P100 OPTIMIZATION AND CUDA 8

Peng Wang

HPC Developer Technology, NVIDIA

OVERVIEW

- GP100 architecture
  - GP100 is the architecture name. P100 is the GPU product name.
- Memory
- Instruction
- CUDA Lambda
- Note: I will focus on P100 (GP100). Titan X (GP102) and GeForce 1080 (GP104) do have some differences in SM, caching and memory.

OPTIMIZATION ON P100

- Mostly the same as in previous gen GPUs: should see significant improvement just by recompile and run
  - The typical best practices, e.g. memory coalescing, shared memory, occupancy, etc, are all the same.
  - May need to tune block size, register limit, etc in some cases
- Improving performance of productivity features
  - Unified memory (UM)
    - You don’t have to use UM if you prefers explicit memory management
    - I expect explicit memory management leads to better perf in most cases
  - CUDA lambda

GPU COMPARISON

	P100 (GP100)	K40 (GK110)
Double Precision TFlop/s	**5.3**	1.4
Single Precision TFlop/s	**10.6**	4.3
Half Precision Tflop/s	**21.2**	NA
Memory Bandwidth (GB/s)	**732**	288
#SM	**56**	15
L2 Cache Size (MB)	**4**	1.5
Memory Size	**16GB**	12GB

fp32/bytes: 7.2 vs 7.5
fp64/bytes: 3.6 vs 2.4

GP100 SM

	GP100	GK110
CUDA Cores	64	192
Register File	256 KB	256 KB
Shared Memory	64 KB	48 KB
Active Threads	2048	2048
Active Blocks	32	16

GP100 SM

Register-bound kernels will have the same occupancy.
Shared memory and block slot bound kernels may see an occupancy increase.
E.g. 64 size block can reach 100% occupancy on P100.

