin(numBins, lowEdge, highEdge, "numHits"))
```

To be independent of data partitioning, aggregators must

be **additive** (independent of *whether* data are partitioned):

$$\text{fill}(\text{data}_1 + \text{data}_2) = \text{fill}(\text{data}_1) + \text{fill}(\text{data}_2)$$

be **homogeneous** in the weights:

$$\text{fill}(\text{data}, \text{weight}) = \text{fill}(\text{data}) \cdot \text{weight}$$

be **associative** (independent of *where* data are partitioned):

$$(h_1 + h_2) + h_3 = h_1 + (h_2 + h_3)$$

have an **identity** (for both $+$ and fill):

$$h + 0 = h, \quad 0 + h = h, \quad \text{fill}(\text{data}, 0) = 0$$

Mathematical basis

To be independent of data partitioning, aggregators must

be **additive** (independent of *whether* data are partitioned):

$$\text{fill}(\text{data}_1 + \text{data}_2) = \text{fill}(\text{data}_1) + \text{fill}(\text{data}_2)$$

be **homogeneous** in the weights:

$$\text{fill}(\text{data}, \text{weight}) = \text{fill}(\text{data}) \cdot \text{weight}$$

} linear

be **associative** (independent of *where* data are partitioned):

$$(h_1 + h_2) + h_3 = h_1 + (h_2 + h_3)$$

have an **identity** (for both $+$ and fill):

$$h + 0 = h, \quad 0 + h = h, \quad \text{fill}(\text{data}, 0) = 0$$

} monoid

Aggregation routines in Histogrammar are spanned by a generator set of linear monoids.

Multi-lingual implementation

Aggregator	Scala	JVM-JIT	Python	Numpy	ROOT	GPU	C++11	Julia	R	Javascript
Count	DONE		DONE	DONE	DONE	DONE	trial	DONE		
Sum	DONE		DONE	DONE	DONE	DONE	trial	DONE		
Average	DONE		DONE	DONE	DONE	DONE		DONE		
Deviate	DONE		DONE	DONE	DONE	DONE		DONE		
Minimize	DONE		DONE	DONE	DONE	DONE		DONE		
Maximize	DONE		DONE	DONE	DONE	DONE		DONE		
Bag	DONE		DONE	DONE	DONE	growable?		DONE		
Bin	DONE		DONE	DONE	DONE	DONE	trial	DONE		
SparselyBin	DONE		DONE	DONE	DONE	growable?				
CentrallyBin	DONE		DONE	DONE	DONE	DONE				
IrregularlyBin	DONE		DONE	DONE	DONE	DONE				
Categorize	DONE		DONE	DONE	DONE	growable?				
Fraction	DONE		DONE	DONE	DONE	DONE				
Stack	DONE		DONE	DONE	DONE	DONE				
Select	DONE		DONE	DONE	DONE	DONE	trial			
Label	DONE		DONE	DONE	DONE	DONE				
UntypedLabel	DONE		DONE	DONE	DONE	DONE				
Index	DONE		DONE	DONE	DONE	DONE				
Branch	DONE		DONE	DONE	DONE	DONE				

The "Bag" specification (multiset for scatter-plots) may be revised soon.

The C++ implementation *will* be revised.

Three aggregators haven't been implemented for the GPU because they require a growable memory allocation. Have to think about a good way to do that...

Tentacles in many different frameworks

diana hep

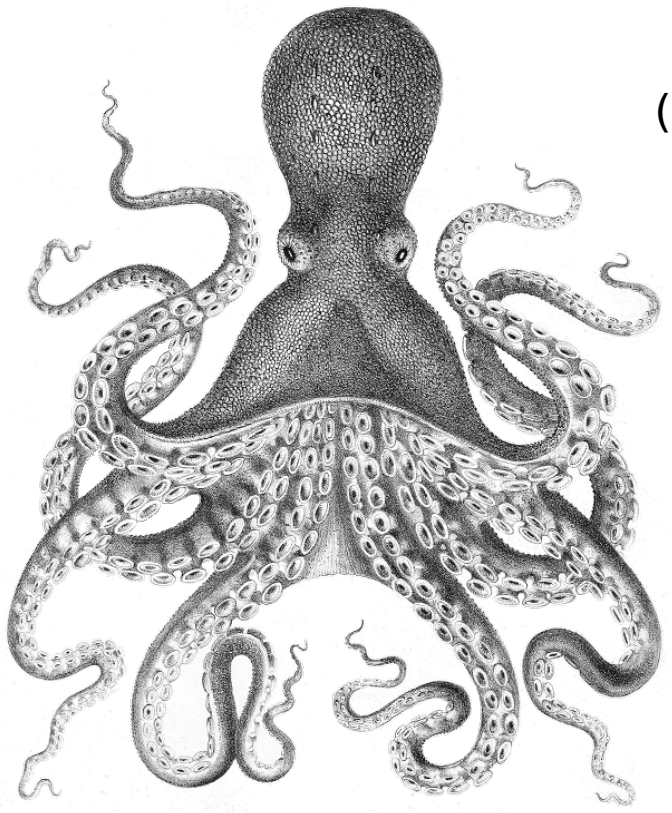
Back-ends (filling)

Scala/Spark
and SparkSQL

Python and
Numpy

ROOT with
JIT optimization

CUDA and
PyCUDA



Front-ends (plotting)

Bokeh
(Scala and Python)

Matplotlib
(Python)

ROOT
(Python and C++)

Vega (future:
all systems)

Intention: Histogrammar should act as a *bridge* between frameworks.

Work to be done

- Help you get started using Histogrammar for your work.
- Learn from you what a more convenient interface is.
  - Command-line script?
  - OpenCL?
- Fix bugs and respond to feature requests.

http://histogrammar.org

Generated code

