ECE 256P– AI Master Program

Hardware for Machine Learning

Matrix Multiplication

- Matrix multiplication is essential in many AI workloads, such as transformers.
- Accelerating matrix multiplication results in a huge speedup in training and inference.
- Many optimisations can be applied to code running matrix multiplication.
- Matrix multiplication is a **regular program** and can be efficiently ported to the GPU.
- This lab will cover running several versions of matrix multiplication on the CPU and the GPU and evaluating the impact on performance for the different optimisations.

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \times \begin{pmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{pmatrix}$$

$$= \begin{pmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{pmatrix}$$

Naïve CPU implementation

- The simplest implementation of CPU calculates $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$.
- This requires three nested loops.
- The outer most two loops sweep over all the values of c_{ij}.
- The last inner most loop enables the calculation of the summation $\sum_{k=1}^{n} a_{ik} b_{kj}$.

```
for i in range(A.shape[0]):
  for j in range(B.shape[1]):
    for k in range(A.shape[1]):
      C[i, j] += A[i, k] * B[k, j]
```

Tiled CPU implementation

$$\boxed{\begin{pmatrix} a_{11} & \cdots & a_{1T} \\ \vdots & \ddots & \vdots \\ a_{T1} & \cdots & a_{TT} \end{pmatrix}} \searrow$$

- The tiled CPU version decomposes the matrices into sub-blocks.
- These sub-blocks undergo the multiplication operation $C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$.
- Each multiplication $A_{ik} B_{kj}$ consists of a sub matrix multiplication.
- This implementation improves the cache locality as these blocks can fit within the cache which reduces memory latency.

$$\begin{pmatrix} \boxed{A_{11}} & \cdots & A_{1K} \\ \vdots & \ddots & \vdots \\ A_{K1} & \cdots & A_{KK} \end{pmatrix} \times \begin{pmatrix} B_{11} & \cdots & B_{1K} \\ \vdots & \ddots & \vdots \\ B_{K1} & \cdots & B_{KK} \end{pmatrix}$$

$$= \begin{pmatrix} C_{11} & \cdots & C_{1K} \\ \vdots & \ddots & \vdots \\ C_{K1} & \cdots & C_{KK} \end{pmatrix}$$

```
for i in range(0, A.shape[0], tile_size):
 for j in range(0, B.shape[1], tile_size):
  for k in range(0, A.shape[1], tile_size):
   for ii in range(i, min(i + tile_size, A.shape[0])):
    for jj in range(j, min(j + tile_size, B.shape[1])):
     for kk in range(k, min(k + tile_size, A.shape[1])):
      C[ii, jj] += A[ii, kk] * B[kk, jj]
```

Unrolled loop CPU implementation

- The unrolled loop CPU version unrolls some iterations of the innermost loop. Here, 4 iterations are unrolled.
- The unrolling factor should be a divisor of the matrix size.
- This reduces the number of times the program checks the loop bound, which reduces the number of instructions to execute. It also improves cache efficiency.
- This sacrifices the size of the code (more lines of code) and flexibility in handling different sizes.

```
for i in range(A.shape[0]):
 for j in range(B.shape[1]):
  sum_val = 0
  for k in range(0, A.shape[1], 4):
   sum_val += A[i, k] * B[k, j]
   sum_val += A[i,k+1] * B[k+1, j]
   sum_val += A[i,k+2] * B[k+2, j]
   sum_val += A[i,k+3] * B[k+3, j]
  C[i, j] = sum_val
```

Naïve GPU implementation

- The simplest implementation of GPU calculates $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$.
- Each CUDA thread runs the calculation of c_{ij}.
- The effect of GPU implementations are more visible with large matrices. Here we use 8192 (compared to 32 on CPU!)

```
if (row < N && col < N) {
 float sum = 0.0f;
 for (int k = 0; k < N; ++k) {
  sum += A[row * N + k] * B[k * N + col];
 }
 C[row * N + col] = sum;
}
```

