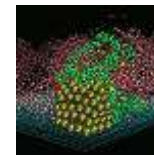
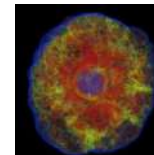
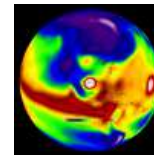
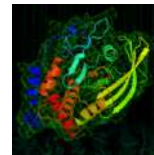
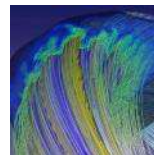
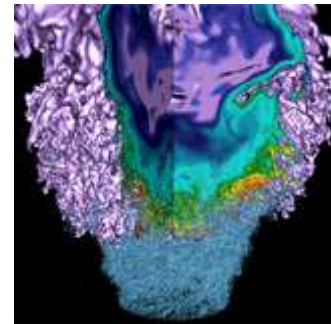
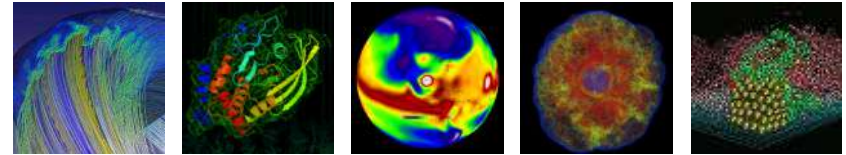


Hierarchical Roofline Analysis on GPUs



Charlene Yang
Lawrence Berkeley National Laboratory
ISC 2019, Frankfurt

- **Hierarchical Roofline on NVIDIA GPUs**
 - **L1, L2, HBM, System Memory**
- **Methodology for Roofline Data Collection**
 - **Machine characterization: peak bandwidth and peak GFLOP/s**
 - **Application characterization: FLOPs, bytes, runtime**
- **Three Examples**
 - **GPP from BerkeleyGW, HPGMG from AMReX, conv2d from TensorFlow**



Hierarchical Roofline

Roofline Performance Model



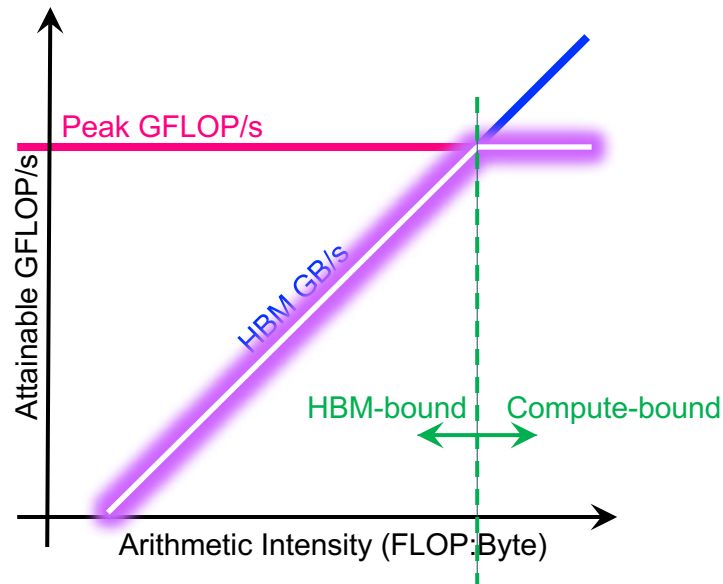
- Sustainable performance is bound by

$$\text{GFLOP/s} = \min \left\{ \begin{array}{l} \text{Peak GFLOP/s} \\ \text{AI} * \text{Peak GB/s} \end{array} \right.$$

- Arithmetic Intensity (AI) =

$$\text{FLOPs} / \text{Bytes}$$

- Log-log scale makes it easy to extrapolate performance along Moore's Law

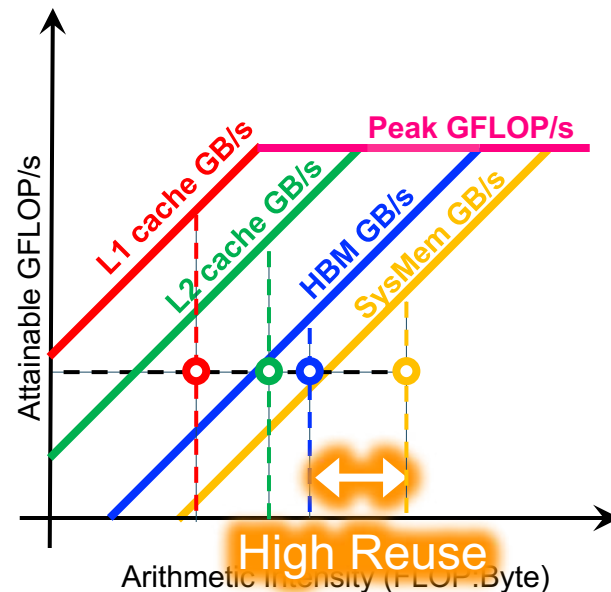


Transition @ AI ==
Peak Gflop/s / Peak GB/s ==
'Machine Balance'

Hierarchical Roofline



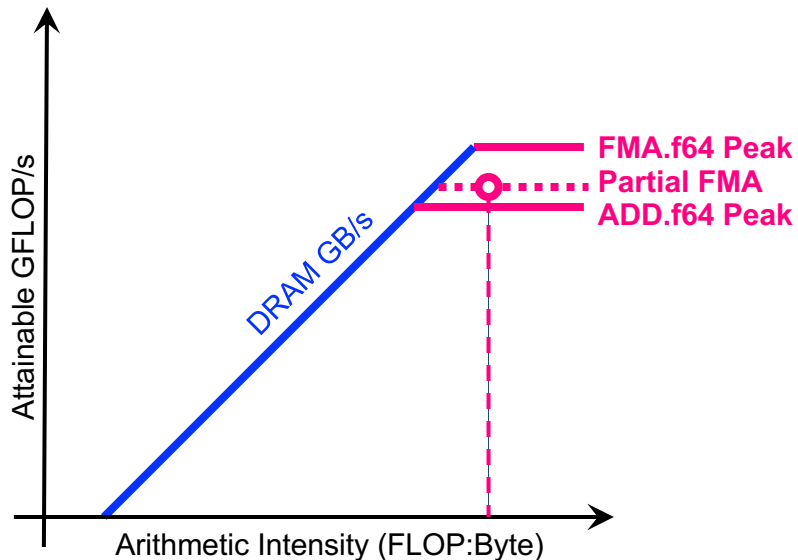
- Superposition of multiple Rooflines
 - Incorporate full memory hierarchy
 - Arithmetic Intensity =
$$\text{FLOPs} / \text{Bytes}_{L1/L2/HBM/SysMem}$$
- Each kernel will have multiple AI's but one observed GFLOP/s performance
- Hierarchical Roofline tells you **cache locality**



Multiple Compute Ceilings



- Impact of **instruction mix**
- Applications are usually a mix of FMA.f64, ADD.f64, MUL.f64...
- Performance is a **weighted** average ... bound by a **partial FMA** ceiling



Hierarchical Roofline on NVIDIA GPUs



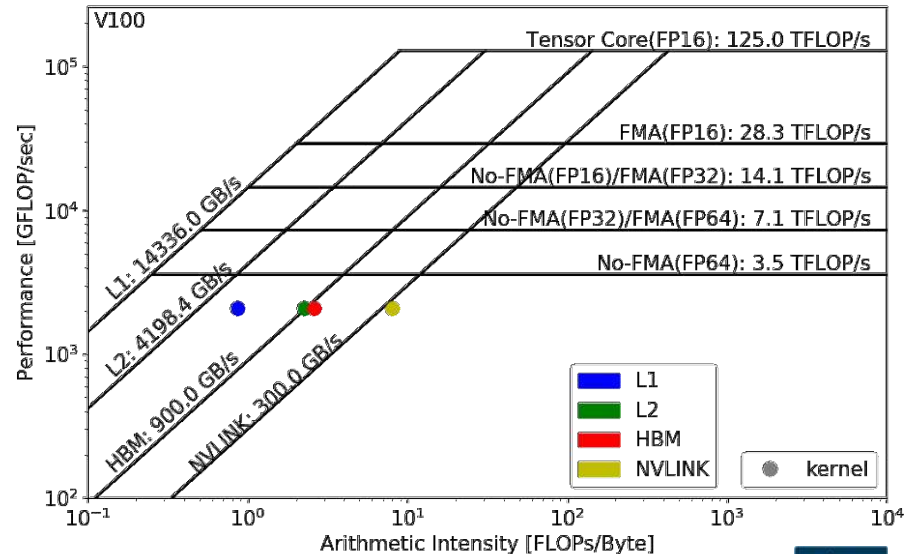
Our goal is to incorporate full memory hierarchy into Roofline

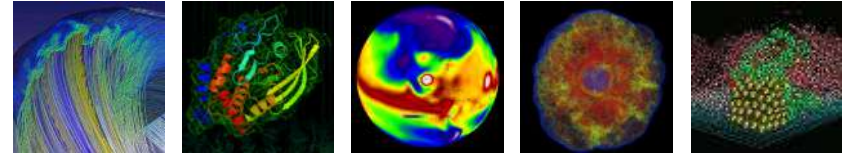
- L1, L2, HBM, System Memory (NVLINK/PCIe)

Also, instruction mix, data type...

