Use Cases of Roofline Analysis

Charlene Yang @NERSC
Use Case 1: BerkeleyGW
Material Science/Chemistry at Exascale

Mat. Sci & Chem apps like VASP, Quantum ESPRESSO, NWChem, GAMESS, QMCPACK, BerkeleyGW, and CP2K are some of the most heavily used apps at DOE facilities.

They are being used to design and understand the fundamental components of Quantum Computers, Solar Cells, OLEDs, Batteries, Catalysts, Bio-Energy, Semiconductors, Sensors, Hydrogen Storage, Carbon Sequestration.
BerkeleyGW

- A massively parallel package for GW calculations
- Sits on top of DFT codes
- Studies **Excited-State properties** of materials
  - Photovoltaics
  - LEDs
  - Quantum Computers
  - Junctions / Interfaces
  - Defect Energy Levels
  - ....
Sigma-GPP

Pseudo Code

\[
\begin{align*}
\text{do } & n1 = 1, \text{ nbands} & n' \text{ e.g. } 2763 \\
\text{do } & \text{igp} = 1, \text{ ngpown} & G' \text{ e.g. } 6633 \\
\text{do } & \text{ig} = 1, \text{ ncouls} & G \text{ e.g. } 26529 \\
\text{do } & \text{iw} = 1, \text{ nw} & E \text{ e.g. } 3
\end{align*}
\]

compute: 1. mixed data types  
 e.g. complex double, double, integer
2. various memory access patterns  
 e.g. (ig,igp)(ig,n1)(igp,n1)(iw,n1)(n1)
3. complex number divisions
4. nw is very small, will be unrolled

reduction: 1. complex numbers
2. all top 3 loops, billions of iterations
Optimization Path

9 Steps:
1. Collapse n’, G’, and G loops
2. Bring n’ loop in; collapse only G’ and G
3. Adjust threadblock size
4. Reduce branching; pull iw loop outside
5. Swap indices to suite parallelisation
6. Simplify code
7. Replace div. with rcp. and mul.
8. Replace abs with power of 2
9. Cache blocking

![ACC PARALLEL LOOP REDUCTION](+)

do n1 = 1, nbands
  do igp = 1, ngpown
    do ig = 1, ncouls
      do iw = 1, nw
        compute and reduction

<table>
<thead>
<tr>
<th></th>
<th>TFLOPs</th>
<th>Time</th>
<th>TFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1.collapse3</td>
<td>3.71</td>
<td>1.63</td>
<td>2.27</td>
</tr>
<tr>
<td>v9.block</td>
<td>2.00</td>
<td>0.57</td>
<td>3.50</td>
</tr>
</tbody>
</table>

3x !!
V1. Naïve Implementation

- Collapse the first 3 loops to gain parallelism

```c
!$ACC PARALLEL LOOP COLLAPSE(3) REDUCTION(+: )
do n1 = 1, nbands
    do igp = 1, ngpown
        do ig = 1, ncouls
            do iw = 1, nw #unrolled
                compute and reduction
```

<table>
<thead>
<tr>
<th></th>
<th>TFLOPs</th>
<th>Time (sec)</th>
<th>TFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1.collapse3</td>
<td>3.71</td>
<td>1.63</td>
<td>2.27</td>
</tr>
</tbody>
</table>
V2. More Compute Per Thread

- Move n’ loop in, and collapse the first 2 loops

```plaintext
!$ACC PARALLEL LOOP COLLAPSE(2) REDUCTION(+: )
do  igp = 1, ngpown
    do  ig = 1, ncouls
        do  n1 = 1, nbands         #unrolled too!
            do  iw = 1, nw           #unrolled
                compute and reduction
```
V2. More Compute Per Thread

- L2/HBM AI increases!
- Very low occupancy
  - 8 warps per SM
  - Register count at 186
- Need more warps to hide latency!
V3. Increase Threadblock Size

- Force threadblock size to be 512, instead of the default 128

```c
!$ACC PARALLEL LOOP COLLAPSE(2) VECTOR_LENGTH(512) REDUCTION(+: )
```

- Register spills but performance may not be bad!

