Dynamic Shared Memory

- Shared memory can be **dynamically allocated** if programmer does not know how much the need beforehand

- Declaration:
  - In kernel, by adding a C **extern** keyword in front of the shared memory declaration and omitting the size of the array in the declaration.

```
extern __shared__ As_Bs[];
```

Dynamic Shared Memory

- Shared memory can be **dynamically allocated** if programmer does not know how much the need beforehand

- Declaration:
  - In kernel, by adding a C **extern** keyword in front of the shared memory declaration and omitting the size of the array in the declaration.

```
extern __shared__ As_Bs[];
```

  - When using dynamic shared memory with CUDA, there is one and only one pointer passed to the kernel, which defines the start of the requested/allocated area in bytes: the declarations for A_s and B_s need to be merged into **one dynamically allocated array**.

```
  size_t size =
calculate_appropriate_SM_usage(devProp.sharedMemPerBlock,
...);

  matrixMulKernel<<<dimGrid,dimBlock,size>>>(A_s, B_s, Pd,
Width, size/2, size/2);
```

Dynamic Shared Memory

- Configuration:
  - At runtime, when we call the kernel, we can dynamically configure the amount of shared memory to be used for each block according to the device query result and supply that as a third parameter to the kernel call.

```
  size_t size =
calculate_appropriate_SM_usage(devProp.sharedMemPerBlock,
...);

  matrixMulKernel<<<dimGrid,dimBlock,size>>>(A, B, Pd,
Width, size/2, size/2);
```

- Note that: size_t is a built-in type for declaring a variable to hold the size information for dynamically allocated data structures. The size is expressed in number of bytes.

- In the tiled matrix-multiplication, for a 4x4 tile, we have a size of 2x4x4x4=128 bytes to accommodate both A_s and B_s

Dynamic Shared Memory

- **Part** of the tiled matrix multiplication kernel with dynamically sized shared memory usage.

```
01    #define TILE_WIDTH 16
02    __global__ void matrixMulKernel(float* A, float* B, float* P, int Width,
                                      unsigned Adz_sz  , unsigned Bdz_sz  ) {
03
04       extern __shared__ char float As_Bs[ ];
05
06       float * A_s  = (float *) As_Bs ;
07       float * B_s  = (float *) As_Bs +Adz_sz;
```

Tiling on CPU

- Tiling also works for CPU
  - No scratchpad memory, but relies on caches
  - Cache is sufficiently reliable because there are fewer threads running on the core and the cache is larger

```
for(unsigned int rowTile = 0; rowTile < N/TILE_DIM; ++rowTile) {
    for(unsigned int colTile = 0; colTile < N/TILE_DIM; ++colTile) {
        for(unsigned int iTile = 0; iTile < N/TILE_DIM; ++iTile) {
            for (unsigned int row = rowTile*TILE_DIM; row < (rowTile + 1)*TILE_DIM; ++row) {
                for (unsigned int col = colTile*TILE_DIM; col < (colTile + 1)*TILE_DIM; ++col) {
                    float sum = 0.0f;
                    for(unsigned int i = iTile*TILE_DIM; i < (iTile + 1)*TILE_DIM; ++i) {
                        sum += A[row*N + i]*B[i*N + col];
                    }
                    if(iTile == 0) {
                        C[row*N + col] = sum;
                    } else {
                        C[row*N + col] += sum;
                    }
                }
            }
        }
    }
}
```