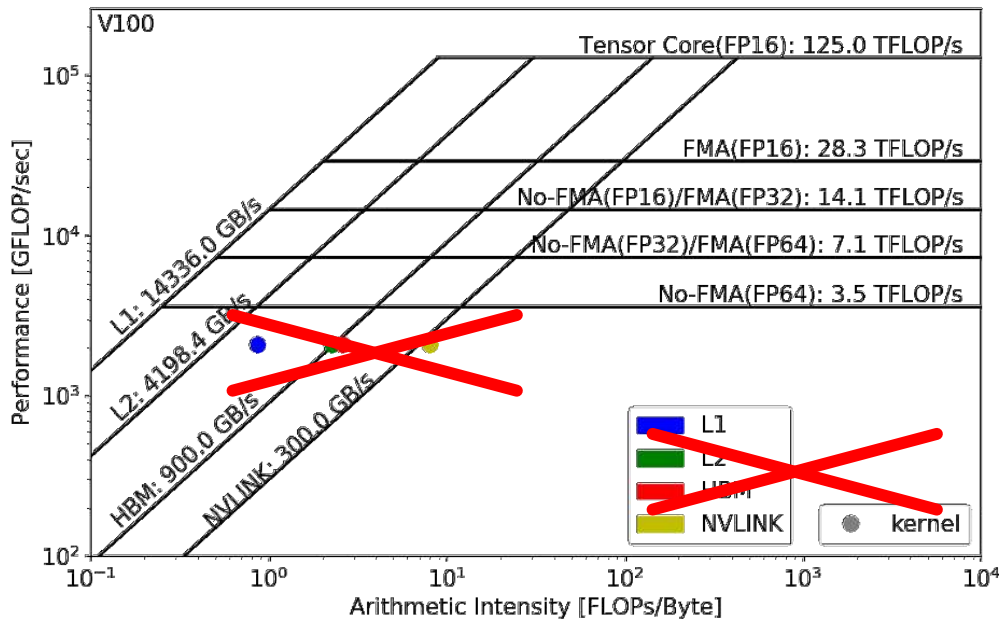
- FMA/no-FMA
- FP64, FP32, FP16
- CUDA core/Tensor core
- ...

Certain ceilings can be omitted
if irrelevant to application





Methodology to Collect Roofline Data



How to get the ceilings?

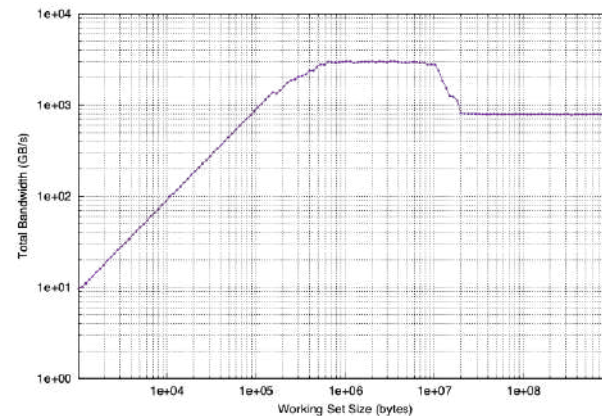
- compute and bandwidth

Theoretical vs Empirical

Empirical Roofline Toolkit (ERT)

- runs micro benchmarks
- **More Realistic**
- power constraints, *etc*

- **Empirical Roofline Toolkit (ERT)**
 - Different than the architecture specs, **MORE REALISTIC**
 - Reflects **actual** execution environment (power constraints, *etc*)
 - Sweeps through a range of configurations, and **statistically stable**
 - Data elements per thread
 - FLOPs per data element
 - Threadblocks/threads
 - Trails per dataset
 - *etc*



Kernel.c

- actual compute
- customizable

Driver.c

- setup
- call kernels
- loop over parameters

config script

- set up ranges of parameters

job script

- submit the job and run it

Machine Characterization



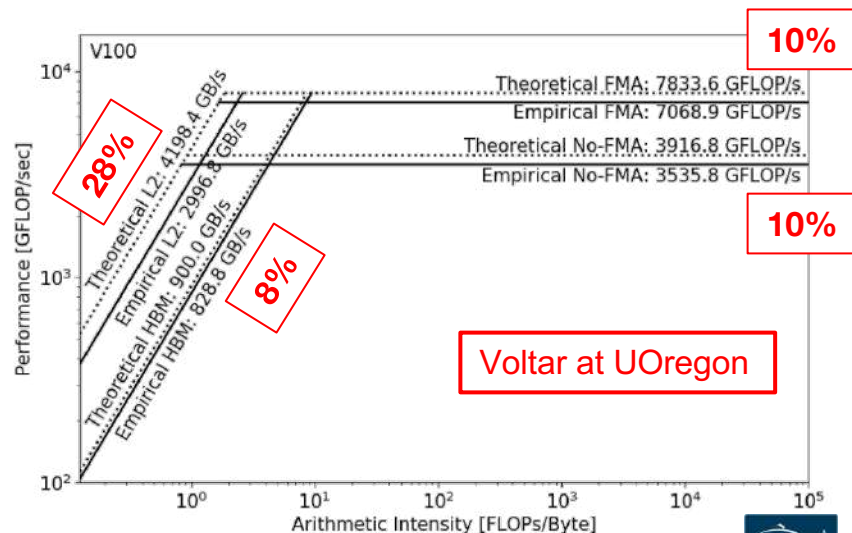
- ERT can't detect all the ceilings yet - IN DEVELOPMENT!
- Theoretical **compute** ceilings on V100:
 - FP64 FMA: 80 SMs x 32 FP64 cores x 1.53 GHz x 2 = 7.83 TFLOP/s
 - FP64 No-FMA: 80 SMs x 32 FP64 cores x 1.53 GHz = 3.92 TFLOP/s
- Theoretical **memory** bandwidths on V100:
 - HBM: 900 GB/s
 - L2: ~4.1 TB/s

Bad News:

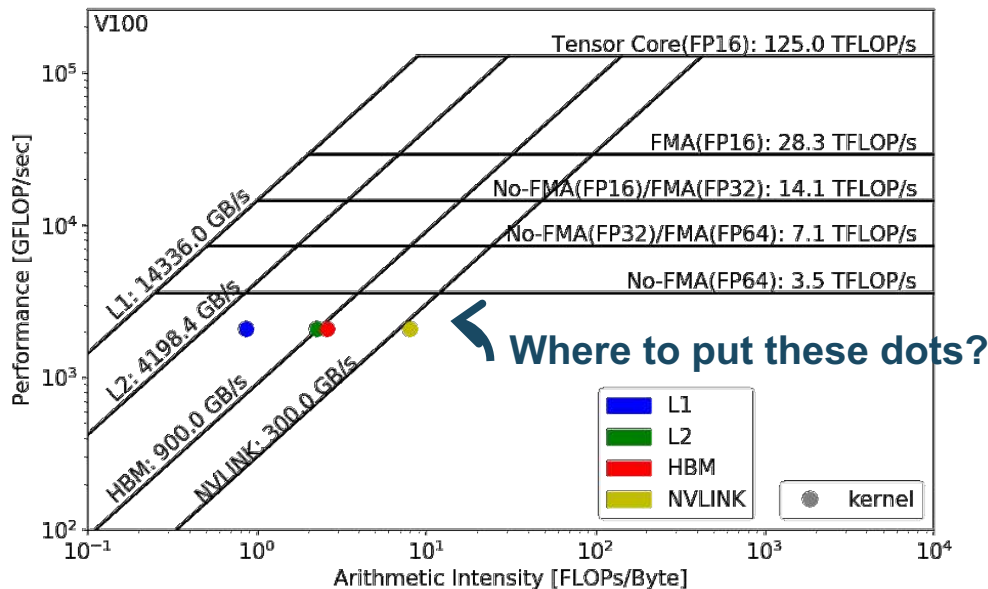
- you may never achieve 7.8 TFLOP/s

Good News:

- you may be closer to the ceiling than you think



Application Characterization



Require three raw measurements:

- Runtime
- FLOPs
- Bytes (on each cache level)

to calculate AI and GFLOP/s:

$$\text{Arithmetic Intensity} = \frac{\text{nvprof FLOPs}}{\text{nvprof Data Movement}}$$

(x: FLOPs/Byte)

$$\text{Performance} = \frac{\text{nvprof FLOPs}}{\text{Runtime}}$$

(y: GFLOP/s)

- **Runtime:**

- Time per invocation of a kernel

```
nvprof --print-gpu-trace ./application
```

- Average time over multiple invocations

```
nvprof --print-gpu-summary ./application
```

- **FLOPs:**

- **CUDA Core:** Predication aware and complex-operation aware (such as divides)

```
nvprof --kernels 'kernel_name' --metrics 'flop_count_xx'
```

```
./application e.g. flop_count_{dp/dp_add/dp_mul/dp_fma, sp*, hp*}
```

- **Tensor Core:** (more details later)

```
--metrics tensor_precision_fu_utilization
```

0-10 integer range, 0-0, 10-125TFLOP/s; multiply by run time -> FLOPs

Application Characterization



- Bytes for different cache levels in order to construct hierarchical Roofline:
 - Bytes = (read transactions + write transactions) x transaction size
 - `nvprof --kernels 'kernel_name' --metrics 'metric_name' ./application`

Level	Metrics	Transaction Size
First Level Cache*	<code>gld_transactions, gst_transactions, atomic_transactions, local_load_transactions, local_store_transactions, shared_load_transactions, shared_store_transactions</code>	32B
Second Level Cache	<code>l2_read_transactions, l2_write_transactions</code>	32B
Device Memory	<code>dram_read_transactions, dram_write_transactions</code>	32B
System Memory	<code>system_read_transactions, system_write_transactions</code>	32B

- **Note:** surface and texture transactions are ignored here for HPC applications

