Accelerating 3D FFT with Half-Precision Floating Point Hardware on GPU

Students: Yanming Kang (HKUST) and Tullia Glaeser (Tulane)
Mentors: Ed D’Azevedo (ORNL) and Stan Tomov (ICL, UTK)
Discrete Fourier Transform (DFT) & Fast Fourier Transform (FFT)

- DFT \([O(N^2)]\): for num. computations in digital signal processing (incl fast convolution, spectrum analysis)
  - \(N\) discrete time series signals \(\rightarrow\) \(N\) discrete frequency components (amplitude + phase)

- FFT \([O(N\log N)]\): Fast algorithms for DFT -- widely used num. Algorithm -- plays vital role in many scientific and engineering applications (image processing, speech recog., data analysis, large scale simulations)
  - Maj. time in large comp. apps
  - Cooley + Tukey Algorithm:
    i. Symmetry of DFT: \(X_{N+k} = X_k = \overline{X_k}\)
    ii. Divide DFT alg. into odd + even parts
      - \(\rightarrow\) halved the computations to be \(O(2M)\) where \(M\) is half of \(N\) \(\rightarrow\) \(O(N)\)
        i. Keep doing this recursively \(\rightarrow\) halves computation cost every time \(\rightarrow\) \(O(N\log N)\)
  - To keep improving performance/time -- implement it on GPU
Implementing 1D, 2D, & 3D FFT

● 1D FFT of x:
  a. $x = 1D$ array, $B \times (4 \times N/4)$ matrices or $1 \times (4 \times N/4 \times B)$ tensor ($B =$ # of batches)
  b. Find DFT of each of those matrices
  c. Multiply by twiddle factor ($W = e^{-i2\pi kn/N}$)

● 2D FFT:
  a. $x = (m \times n \times batch)$
  b. Reshape $x$ to be 1D array $[m \times n \times batch, 1, 1]$
  c. Call 1D FFT on it
  d. Transpose & do 1D FFT in other direction

● 3D (breakdown shown in pic):
  a. Take 1D FFT in each direction OR
  b. Take 2D FFT in 2 directions & 1D in last dir.

● MATLAB + CUDA
  a. Currently use CUBLAS/CUTLASS and Radix-4
Mixed Precision & Tensor Cores

- Tensor: “a mathematical object analogous to but more general than a vector, represented by an array of components that are functions of the coordinates of a space” -- large dense matrix
- NVIDIA Volta microarchitecture ft. specialized computing units, Tensor Cores
- Tensor Core support → mixed precision -- matrix multiplication operations done w/ half-precision input data (FP16) -- the rest FFT done on single precision data (FP32)
- FP16 arithmetic enables Volta Tensor Cores which offer 125 TFlops of computational throughput on generalized matrix-matrix multiplications (GEMMs) and convolutions, an 8X increase over FP32
- Matrix entries multiplied in neural networks are small w/ respect to value of prev. Iter. → can use half precision, result is still small in val. → result accumulated to other much larger val., in single precision to avoid precision loss
- Deep neural network training = tolerant to precision loss up to certain degree
Inefficiency with Transform -- volta_sgemm_fp16_128x64_nn

1D, \(n=256\), \(\text{batch}=1\), \(\text{iter}=1\) 
69.6%

2D, \(m=256\), \(n=256\), \(\text{batch}=1\), \(\text{iter}=1\) 
75.1%

3D, \(m=256\), \(n=256\), \(k=256\), \(\text{batch}=1\), \(\text{iter}=1\) 
86.5%
The FFT (radix-n1) in matrix form

We use n1 = 4 since $F_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}$ can be stored in fp16 with no error.

$$Y = F_N X = (W_N \times F_{n2} \cdot \text{reshape}(X, n1, n2)^T) F_{n1}$$

where $(W_N)_{k,l} = e^{\frac{2\pi ikl}{N}}$
The algorithm

//Batched 1d FFT of length N
Radix_4_FFT_recursion(X, N, Batch):
   If N=4 then
      Return F4 * X  //See X as a (4 by N/4 by Batch) array
     permute(X, [2,1,3])  //X as a (N/4 by 4 by Batch) array
     Y ← Radix_4_FFT_recursion(X, N/4, Batch*4)
     Multiply elementwise Y with W_N
      Return Y * F4
   End