```
// Auto-generated on 2016-09-29 11:12:23
// If you edit this file, it will be hard to swap it out for another auto-generated copy.

#ifndef HISTOGRAMMARCUDA_0
#define HISTOGRAMMARCUDA_0

#include <stdio.h>
#include <math_constants.h>

namespace HistogrammarCUDA_0 {
  // How the aggregator is laid out in memory (CPU main memory and GPU shared memory).

  typedef struct {
    float entries;
    float underflow;
    float overflow;
    float nanflow;
    float values[100];
  } Bn100CtCtCtCt;

  typedef Bn100CtCtCtCt Aggregator;


  // Specific logic of how to zero out the aggregator.
  __device__ void zero(Aggregator* aggregator) {
    int bin_0;
    (*aggregator).entries = 0.0f;
      (*aggregator).nanflow = 0.0f;
      (*aggregator).underflow = 0.0f;
      (*aggregator).overflow = 0.0f;
    for (bin_0 = 0;  bin_0 < 100;  ++bin_0) {
      (*aggregator).values[bin_0] = 0.0f;
    }
  }
```

Generated code

```
// Specific logic of how to increment the aggregator with input values.
__device__ void increment(Aggregator* aggregator, float input_residual) {
  const float weight_0 = 1.0f;
  int bin_0;

  atomicAdd(&(*aggregator).entries, weight_0);
  if (isnan(input_residual)) {
    atomicAdd(&(*aggregator).nanflow, weight_0);
  }
  else if (input_residual < -5.0) {
    atomicAdd(&(*aggregator).underflow, weight_0);
  }
  else if (input_residual >= 5.0) {
    atomicAdd(&(*aggregator).overflow, weight_0);
  }
  else {
    bin_0 = floor((input_residual - -5.0) * 10.0);
    atomicAdd(&(*aggregator).values[bin_0], weight_0);
  }
}

// Specific logic of how to combine two aggregators.
__device__ void combine(Aggregator* total, Aggregator* item) {
  int bin_0;
  atomicAdd(&(*total).entries, (*item).entries);
  atomicAdd(&(*total).nanflow, (*item).nanflow);
  atomicAdd(&(*total).underflow, (*item).underflow);
  atomicAdd(&(*total).overflow, (*item).overflow);
  for (bin_0 = 0;  bin_0 < 100; ++bin_0) {
    atomicAdd(&(*total).values[bin_0], (*item).values[bin_0]);
  }
}
```

Generated code