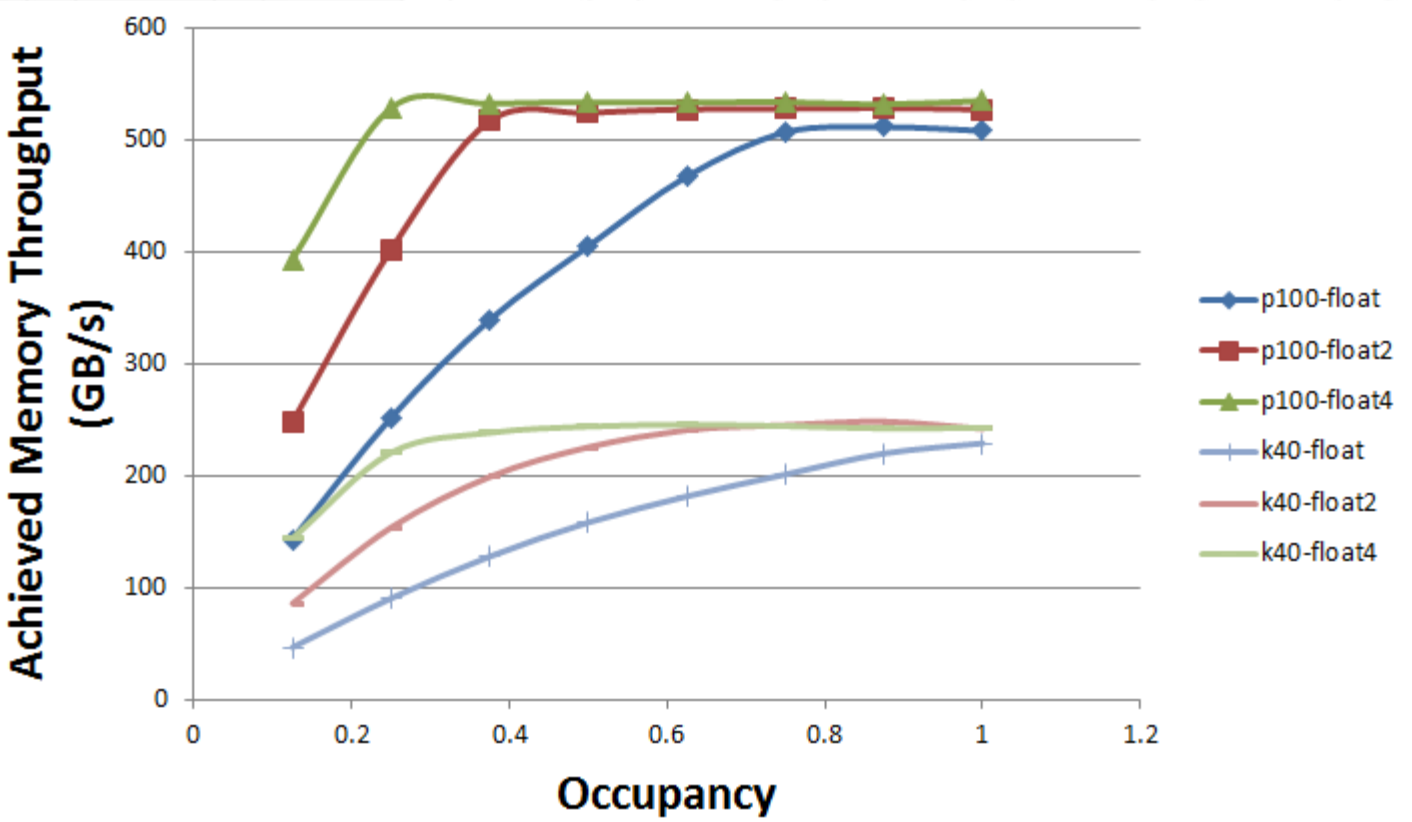
MEMORY

GLOBAL MEMORY

- HBM2: optimization techniques basically same as GDDR5: coalescing, alignment, SOA, etc
- Bandwidth: peak 732 GB/s = 4096bit * 0.715 GHz * 2 (DDR) / 8 (bit-to-byte)
- Latency: similar to GDDR5
  - A BOE calculation
    - Total required concurrency (aka Little's law): ~ BW*L ~ 2.5X of K40
    - SM ratio: 56/15~3.7X of K40.
    - Required concurrency/SM: ~2.5/3.7~0.7
  - P100 should requires similar, sometimes less bytes-in-flight to saturate bandwidth.

COPY EXPERIMENT

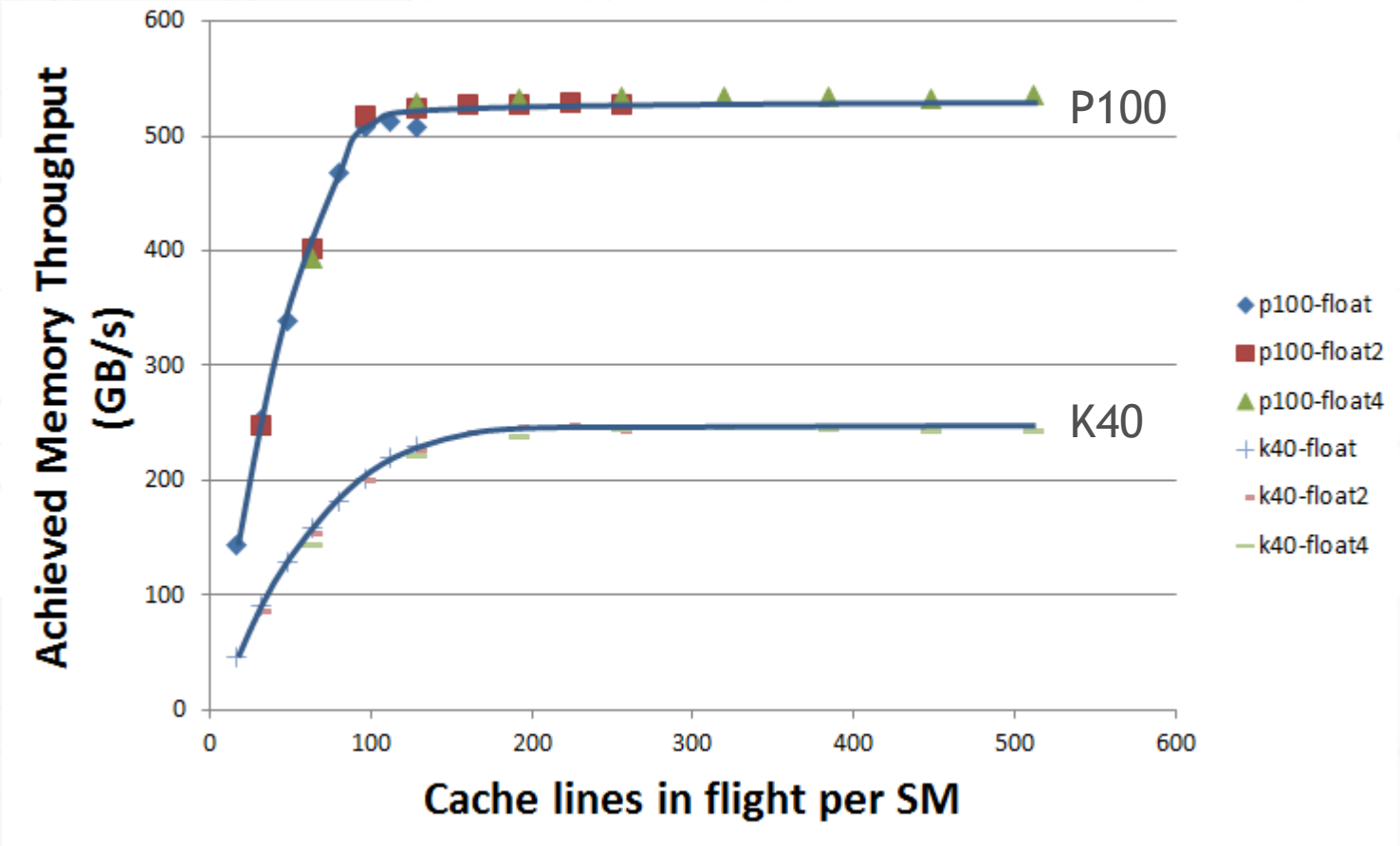
Copy kernel: each thread read one word, writes one word.