Example Output



```
[cjyang@voltar source]$ nvprof --kernels "1:7:smooth_kernel:1" --metrics  
flop_count_dp --metrics gld_transactions --metrics gst_transactions --  
metrics l2_read_transactions --metrics l2_write_transactions --metrics  
dram_read_transactions --metrics dram_write_transactions --metrics  
systemem_read_bytes --metrics systemem_write_bytes ./hpgmg-fv-fp 5 8
```

context : stream : kernel : invocation

- Export to CSV: `--csv -o nvprof.out`

Invocations	Metric Name	Metric Description	Min	Max	Avg
Device "Tesla V100-PCIe-16GB (0)"					
Kernel: void smooth_kernel<int=6, int=32, int=4, int=8>(level_type, int, int, double, double, int, double*, double*)					
1	flop_count_dp	Floating Point Operations(Double Precision)	30277632	30277632	30277632
1	gld_transactions	Global Load Transactions	4280320	4280320	4280320
1	gst_transactions	Global Store Transactions	73728	73728	73728
1	l2_read_transactions	L2 Read Transactions	890596	890596	890596
1	l2_write_transactions	L2 Write Transactions	85927	85927	85927
1	dram_read_transactions	Device Memory Read Transactions	702911	702911	702911
1	dram_write_transactions	Device Memory Write Transactions	151487	151487	151487
1	systemem_read_bytes	System Memory Read Bytes	0	0	0
1	systemem_write_bytes	System Memory Write Bytes	160	160	160

Plot Roofline with Python



- Calculate Arithmetic Intensity and GFLOP/s performance
 - x coordinate: Arithmetic Intensity
 - y coordinate: GFLOP/s performance

$$\text{Performance (GFLOP/s)} = \frac{\text{nvprof FLOPs}}{\text{Runtime}}, \quad \text{Arithmetic Intensity (FLOPs/Byte)} = \frac{\text{nvprof FLOPs}}{\text{nvprof Data Movement}}$$

- Plot Roofline with Python Matplotlib
 - Example scripts:
 - <https://github.com/cyanguwa/nersc-roofline/tree/master/Plotting>
 - Tweak as needed for more complex Rooflines

Plot Roofline with Python

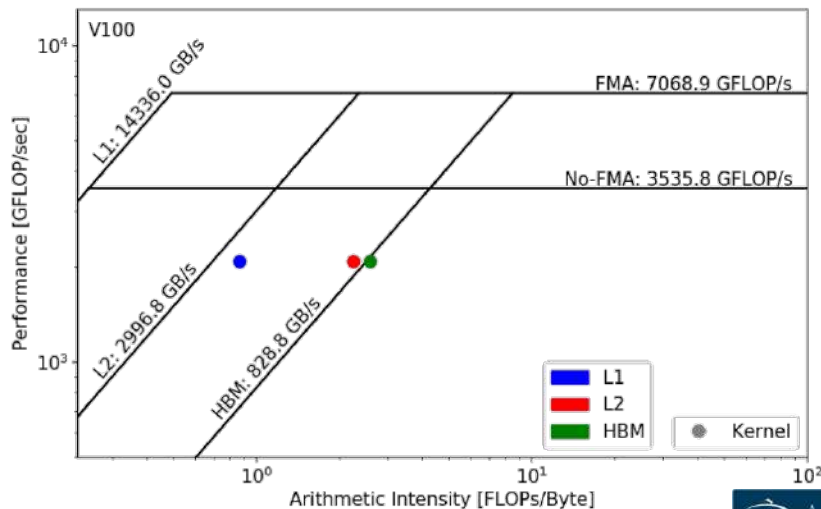


- Quick example: `plot_roofline.py data.txt`
- Accepts space-delimited list for values
- Use quotes to separate names/labels

data.txt

```
# all data is space delimited
memroofs 14336.0 2996.8 828.758
mem_roof_names 'L1' 'L2' 'HBM'
comproofs 7068.86 3535.79
comp_roof_names 'FMA' 'No-FMA'

# omit the following if only plotting roofs
# AI: arithmetic intensity; GFLOPs: performance
AI 0.87 2.25 2.58
GFLOPs 2085.756683
labels 'Kernel'
```



Recap: Methodology to Construct Roofline

1. Collect Roofline ceilings

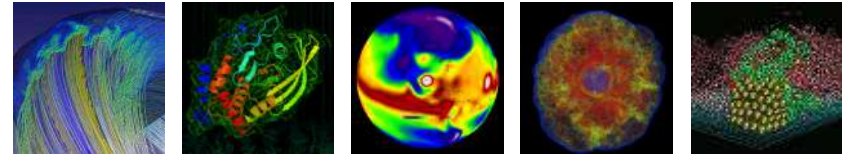
- ERT: <https://bitbucket.org/berkeleylab/cs-roofline-toolkit>
- **compute** (FMA/no FMA) and **bandwidth** (DRAM, L2, ...)

2. Collect application performance

- `nvprof: --metrics, --events, --print-gpu-trace`
- **FLOPs, bytes** (DRAM, L2, ...), **runtime**

3. Plot Roofline with Python Matplotlib

- **arithmetic intensity, GFLOP/s** performance, **ceilings**
- example scripts: <https://github.com/cyanguwa/nersc-roofline>



Roofline Analysis: Three Examples

Example 1: GPP



- GPP (General Plasmon Pole) kernel from BerkeleyGW (Material Science)
- <https://github.com/cyanguwa/BerkeleyGW-GPP>
- Medium problem size: 512 2 32768 20
- Tensor-contraction, abundant parallelism, large reductions
- Low FMA counts, divides, complex double data type, HBM data 1.5GB

Pseudo Code

```
do band = 1, nbands           #blockIdx.x
  do igp = 1, ngpown          #blockIdx.y
    do ig = 1, ncouls         #threadIdx.x
      do iw = 1, nw           #unrolled
        compute; reductions
```

Example 1: GPP



- **Three experiments:**

Vary n_w from 1 to 6	To study impact of varying Arithmetic Intensity on performance
Compile w/wo FMA	To study impact of instruction mix on performance on performance
Stride i_g loop	To study impact of suboptimal memory coalescing on performance

- **Note that `nvprof` has already taken care of**
 - **Appropriate counting of FLOPs for complex instructions**
 - **div, exp, log and sin/cos should be counted as multiple FLOPs rather than 1**
 - **Appropriate counting of FLOPs for predicated-out threads**
 - **FLOPs are only counted on non-predicated threads**

Example 1: GPP



- Highly parameterizable
 1. Varying `nw` from 1 to 6 to increase arithmetic intensity
 - FLOPs increases, but data movement stays (at least for HBM)

Pseudo Code

```
do band = 1, nbands           #blockIdx.x
  do igp = 1, ngpown          #blockIdx.y
    do ig = 1, ncouls         #threadsIdx.x
      do iw = 1, nw           #unrolled
        compute; reductions
```

2. Compiling with and without FMA

- `-fmad=true/false`

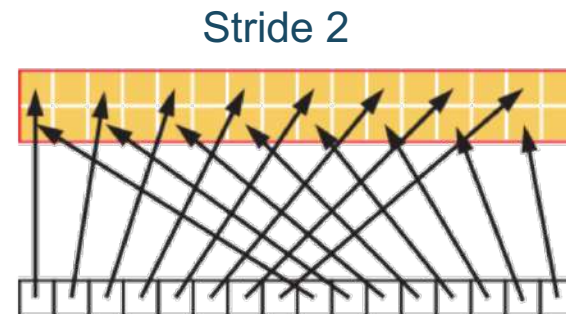
Example 1: GPP



- Highly parameterizable
 3. Striding `ig` loop to analyze impact of suboptimal memory coalescing
 - Split `ig` loop to two loops and place the 'blocking' loop outside

Pseudo Code

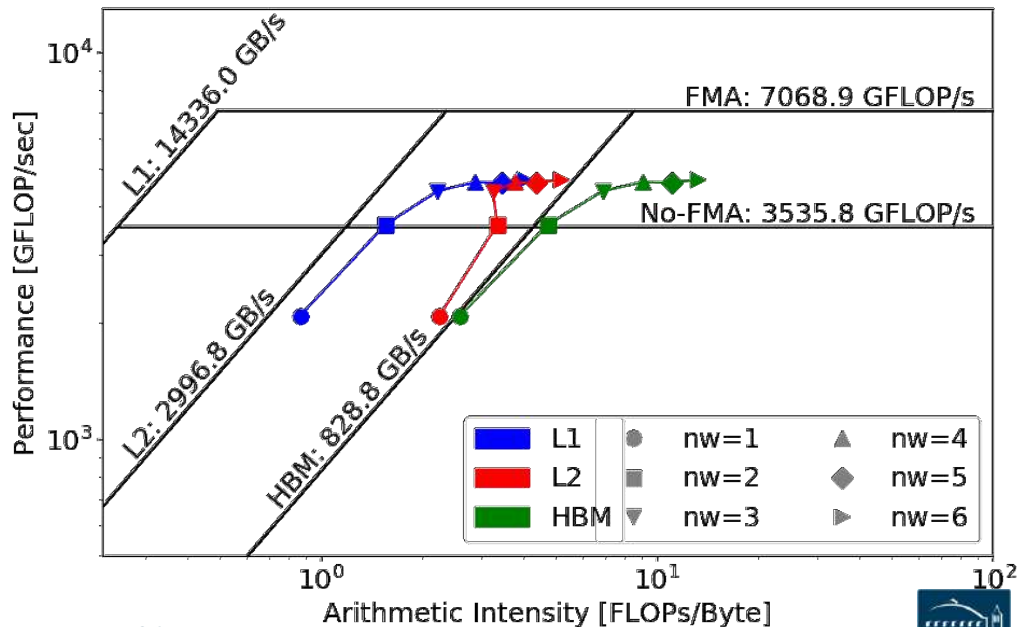
```
do band = 1, nbands           #blockIdx.x
  do igp = 1, ngpown          #blockIdx.y
    do igs = 0, stride - 1
      do ig = 1, ncouls/stride #threadIdx.x
        do iw = 1, nw         #unrolled
          compute; reductions
```