(batched gemm)

Splitting is done before gemm
Combining is done after gemm
x(32) = s1(32) * x_hi(16) + s2(32) * x_lo(16),
Gemm is done to x_hi, x_lo
Let $b=0$, $b=1$, $b=\text{Batch}-1$. Then:

- $\text{reshape}$
  - $b=0$
  - $b=1$
  - $b=\text{Batch}-1$

- $\frac{N}{4}$

- $\text{permute}$
  - $\frac{N}{4}$

- The FFT is applied to $\left(\frac{N}{4}, \text{Batch} \times 4\right)$

- The reshaped output is:
  - $b=0$
  - $b=1$
  - $b=\text{Batch}-1$

- $W_N$

- The FFT is applied to $(X, N, \text{Batch})$
**CUTLASS (CUDA Templates for Linear Algebra Subroutines)**

The most expensive step in the recursion: the second batched gemm

Result1 = X * F4_re; Result2 = X * F4_im

where

- F4_re, F4_im: 4 by 4, fp16
- X=[X_re_hi, X_re_lo, X_im_hi, X_im_lo]: m by 4 by Batch*4, fp16
- Result1, Result2: m by 4 by Batch*4, fp32

For batch size = B, length = N input, will do gemms for:

- m = N, Batch = B
- m = N/4, Batch = 4B
- ...
- m = 4, Batch = NB/4

cuBlas is not optimized for slender matrix multiplication (volta_sgemm_fp16_128x64_nn)
## CUTLASS vs cuBlas
### m by 4 * 4 by 4 matrix multiplication

<table>
<thead>
<tr>
<th>m</th>
<th>Batch size</th>
<th>cuBlas (ms)</th>
<th>cutlass (ms)</th>
<th>Mean error</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>1048576</td>
<td>40.7779</td>
<td>13.4457</td>
<td>1.23754e-12</td>
</tr>
<tr>
<td>256</td>
<td>65536</td>
<td>5.10469</td>
<td>3.07621</td>
<td>1.27887e-12</td>
</tr>
<tr>
<td>256</td>
<td>262144</td>
<td>20.4031</td>
<td>12.2688</td>
<td>1.24481e-12</td>
</tr>
<tr>
<td>1024</td>
<td>16384</td>
<td>5.07802</td>
<td>3.00108</td>
<td>1.23879e-12</td>
</tr>
<tr>
<td>1024</td>
<td>65536</td>
<td>20.2993</td>
<td>11.9628</td>
<td>1.24625e-12</td>
</tr>
<tr>
<td>4096</td>
<td>4096</td>
<td>5.08486</td>
<td>3.00046</td>
<td>1.26754e-12</td>
</tr>
<tr>
<td>4096</td>
<td>16384</td>
<td>20.2965</td>
<td>11.882</td>
<td>1.22616e-12</td>
</tr>
<tr>
<td>16384</td>
<td>4096</td>
<td>44.524</td>
<td>11.8838</td>
<td>1.23812e-12</td>
</tr>
</tbody>
</table>
In the Near Future: Radix-2 vs. Radix-4

- Radix-2 algorithms: $2^\nu$ data points
  - \textit{decimation-in-time} (DIT): Simplest + most common form of Cooley-Tukey alg
    - i. DFTs of even- & odd-indexed inputs, repeat recursively ($O(N\log N)$)
  - \textit{Decimation-in-frequency} (DIF): ($O(N\log N)$) -- divide + conquer
    - i. split DFT into 2 summations $[(0 \rightarrow N/2) + (N/2 \rightarrow N)]$
    - ii. Split those split summations into even & odd
    - iii. Repeat recursively

- Currently using radix-4 ($4^\nu$ data pts)

- Why radix-2?
  - a. DFT of identity [2,2] matrix = real matrix (not complex) & exactly representable in FP16
  - b. Use tensor cores to implement it
  - c. \textbf{ALTHOUGH} radix-4 = more efficient when $N = 2^\nu$
Citations

- [https://jakevdp.github.io/blog/2013/08/28/understanding-the-fft/](https://jakevdp.github.io/blog/2013/08/28/understanding-the-fft/)