Shared memory GPU implementation

- Using the shared memory to keep data close to the CUDA cores.
- It reduces traffic to the global memory.
- It's similar to caching in the CPU but explicitly managed.
- Shared memory is made of memory banks that hold chunks of data.
- One issue with this method can be bank conflict, where multiple threads try to access the same bank.

```
__shared__ float tileA[TILE_SIZE][TILE_SIZE];
__shared__ float tileB[TILE_SIZE][TILE_SIZE];

int row = blockIdx.y * TILE_SIZE + threadIdx.y;
int col = blockIdx.x * TILE_SIZE + threadIdx.x;

float val = 0.0;
for (int t = 0; t < (N + TILE_SIZE - 1) / TILE_SIZE; ++t) {
          ... // shared memory initialisation
  __syncthreads();

  for (int i = 0; i < TILE_SIZE; ++i)
    val += tileA[threadIdx.y][i] * tileB[i][threadIdx.x];

  __syncthreads();
}

if (row < N && col < N)
  C[row * N + col] = val;
```

Register tiling GPU implementation

- Using the register tiling (in addition to shared memory) keeps data even closer to the CUDA cores.
- Registers are small in size and number, so tiling happens on small tiles of the matrix (2x2 here).
- If more registers are required than the available ones, this causes flushing all the content of all registers to the global memory (register spilling), which significantly reduces performance.

```
__shared__ float tileA[TILE_SIZE][TILE_SIZE];
__shared__ float tileB[TILE_SIZE][TILE_SIZE];

int row = blockIdx.y * TILE_SIZE + threadIdx.y;
int col = blockIdx.x * TILE_SIZE + threadIdx.x;

float val = 0.0;
for (int t = 0; t < (N + TILE_SIZE - 1) / TILE_SIZE; ++t) {
          ... // shared memory initialisation
  __syncthreads();

  // Multiply shared memory tiles
  for (int k = 0; k < TILE_SIZE; ++k) {
    float a0 = tileA[threadIdx.y * 2][k];
    float a1 = tileA[threadIdx.y * 2 + 1][k];
    float b0 = tileB[k][threadIdx.x * 2];
    float b1 = tileB[k][threadIdx.x * 2 + 1];

    c00 += a0 * b0;
    c01 += a0 * b1;
    c10 += a1 * b0;
    c11 += a1 * b1;
}

__syncthreads();
}
  // Write back to global memory
if (row < N && col < N) {
  C[row * N + col] = c00;
  if (col + 1 < N) C[row * N + col + 1] = c01;
  if (row + 1 < N) C[(row + 1) * N + col] = c10;
  if (row + 1 < N && col + 1 < N) C[(row + 1) * N + col + 1] = c11;
}
```

Register tiling GPU implementation

- Using the register tiling (in addition to shared memory) keeps data even closer to the CUDA cores.
- Registers are small in size and number, so tiling happens on small tiles of the matrix (2x2 here).
- If more registers are required than the available ones, this causes flushing all the content of all registers to the global memory (register spilling), which significantly reduces performance.

```
__shared__ float tileA[TILE_SIZE][TILE_SIZE];
__shared__ float tileB[TILE_SIZE][TILE_SIZE];

int row = blockIdx.y * TILE_SIZE + threadIdx.y;
int col = blockIdx.x * TILE_SIZE + threadIdx.x;

float val = 0.0;
for (int t = 0; t < (N + TILE_SIZE - 1) / TILE_SIZE; ++t) {
          ... // shared memory initialisation
  __syncthreads();

  // Multiply shared memory tiles
  for (int k = 0; k < TILE_SIZE; ++k) {
    float a0 = tileA[threadIdx.y * 2][k];
    float a1 = tileA[threadIdx.y * 2 + 1][k];
    float b0 = tileB[k][threadIdx.x * 2];
    float b1 = tileB[k][threadIdx.x * 2 + 1];

    c00 += a0 * b0;
    c01 += a0 * b1;
    c10 += a1 * b0;
    c11 += a1 * b1;
}

__syncthreads();
}
  // Write back to global memory
if (row < N && col < N) {
  C[row * N + col] = c00;
  if (col + 1 < N) C[row * N + col + 1] = c01;
  if (row + 1 < N) C[(row + 1) * N + col] = c10;
  if (row + 1 < N && col + 1 < N) C[(row + 1) * N + col + 1] = c11;
}
```

Optimised PyTorch implementations

- PyTorch provides optimised implementation of matrix multiplication on CPU, TPU, and GPU.
- This can be achieved by changing the runtime and applying the appropriate changes to the code.

```
cpu = torch.device("cpu")
... // Init instructions
C = torch.matmul(A, B)
```

```
import torch_xla.core.xla_model as xm

tpu = xm.xla_device()
... // Init instructions
C = torch.matmul(A, B)
```

```
gpu = torch.device("cuda")

tpu = xm.xla_device()
... // Init instructions
C = torch.matmul(A, B)
```