Example 1: GPP Analysis



- Hierarchical Roofline, i.e. bytes are HBM, L2 and unified L1 cache bytes
 - GPP is HBM bound at low nw 's and compute bound at high nw 's
 - FLOPs $\propto nw$
 - HBM bytes: constant
 - L2 bytes: increasing at $\alpha > 1$
 - L1 bytes: constant
- Hierarchical Roofline captures more details about **cache locality**

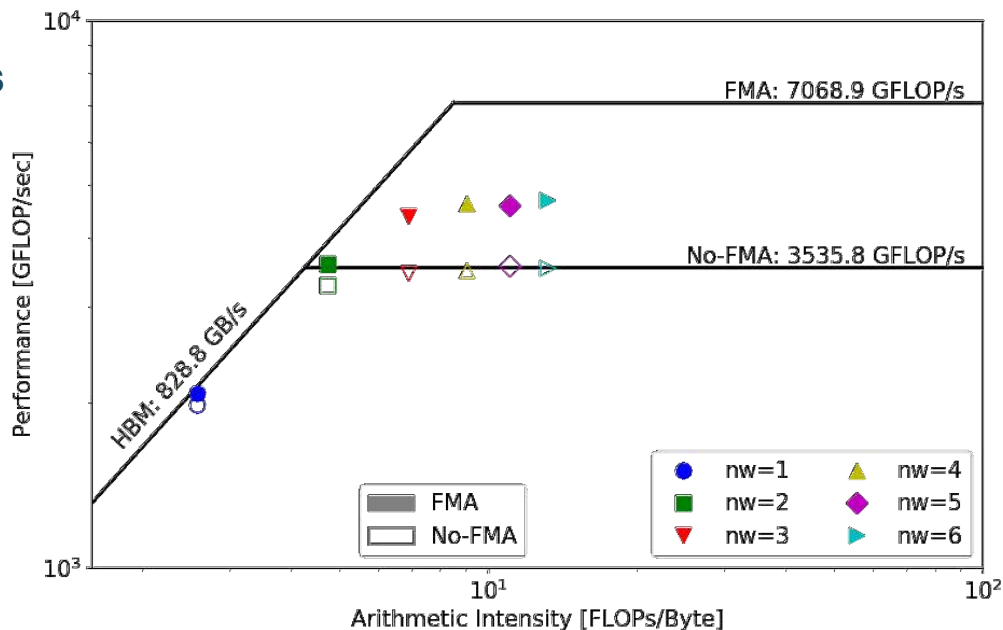


Example 1: GPP Analysis



HBM Roofline, i.e. bytes are HBM bytes

- No-FMA performance converges to no-FMA ceiling, but FMA performance is still far from the FMA ceiling
- Not reaching FMA ceiling due to lack of FMA instructions



Example 1: GPP Analysis



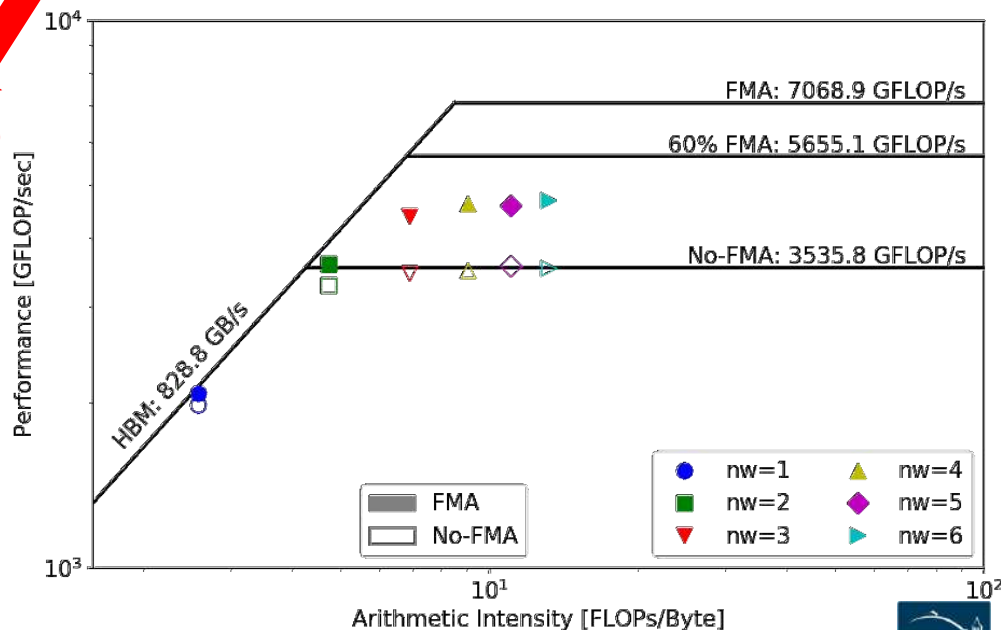
- At $n_w=6$, GPP has $\alpha = \frac{\text{FMA FP64 instr.}}{\text{FMA FP64 instr.} + \text{non-FMA FP64 instr.}} = 60\%$ of FMA instructions

- Expected performance is

$$\beta = \frac{\alpha \times 2 + (1 - \alpha)}{2} = 80\% \text{ of peak}$$

But at $n_w=6$, GPP only achieves 66%

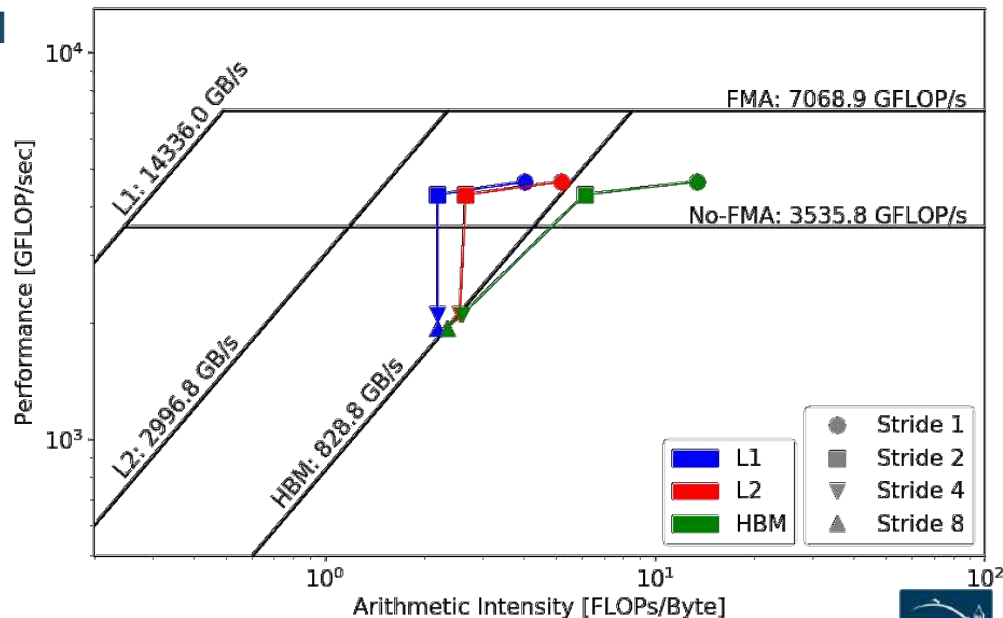
- Other FP/non-FP instructions may be taking up the instruction issue/execution pipeline
- Roofline captures effects of **instruction mix**



Example 1: GPP Analysis



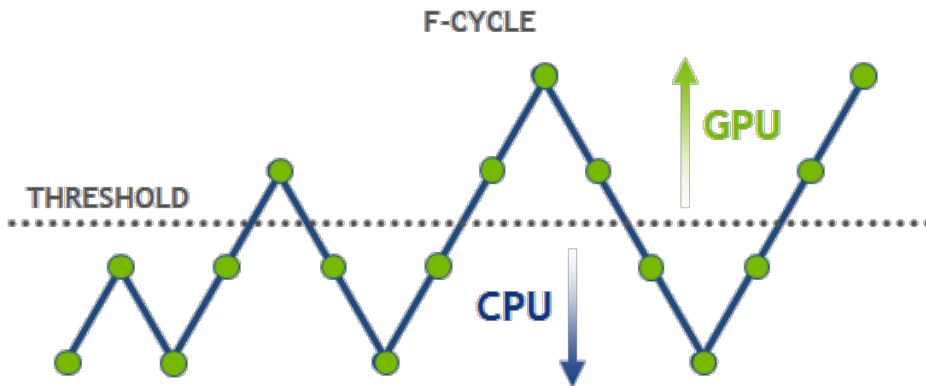
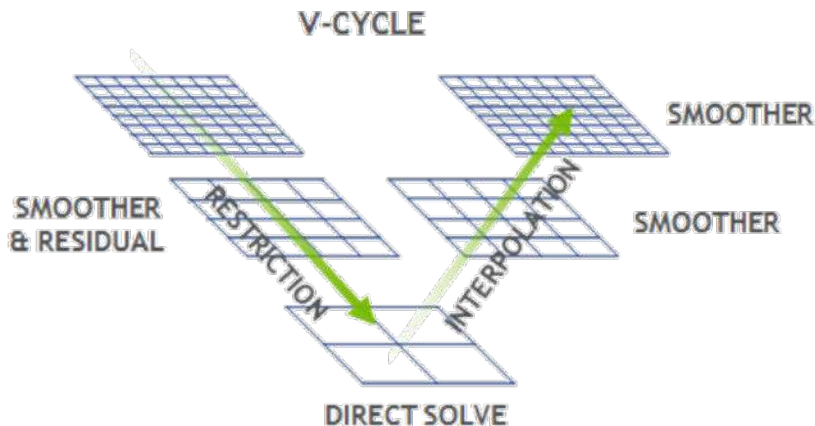
- Hierarchical Roofline, i.e. bytes are HBM, L2 and unified L1 cache bytes
 - L1/L2 bytes doubles from stride 1 to 2, but stays almost constant afterwards
 - at $n_w=6$, GPP moves from compute bound to bandwidth bound
 - Eventually all converge to HBM
- Roofline captures effects of suboptimal **memory coalescing**