  0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
  ptxas info : Used **186 registers**, 624 bytes cmem[0], 32 bytes cmem[2]

  104 bytes stack frame, **188 bytes** spill stores, **168 bytes** spill loads
  ptxas info : Used **128 registers**, 624 bytes cmem[0], 32 bytes cmem[2]
V3. Increase Threadblock Size

- More bandwidth bound now but latency hiding is successful!

<table>
<thead>
<tr>
<th></th>
<th>TFLOPs</th>
<th>Time</th>
<th>TFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>v2.collapse2</td>
<td>3.71</td>
<td>1.73</td>
<td>2.15</td>
</tr>
<tr>
<td>v3.vector512</td>
<td>3.71</td>
<td>1.40</td>
<td>2.65</td>
</tr>
</tbody>
</table>
V4. Reduce Branching

- Bring iw loop outside of the kernel

```c
do iw = 1, nw  #reduce branching
 !$ACC PARALLEL LOOP COLLAPSE(2) VECTOR_LENGTH(512) REDUCTION(+: )
 do igp = 1, ngpown
   do ig = 1, ncouls
     do n1 = 1, nbands    #unrolled
       compute and reduction
   end do
end do
end do
```

- Fewer variables to be reduced -> lower register pressure

  0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads

  `ptxas info : Used 122 registers, 600 bytes cmem[0], 32 bytes cmem[2]`
V4. Reduce Branching

- Aggregated data for all kernels
  
- BRA instruction count
  14,278,897,053
  5,975,051,812 $\times$ 2

16%

<table>
<thead>
<tr>
<th></th>
<th>TFLOPs</th>
<th>Time</th>
<th>TFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>v3.vector512</td>
<td>3.71</td>
<td>1.40</td>
<td>2.65</td>
</tr>
<tr>
<td>v4.iwoutside</td>
<td>3.52</td>
<td>1.17</td>
<td>3.00</td>
</tr>
</tbody>
</table>
V5. Swap Indices

```
do iw = 1, nw
!$ACC PARALLEL LOOP
  do igp = 1, ngpown
    do ig = 1, ncouls
      do n1 = 1, nbands
        wx_array(iw,n1) to (n1,iw)
  end do
end do
end do
end do
```

<table>
<thead>
<tr>
<th></th>
<th>TFLOPs</th>
<th>Time</th>
<th>TFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>v4.iwoutside</td>
<td>3.52</td>
<td>1.17</td>
<td>3.00</td>
</tr>
<tr>
<td>v5.swapindices</td>
<td>3.52</td>
<td>1.16</td>
<td>3.03</td>
</tr>
</tbody>
</table>
V6. Simplify Code

- Fewer instructions -> less work
  - Pull repeated instructions outside the loop
  - Use temporary variables to hold intermediate values for reuse
- Less branches -> better programming
  - 3 branches is more than 1 branch worse than 2 branches!

<table>
<thead>
<tr>
<th></th>
<th>TFLOPs</th>
<th>Time</th>
<th>TFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>v5.swapindices</td>
<td>3.52</td>
<td>1.16</td>
<td>3.03</td>
</tr>
<tr>
<td>v6.simplify</td>
<td>3.30</td>
<td>1.10</td>
<td>3.00</td>
</tr>
</tbody>
</table>
V7. Replace Divides

- Replace (complex) div. with (double) rcp. and (complex) mul.
- Lower instruction count: 40%
- More bandwidth bound now!

<table>
<thead>
<tr>
<th></th>
<th>TFLOPs</th>
<th>Time</th>
<th>TFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>v6.simplify</td>
<td>3.30</td>
<td>1.10</td>
<td>3.00</td>
</tr>
<tr>
<td>v7.divs</td>
<td>2.09</td>
<td>0.66</td>
<td>3.18</td>
</tr>
</tbody>
</table>
V7. Replace Divides

- Can be confirmed by Nsight Compute profiles
V8. Replace abs(x) with x**2

- $\sqrt{x^2 + y^2} < z \quad \rightarrow \quad x^2 + y^2 < z^2$
- `sqrt(complex)` vs power of 2
- Causing pipeline to wait