```
__host__ void floatToJson(FILE* out, float x) {
  if (isnan(x))
    fprintf(out, "\"nan\"");
  else if (isinf(x)  &&  x > 0.0f)
    fprintf(out, "\"inf\"");
  else if (isinf(x))
    fprintf(out, "\"-inf\"");
  else
    fprintf(out, "g", x);
}

// Specific logic of how to print out the aggregator.
__host__ void toJson(Aggregator* aggregator, FILE* out) {
  int bin_0;
  fprintf(out, "{\"version\": \"1.0\", \"type\": \"Bin\", \"data\": ");
  fprintf(out, "{\"low\": -5.0, \"high\": 5.0, \"entries\": ");
  floatToJson(out, (*aggregator).entries);
  fprintf(out, ", \"nanflow:type\": \"Count\"");
  fprintf(out, ", \"nanflow\": ");
  floatToJson(out, (*aggregator).nanflow);
  fprintf(out, ", \"underflow:type\": \"Count\"");
  fprintf(out, ", \"underflow\": ");
  floatToJson(out, (*aggregator).underflow);
  fprintf(out, ", \"overflow:type\": \"Count\"");
  fprintf(out, ", \"overflow\": ");
  floatToJson(out, (*aggregator).overflow);
  fprintf(out, ", \"values:type\": \"Count\"");
  fprintf(out, ", \"values\": [");
  for (bin_0 = 0;  bin_0 < 100;  ++bin_0) {
    floatToJson(out, (*aggregator).values[bin_0]);
    if (bin_0 != 99)
      fprintf(out, ", ");
  }
```

Generated code

```
  fprintf(out, "], \"name\": \"residual\"}");
  fprintf(out, "}\n");
}

// Generic blockId calculation (3D is the most general).
__device__ int blockId() {
  return blockIdx.x + blockIdx.y * gridDim.x + blockIdx.z * gridDim.x * gridDim.y;
}

// Generic blockSize calculation (3D is the most general).
__device__ int blockSize() {
  return blockDim.x * blockDim.y * blockDim.z;
}

// Generic threadId calculation (3D is the most general).
__device__ int threadId() {
  return threadIdx.x + threadIdx.y * blockDim.x + threadIdx.z * blockDim.x * blockDim.y;
}

// Wrapper for CUDA calls to report errors.
void errorCheck(cudaError_t code) {
  if (code != cudaSuccess) {
    fprintf(stderr, "CUDA error: s\n", cudaGetErrorString(code));
    exit(code);
  }
}

// User-level API for initializing the aggregator (from CPU or GPU).
//
//   aggregators: array of aggregators to fill in parallel.
//   numAggregators: number of aggregators to fill.
//
__global__ void initialize(Aggregator* aggregators, int numAggregators) {
  zero(&aggregators[(threadId() + blockId() * blockSize()) numAggregators]);
```

Generated code

```
}

// User-level API for filling the aggregator with a single value (from GPU).
//
//   aggregators: array of aggregators to fill in parallel.
//   numAggregators: number of aggregators to fill.
//   input_*: the input variables you used in the aggregator's fill rule.
//
__device__ void fill(Aggregator* aggregators, int numAggregators, float input_residual) {
  increment(&aggregators[(threadId() + blockId() * blockSize()) numAggregators], input_resi
}

// User-level API for filling the aggregator with arrays of values (from CPU or GPU).
//
//   aggregators: array of aggregators to fill in parallel.
//   numAggregators: number of aggregators to fill.
//   input_*: the input variables you used in the aggregator's fill rule.
//   numDataPoints: the number of values in each input_* array.
//
__global__ void fillAll(Aggregator* aggregators, int numAggregators, float* input_residual,
  int id = threadId() + blockId() * blockSize();
  if (id < numDataPoints)
    fill(aggregators, numAggregators, input_residual[id]);
}