Example 2: HPGMG



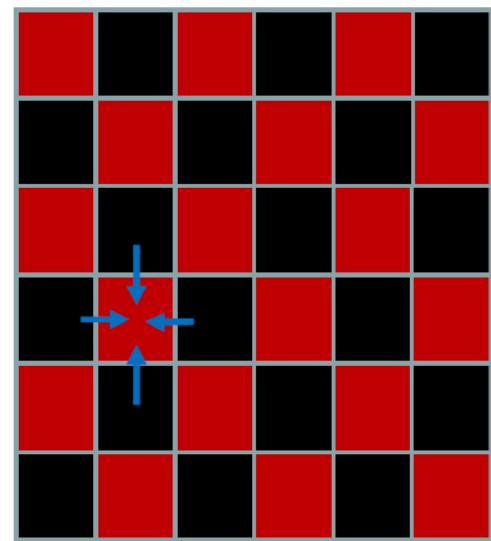
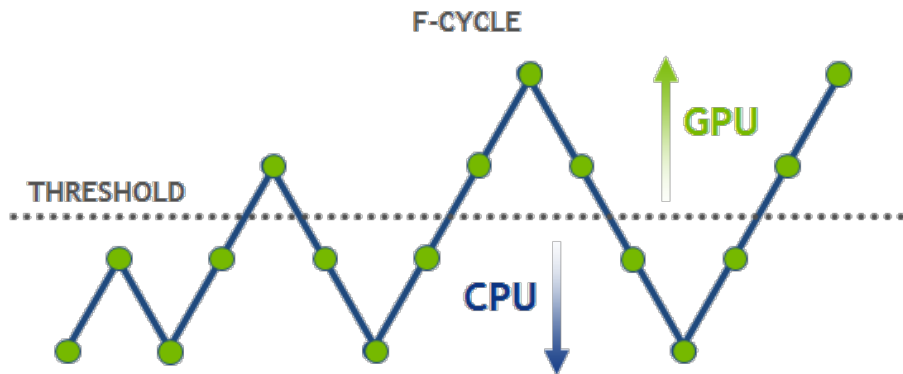
- HPGMG (High-performance Geometric Multigrid) from Adaptive Mesh Refinement code
- <https://bitbucket.org/nsakharnykh/hpgmg-cuda>
- Stencil code, F-cycles and V-cycles, GSRB smoother kernel (Gauss-Seidel Red-Black)



Example 2: HPGMG



- Hybrid GPU and CPU code
 - Example: `hpgmg-fv 7 8`
 - 128^3 box x 8, Level 5-8 run on GPU, Level 1-4 on CPU
- Three versions of GSRB kernel
 - GSRB_FP, GSRB_BRANCH, GSRB_STRIDE2



Example 2: HPGMG



GSRB_FP

```
for(int k=klo; k<(klo+kdim); k++){
  const int ijk = i + j*jStride + k*kStride;
  const double *__restrict__ RedBlack =
    level.RedBlack_FP + ghosts*(1+jStride)
    + ((k^color000)&1)*kStride;
  const double Ax = apply_op_ijk();
  const double lambda = Dinv_ijk();
  const int ij = i + j*jStride;
  xo[ijk] = X(ijk) + RedBlack[ij]*lambda*(rhs[ijk]-
Ax);
}
```

GSRB_BRANCH

```
for(int k=klo; k<(klo+kdim); k++){
  const int ijk = i + j*jStride + k*kStride;
  if(((i^j^k^color000^1)&1)){
    const double Ax = apply_op_ijk();
    const double lambda = Dinv_ijk();
    xo[ijk] = X(ijk) + lambda*(rhs[ijk]-Ax);
  }else{
    xo[ijk] = X(ijk);
  }
}
```

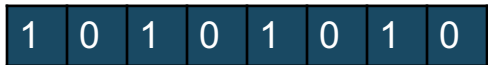


8 elements



8 elements

Sweep



8 threads



8 threads

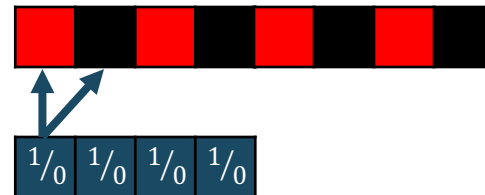
- GSRB_BRANCH has half the FLOPs as GSRB_FP but the same HBM/L1/L2 bytes

Example 2: HPGMG



GSRB_STRIDE2

```
for(int k=klo; k<klo+kdim; k++){
  i = ilo + !((ilo^j^k^color000)&1) + threadIdx.x*2;
  if(i < ilo+idim){
    const int ijk = i + j*jStride + k*kStride;
    xo[ijk] = X(ijk);
  }
  i = ilo + ((ilo^j^k^color000)&1) + threadIdx.x*2;
  if(i < ilo+idim){
    const int ijk = i + j*jStride + k*kStride;
    const double Ax = apply_op_ijk();
    const double lambda = Dinv_ijk();
    xo[ijk] = X(ijk) + lambda*(rhs[ijk]-Ax);
  }
}
```



8 elements

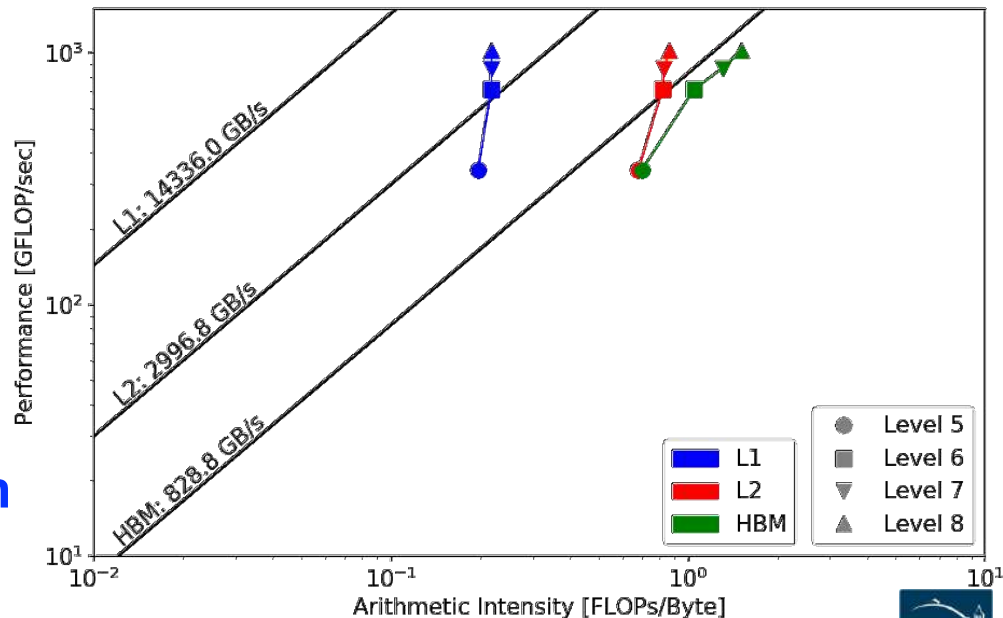
4 threads

- GSRB_STRIDE2 should have the same FLOPs as GSRB_BRANCH, but more bytes? More writes than GSRB_BRANCH.