<table>
<thead>
<tr>
<th></th>
<th>TFLOPs</th>
<th>Time</th>
<th>TFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>v7.divs</td>
<td>2.09</td>
<td>0.66</td>
<td>3.18</td>
</tr>
<tr>
<td>v8.abs</td>
<td>1.99</td>
<td>0.62</td>
<td>3.23</td>
</tr>
</tbody>
</table>
V8. Replace abs(x) with x**2

Before:

- **Wait:** warp stalled waiting on a fixed latency execution dependency
V8. Replace abs(x) with x**2

After:

- Wait: 46.6% -> 23.7%
V9. Cache Blocking

- Non-coalesced memory access for `aqsntemp`
- Causing Long Scoreboard Warp State
  - Warp stalled waiting for L1TEX (local, global, surface, tex) memory operation
V9. Cache Blocking

- Break loops into chunks and reuse data across threadblocks
- Increase L2 hit rate

```
$ACC LOOP GANG VECTOR
do ig_blk = 1, ig_blksize
$ACC LOOP SEQ
  do ig = ig_blk, ncouls, ig_blksize
```

<table>
<thead>
<tr>
<th></th>
<th>TFLOPs</th>
<th>Time</th>
<th>TFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>v8.abs</td>
<td>1.99</td>
<td>0.62</td>
<td>3.23</td>
</tr>
<tr>
<td>v9.block</td>
<td>2.00</td>
<td>0.57</td>
<td>3.50</td>
</tr>
</tbody>
</table>
V9. Cache Blocking

- Less Long Scoreboard samples and higher L2/L1 hit rate
Summary

9 Steps to Optimize Sigma-GPP

1. Collapse n’, G’, and G loops
2. Bring n’ loop in; collapse only G’ and G
3. Adjust threadblock size
4. Reduce branching; pull iw loop outside
5. Swap indices to suite parallelisation
6. Simplify code
7. Replace div. with rcp. and mul.
8. Replace abs with power of 2
9. Cache blocking

```c
!$ACC PARALLEL LOOP REDUCTION(+: )
do n1 = 1, nbands
   do  igp = 1, ngpown
      do  ig = 1, ncouls
         do  iw = 1, nw
             compute and reduction
```

<table>
<thead>
<tr>
<th></th>
<th>TFLOPs</th>
<th>Time</th>
<th>TFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1.collapse3</td>
<td>3.71</td>
<td>1.63</td>
<td>2.27</td>
</tr>
<tr>
<td>v9.block</td>
<td>2.00</td>
<td>0.57</td>
<td>3.50</td>
</tr>
</tbody>
</table>

3x !!
Summary

- Code is still bandwidth and latency bound
  - shared memory
  - lower register count
  - improve FMA ratio

- Together with profilers, Roofline provides the complete solution for your performance analysis and optimization needs!
Use Case 2: conv2d from TensorFlow
conv2d from TensorFlow

- Kernel tf.nn.conv2d

\[ B_{n h w c} = \sum_{m=0}^{C-1} \sum_{k_h=0}^{K_H-1} \sum_{k_w=0}^{K_W-1} A_{n h+k_h w+w_h m} K_{k_h k_w m c} \]
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.GradientDescentOptimizer(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)
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

```python
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()
```
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
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 these quantities across all subkernels.

CAVEATS:
- Housekeeping subkernels may run in FP32 even when input 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.
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
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**
conv2d Analysis

- **Number of Output Filters**: 64, 128, 256, and 512, forward pass

- **Increasing intensity and performance**

- **Good L1 locality**

- cuDNN uses shared mem
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
**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
  - More robust autotuning
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**

- **Comply with dimension requirements**
Summary
Summary

• We presented an effective methodology to collect machine and application data, and construct hierarchical Roofline on NVIDIA GPUs.

• Two use cases demonstrated the value of this methodology and showed its ability to readily understand various aspects of performance and performance bottlenecks on NVIDIA GPUs:
  - GPP from BerkeleyGW, and conv2d from TensorFlow.
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

Sam Williams, swwilliams@lbl.gov
Yunsong Wang, yunsongwang@lbl.gov
Charlene Yang, cijyang@lbl.gov