// User-level API for combining all aggregators in a block (from CPU or GPU).
//
// Assumes that at least `numAggregators` threads are all running at the same time (can be
// synchronized with `__syncthreads()`. Generally, this means that `extract` should be call
// no more than one block, which suggests that `numAggregators` ought to be equal to the ma
// number of threads per block.
//
//   aggregators: array of aggregators to fill in parallel.
//   numAggregators: number of aggregators to fill.
```

Generated code

```
//   result: single output
//
__global__ void extract(Aggregator* aggregators, int numAggregators, Aggregator* result) {
  // merge down in log(N) steps until the thread with id == 0 has the total for this block
  int id = threadId();

  // i should be the first power of 2 larger than numAggregators/2
  int i = 1;
  while (2*i < numAggregators) i <<= 1;

  // iteratively split the sample and combine the upper half into the lower half
  while (i != 0) {
    if (id < i  &&  id + i < numAggregators) {
      Aggregator* ours = &aggregators[id numAggregators];
      Aggregator* theirs = &aggregators[(id + i) numAggregators];
      combine(ours, theirs);
    }

    // every iteration should be in lock-step across threads in this block
    __syncthreads();
    i >>= 1;
  }

  // return the result, which is in thread 0's copy (aggregators[0])
  if (id == 0) {
    Aggregator* blockLocal = &result[blockId()];
    memcpy(blockLocal, aggregators, sizeof(Aggregator));
  }
}

// Test function provides an example of how to use the API.
//
//   numAggregators: number of aggregators to fill.
//   numBlocks: number of independent blocks to run.
```

Generated code

```
//   numThreadsPerBlock: number of threads to run in each block.
//   input_*: the input variables you used in the aggregator's fill rule.
//   numDataPoints: the number of values in each input_* array.
//
void test(int numAggregators, int numBlocks, int numThreadsPerBlock, float* input_residual,
  // Create the aggregators and call initialize on them.
  Aggregator* aggregators;
  cudaMalloc((void**)&aggregators, numAggregators * sizeof(Aggregator));
  initialize<<<1, numThreadsPerBlock>>>(aggregators, numAggregators);
  errorCheck(cudaPeekAtLastError());
  errorCheck(cudaDeviceSynchronize());

  float* gpu_residual;
  errorCheck(cudaMalloc((void**)&gpu_residual, numDataPoints * sizeof(float)));
  errorCheck(cudaMemcpy(gpu_residual, input_residual, numDataPoints * sizeof(float), cudaMe

  // Call fill next, using the same number of blocks, threads per block, and memory allocat
  // fillAll is a __global__ function that takes arrays; fill is a __device__ function that
  // takes single entries. Use the latter if filling from your GPU application.
  fillAll<<<numBlocks, numThreadsPerBlock>>>(aggregators, numAggregators, gpu_residual, numD
  errorCheck(cudaPeekAtLastError());
  errorCheck(cudaDeviceSynchronize());

  errorCheck(cudaFree(gpu_residual));

  // Call extract and give it an Aggregator to overwrite.
  Aggregator* resultGPU;
  cudaMalloc((void**)&resultGPU, sizeof(Aggregator));
  extract<<<1, numThreadsPerBlock>>>(aggregators, numAggregators, resultGPU);

  // Now you can free the collection of subaggregators from the GPU.
  cudaFree(aggregators);
```

Generated code

```
    Aggregator resultCPU;
    cudaMemcpy(&resultCPU, resultGPU, sizeof(Aggregator), cudaMemcpyDeviceToHost);

    // Now you can free the total aggregator from the GPU.
    cudaFree(resultGPU);

    // This Aggregator can be written to stdout as JSON for other Histogrammar programs to in
    // (and plot).
    toJson(&resultCPU, stdout);
  }
}

// Optional main runs a tiny test.
/*
int main(int argc, char** argv) {
  int numAggregators = 5;
  int numBlocks = 2;
  int numThreadsPerBlock = 5;

  int numDataPoints = 10;
  float input_residual[10] = {-0.09f, 0.81f, 1.52f, -1.22f, -0.58f, 1.03f, -1.52f, -1.57f, 1.

  HistogrammarCUDA_0::test(numAggregators, numBlocks, numThreadsPerBlock, input_residual, num
}
*/

#endif  // HISTOGRAMMARCUDA_0
```