Example 2: HPGMG Analysis



- **GSRB_FP**, Hierarchical Roofline, i.e. bytes are HBM, L2 and unified L1 cache bytes
- Highly bandwidth bound, inherent to stencil codes
- From Level 5 to Level 8:
 - HBM AI increases due to better Surface: Volume ratio
 - Roughly constant L1/L2 AI due to stencils being ‘tiled’
- Roofline captures **computational characteristics** of the algorithm

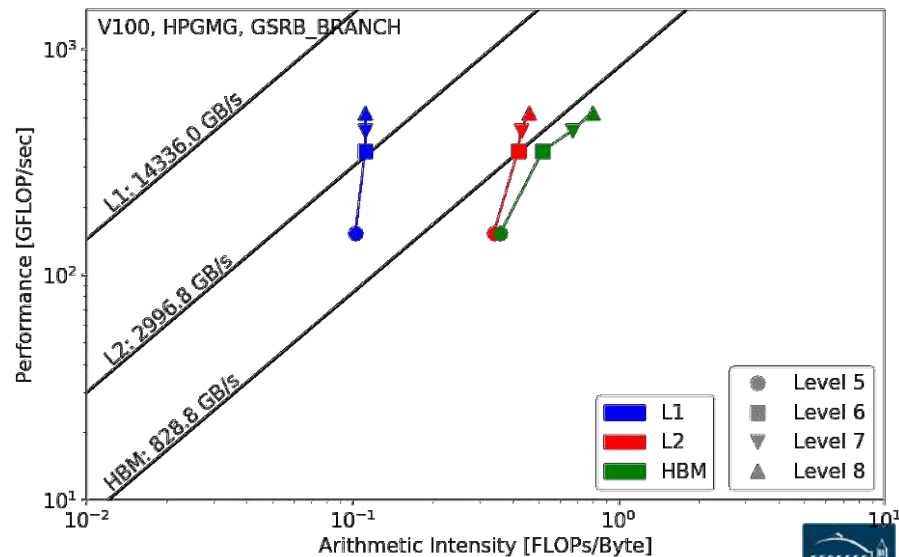
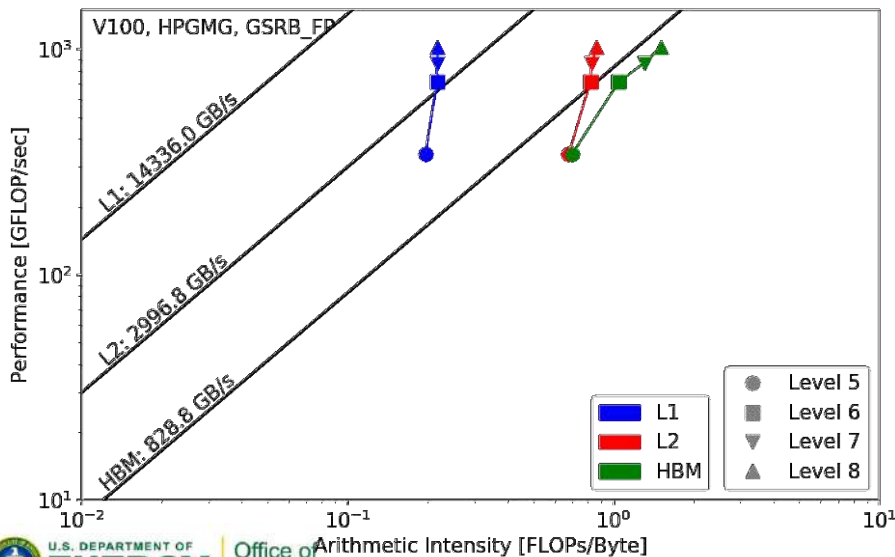


Example 2: HPGMG Analysis



GSRB_FP vs. GSRB_BRANCH

- FLOPs halves, bytes doesn't change, thus AI halves and GFLOP/s halves
- Runtime is comparable even though GFLOP/s has halved
- Same number of threads occupied, only with half predicated in GSRB_BRANCH

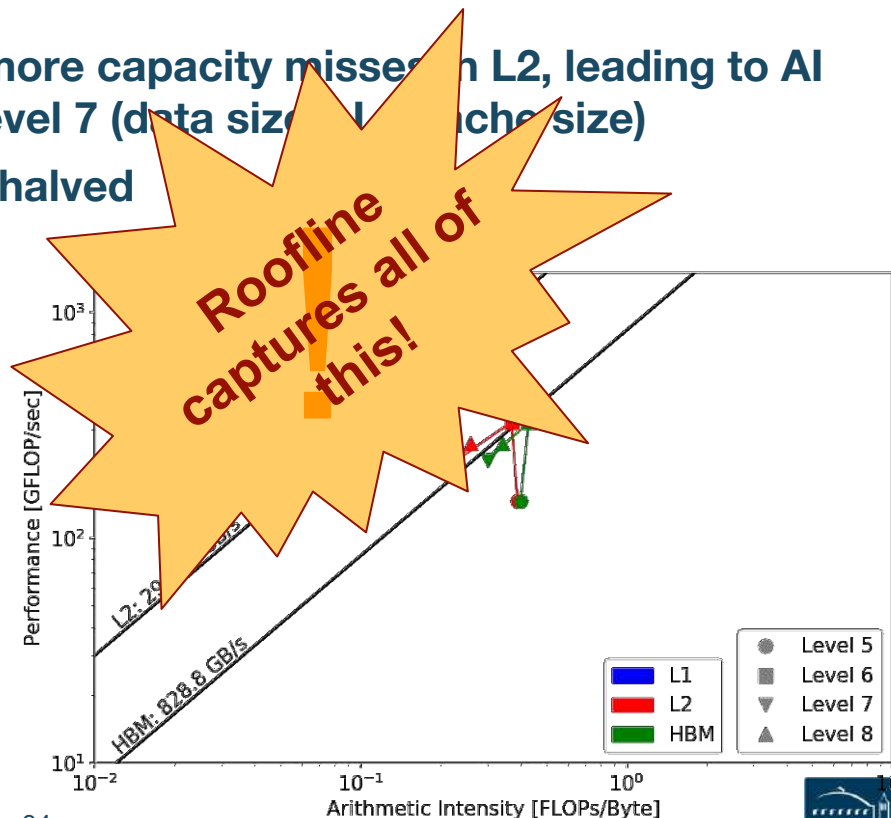
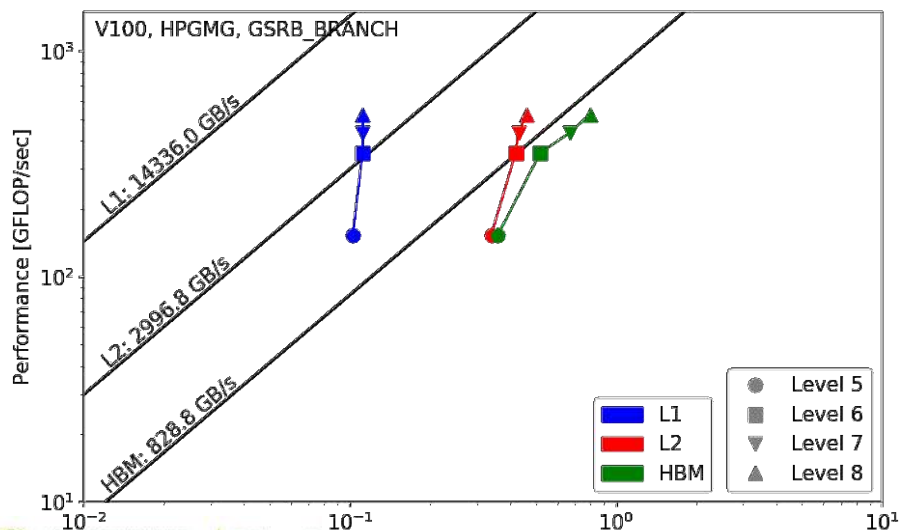


Example 2: HPGMG Analysis



GSRB_BRANCH vs. GSRB_STRIDE2

- Extra writes in GSRB_STRIDE2 cause more capacity misses on L2, leading to AI drop on L2 and DRAM, starting from Level 7 (data size \approx cache size)
- Runtime almost doubled and GFLOP/s halved



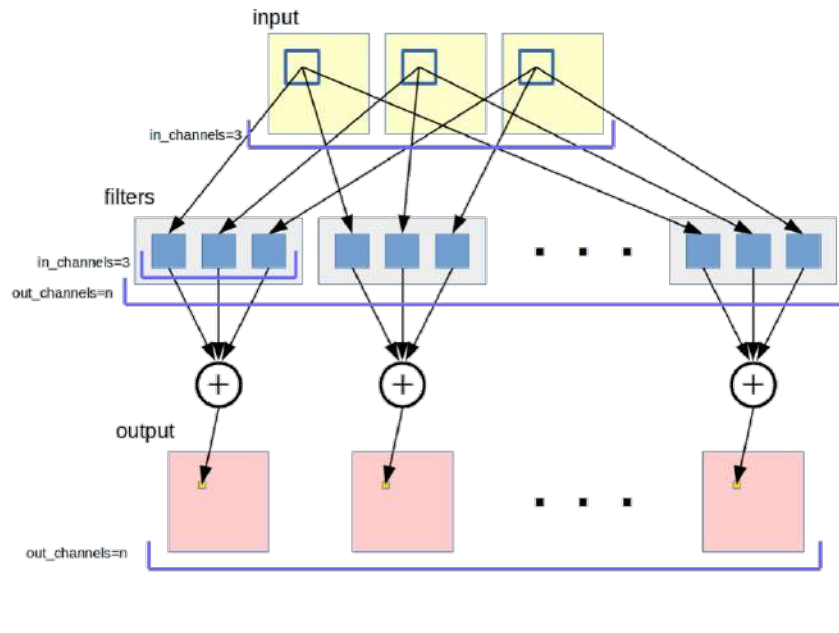
Example 3: conv2d from TensorFlow



- Kernel `tf.nn.conv2d`



<https://www.tensorflow.org>



$$B_{nhwc} = \sum_{m=0}^{C-1} \sum_{k_h=0}^{K_H-1} \sum_{k_w=0}^{K_W-1} A_{n \ h+k_h \ w+k_w \ m} K_{k_h \ k_w \ m \ c}$$

Example 3: conv2d from TensorFlow



- TensorFlow autotuning mechanism
- Split the loop into 'warm-up' and 'measurement'
 - 5 iters and 20 iters
- pyc.driver from PyCUDA
 - need to launch nvprof with
`--profile-from-start off`

```
with tf.Session(config=...) as sess:  
    ...  
  
    #warm-up  
    for i in range(n_warm):  
        result = sess.run(exec_op)  
  
    #measurement  
    pyc.driver.start_profiler()  
    for i in range(n_iter):  
        result = sess.run(exec_op)  
    pyc.driver.stop_profiler()
```