- For latency-bound kernel, higher occupancy increase perf
- When occupancy is good enough, improving it no longer helps
- For the same occupancy, increasing bytes-in-flight per thread can increase perf

COPY KERNEL

Redraw the data on a different x-axis



For the copy kernel, P100 reaches peak throughput with less bytes-in-flight.

ECC

- ECC-protected: Reg/Tex/L1/SMEM/L2/DRAM
- ECC is mostly overhead-free
  - No overhead in storage space
  - No overhead in bandwidth
    - Only exception is scattered write. But still much less overhead than GDDR5.

TEX/L1

- Unified tex/L1 cache
- Global loads are cached by default (-dlcm=ca by default)
  - 32B transaction.
    - 128B in K40. For scattered access, no need to turn L1 off to reduce transaction size.
  - On GP104, default is uncached
    - To ensure caching on both GP100 and GP104, use __ldg
- Selective caching to reduce thrashing
  - Use -dlcm=cg to turn off L1 caching.
  - Add __ldg explicitly to selected variables

LDG OR TEX?

- 1D data
  - Global loads are issued in group of 8 threads
  - TEX are issued in group of 4 threads.
  - For 4B word, since transaction size is 32B, global loads achieves better bandwidth for 4B data
  - For larger word, TEX has similar memory throughput as LDG
  - TEX and LDG have different indexing mode. May affect instruction bound code.
- 2D/3D data
  - To utilize 2D/3D locality, or interpolation capability, use TEX

CACHING EXPERIMENT

```
for(int i=0; i<num_fetches; i++)
    sum = sum + load(a);
```

where load(a) is a[i*scale] or tex1D(a, i*scale)
*On GP10x, need to use __ldg(&a[i*scale])

	Tex	LDG
float	1.28 TB/s	1.85 TB/s
float2	2.54 TB/s	2.59 TB/s
float4	2.54 TB/s	2.60 TB/s

SHARED MEMORY

- Only 4B bank mode.
- 64KB.
  - Each block can use at most 48KB.
  - No longer split with L1. Previous call to cudaDeviceSetCacheConfig will just be ignored on Pascal

LOCAL MEMORY

- Local is cached in L1
- If you see more spills when compiling for sm_60
  - Try increasing maxrregcount

INSTRUCTION

FP64 ATOMICS

- Native fp64 atomics
  - Kepler uses a SW solution based on atomicCAS
  - For cases with lots of collision, improves performance significantly

```
if (PATTERN == 0) {
  atomicAdd(&a[gid], 1); // DIST
} else if (PATTERN == 1) {
  atomicAdd(&a[threadIdx.x], 2); // SINGLE
} else if (PATTERN == 2) {
  atomicAdd(&a[0], 1); // SAME
}
```

GOPS/s	K40	P100	P100/K40
DIST	13.9	28	2X
SINGLE	0.08	2.8	35X
SAME	0.000113	0.48	4248X

FP16 MATH

- fp16: 2x throughput of fp32
  - half, half2
  - See cuda_fp16.h: also has half2float, etc.
- For peak throughput, use paired operation for 2 fp16 instructions w/ half2.
  - E.g. half2 hmul2(half2 a, half2 b)
- Kepler also supports fp16 as storage, math in fp32
  - Need CUDA 7.5+
- fp16 atomics

CUDA LAMBDA

LAMBDA

- C++11 feature, CUDA 7.5+
- Concise syntax for defining anonymous functions
- Write CUDA in “directive-like” way
  - No need to define kernel for every loop
  - Unified CPU and GPU loops

NO NEED TO DEFINE EXPLICIT KERNEL

```
for (int i = 0; i < n; i++) {
  z[i] = a * x[i] + y[i];
 };
```

C++

```
__global__ void vec_add(float *x, float *y, float *z, float a, int n) {
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) z[i] = a * x[i] + y[i];
};

vec_add<<<(n+127)/128, 128>>>(d_x, d_y, d_z, a, n);
```

CUDA

```
for_each(counting_iterator<int>(0), counting_iterator<int>(n), [=] __device__ (int i) {
    d_z[i] = a * d_x[i] + d_y[i];
  });
```

CUDA+Lambda

UNIFIED CPU AND GPU LOOPS

With Unified Memory, pointer is unified

```
#ifdef USE_GPU
for_each(counting_iterator<int>(0), counting_iterator<int>(n), [=] __device__ (int i) {
#else
for (int i = 0; i < n; i++) {
#endif
    z[i] = a * x[i] + y[i];
 });
```

P100 TUNING SUMMARY

- Memory: better latency hiding, L1/tex caching
- Instruction: fp64 atomics, fp16
- CUDA Lambda
- Unified memory