Example 3: conv2d from TensorFlow



exec_op:

- forward pass -- conv in 2D
- backward pass -- conv + derivative
- calibrate -- tensor generation

```
#choose operation depending on pass
if pass=="forward":
    with tf.device(gpu_dev):
        exec_op = output_result
elif pass=="backward":
    with tf.device(gpu_dev):
        opt = tf.train.Gradient\
            DescentOptimizer(0.5)
        exec_op = opt.compute\
            _gradients(output_result)
elif pass=="calibrate":
    with tf.device(gpu_dev):
        exec_op = input_image
```

```
#generate random input tensor
input_image = tf.random_uniform(shape=input_size, minval=0., maxval=1., dtype=dtype)
#create network
output_result = conv2d(input_image, 'NHWC', kernel_size, stride_size, dtype)
```

Example 3: conv2d from TensorFlow



- Each TensorFlow kernel translates to a series of subkernels
 - padding, shuffling, data conversion, *etc*
- TensorFlow based on heuristics decides what subkernels to call
- cuDNN also has some algorithm selection mechanism
- We **INCLUDE** the housekeeping subkernels in our measurements, but **EXCLUDE** the autotuning subkernels

Example 3: conv2d from TensorFlow



- Our FLOP count comes from
`flop_count_sp, flop_count_hp, tensor_precision_fu_utilization`
- Byte count and run time are the sum of the `byte_count` and `run_time` for all subkernels

**FP16 and FP32
are the
input/output
data types**

CAVEATS:

- Housekeeping subkernels may run in FP32, but the main computation is FP16
- TensorFlow may execute computation in FP32 even when input is FP16
- Very coarse quantization for `tensor_precision_fu_utilization`
 - 0-10 integer range, 0 maps to 0 TFLOP/s and 10 maps to 125 TFLOP/s

Example 3: conv2d Analysis

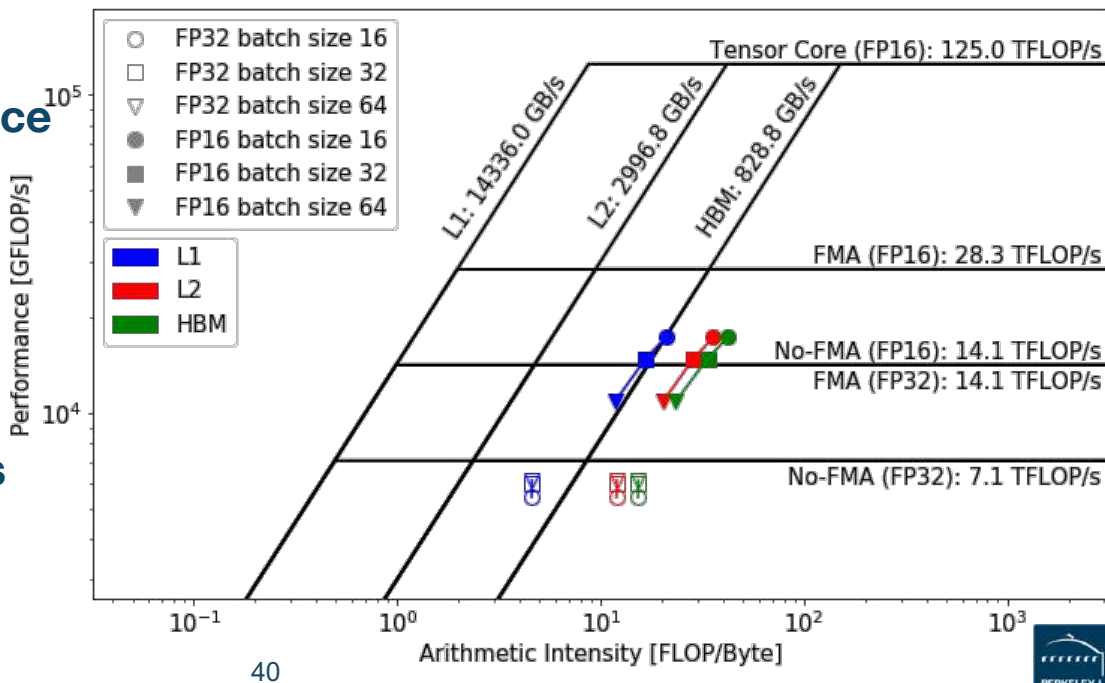


- **Batch Size** 16, 32 and 64, forward pass

- Same underlying algorithm
→ should be same performance

- But, housekeeping kernels are mostly bandwidth bound

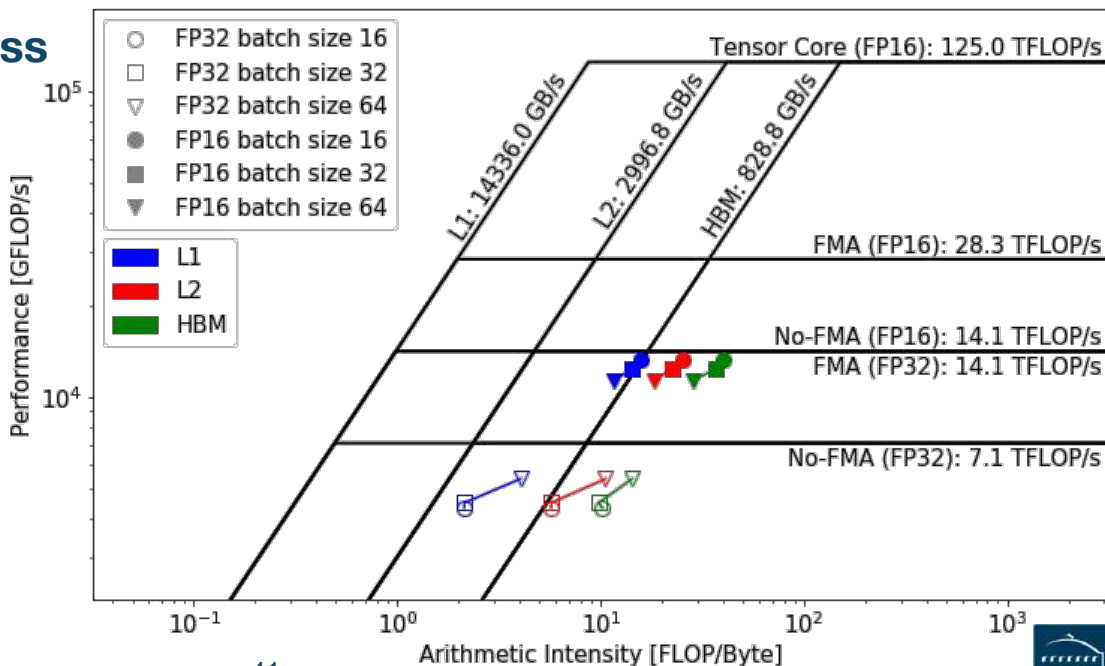
- One reason TF applications are not reaching peak



Example 3: conv2d Analysis



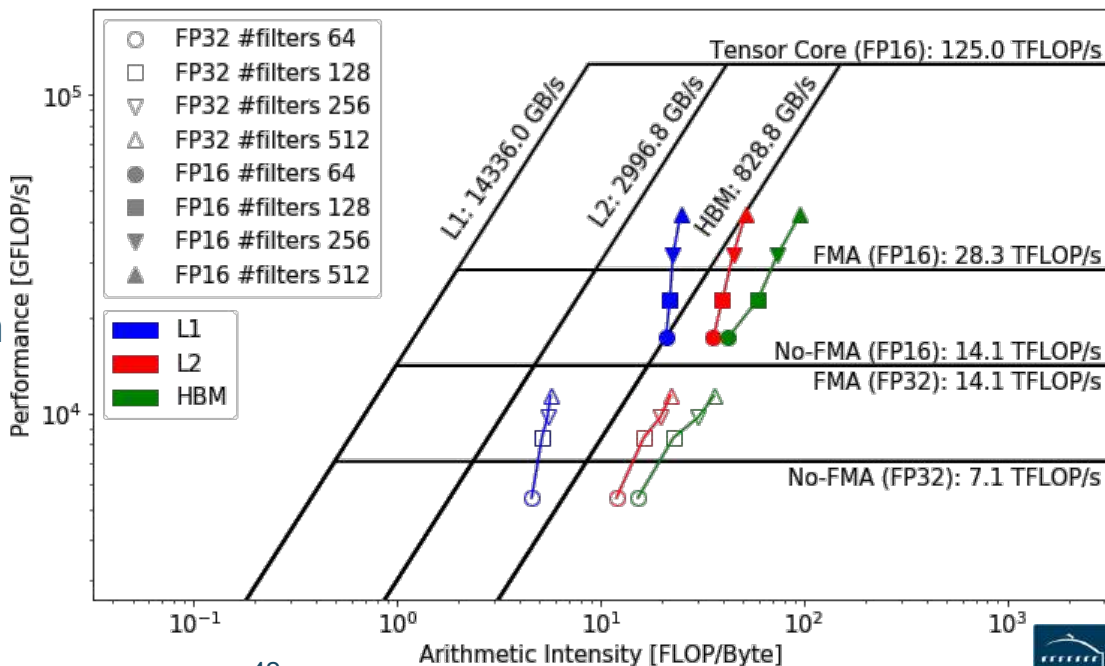
- **Batch Size** 16, 32 and 64, backward pass
- **Similar trend as forward pass**
- **But algorithm changes for FP32 at batch size 64, leading to slightly better performance**



Example 3: conv2d Analysis



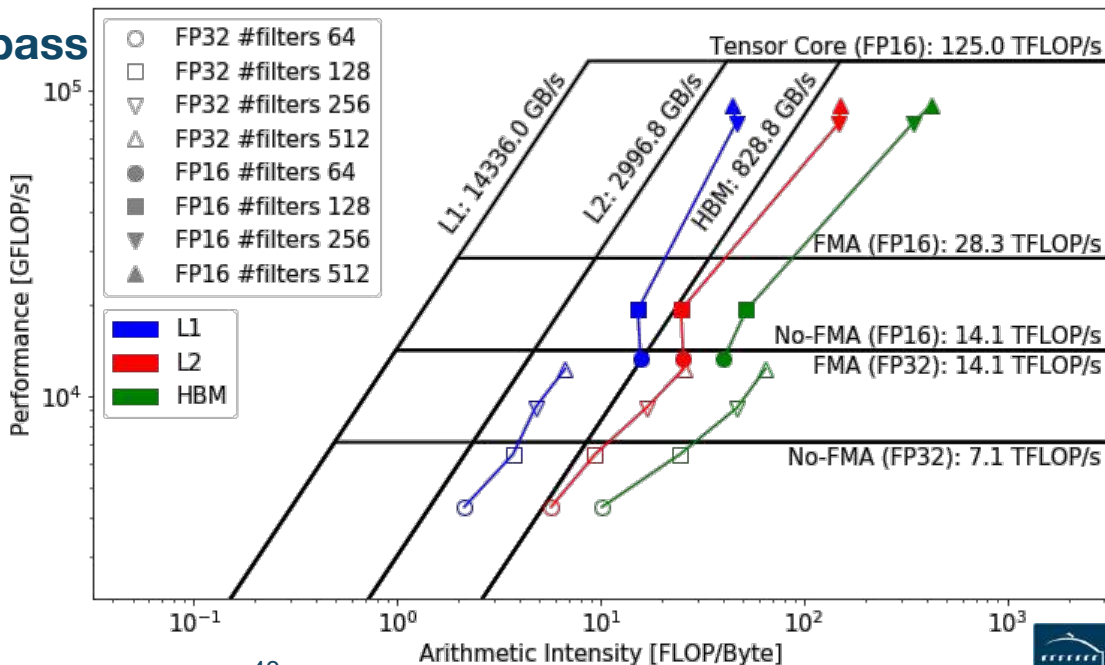
- **Number of Output Filters** 64, 128, 256 and 512, forward pass
- **Increasing intensity and performance**
- **Good L1 locality**
- **cuDNN uses shared mem**



Example 3: conv2d Analysis



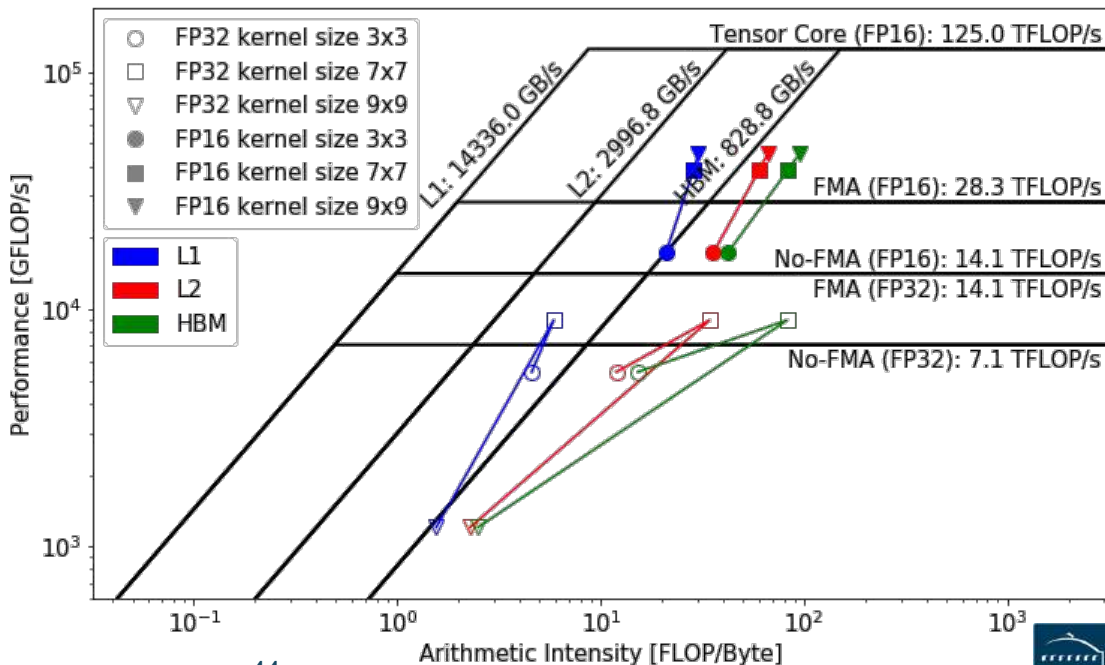
- **Number of Output Filters** 64, 128, 256 and 512, backward pass
- **Similar trend as forward pass**
- **Almost reaching TC peak and FP32 FMA peak**



Example 3: conv2d Analysis



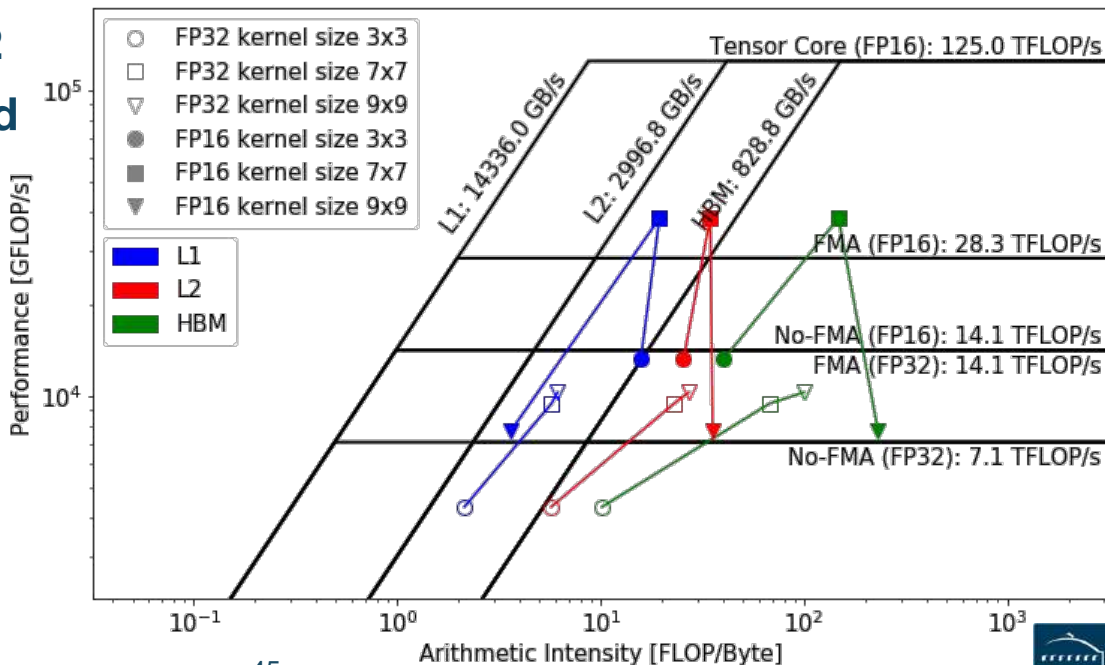
- **Kernel Size** 3x3, 7x7 and 9x9, forward pass
- Increasing intensity and performance
- Algorithm change at 9x9
 - wgrad to FFT
 - may not be efficient use of FFT kernels



Example 3: conv2d Analysis



- **Kernel Size** 3x3, 7x7 and 9x9, backward pass
- TF decides to run in FP32 even though both input and output are in FP16; Data needs to be converted back and forth
- More robust autotuning



- **An effective methodology to construct hierarchical Roofline on NVIDIA GPUs**
 - **ERT for machine characterization**
 - **nvprof for application characterization**

- **Three examples demonstrated the value of this methodology and its ability to understand various aspects of performance on NVIDIA GPUs**
 - **cache locality, instruction mix, memory coalescing, thread predication, reduced precision and Tensor Cores**
 - **GPP from BerkeleyGW, HPGMG from AMReX, and conv2d from TensorFlow**

- S. Williams, A. Waterman and D. Patterson, “Roofline: An Insightful Visual Performance Model for Multicore Architectures,” *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009
- Empirical Roofline Toolkit (ERT):
 - <https://bitbucket.org/berkeleylab/cs-roofline-toolkit>
- Example scripts for plotting Roofline:
 - <https://github.com/cyanguwa/nersc-roofline>
- General Plasmon Pole kernel:
 - <https://github.com/cyanguwa/BerkeleyGW-GPP>
- HPGMG-CUDA kernel:
 - <https://bitbucket.org/nsakharnykh/hpgmg-cuda>

Acknowledgement



- **This material is based upon work supported by the Advanced Scientific Computing Research Program in the U.S. Department of Energy, Office of Science, under Award Number DE-AC02-05CH11231.**
- **This material is based upon work supported by the DOE RAPIDS SciDAC Institute.**
- **This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-05CH11231.**



Thank You