Data Parallelism and Matrix Multiplication

1. Data Parallelism
   - matrix-matrix multiplication
   - CUDA program structure
2. Code for Matrix-Matrix Multiplication
   - linear address system for 2-dimensional array
   - defining the kernel
   - the main program
   - using `threadIdx.x` and `threadIdx.y`
3. Examining Performance
   - counting flops
4. using CUDA.jl and Metal.jl
   - a plain matrix matrix multiplication in Julia

MCS 572 Lecture 19
Introduction to Supercomputing
Jan Verschelde, 9 October 2024

Data Parallelism and Matrix Multiplication

1. Data Parallelism
   - matrix-matrix multiplication
   - CUDA program structure
2. Code for Matrix-Matrix Multiplication
   - linear address system for 2-dimensional array
   - defining the kernel
   - the main program
   - using `threadIdx.x` and `threadIdx.y`
3. Examining Performance
   - counting flops
4. using CUDA.jl and Metal.jl
   - a plain matrix matrix multiplication in Julia

data parallelism

Many applications process large amounts of data.

Data parallelism refers to the property where many arithmetic operations can be safely performed on the data simultaneously.

Consider the multiplication of matrices A and B: $C = A \cdot B$, with

$$A = [a_{i,j}] \in \mathbb{R}^{n \times m}, \quad B = [b_{i,j}] \in \mathbb{R}^{m \times p}, \quad C = [c_{i,j}] \in \mathbb{R}^{n \times p}.$$

$c_{i,j}$ is the inner product of the ith row of A with the jth column of B:

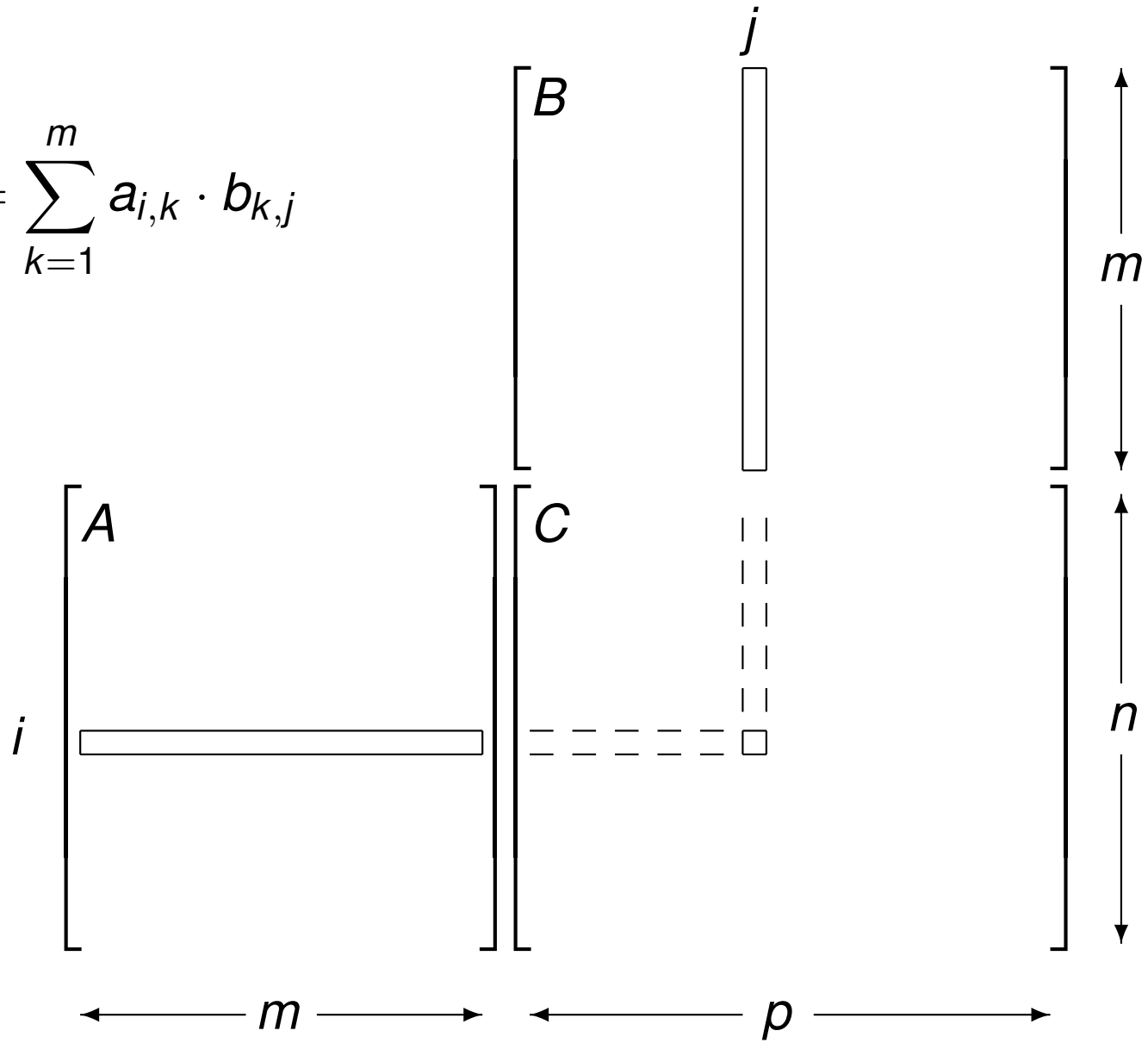
$$c_{i,j} = \sum_{k=1}^{m} a_{i,k} \cdot b_{k,j}.$$

All $c_{i,j}$'s can be computed independently from each other.

For $n = m = p = 1,024$ we have 1,048,576 inner products.

data parallelism in matrix multiplication

$$c_{i,j} = \sum_{k=1}^{m} a_{i,k} \cdot b_{k,j}$$



matrix-matrix multiplication on a GPU

Code for a device (the GPU) is defined in functions using the keyword `__global__` before the function definition.

Data parallel functions are called *kernels*.

Kernel functions generate a large number of threads.

In matrix-matrix multiplication, the computation can be implemented as a kernel where each thread computes one element in the result matrix.

To multiply two 1,024-by-1,024 matrices, the kernel using one thread to compute one element generates 1,048,576 threads when invoked.

CUDA threads are much lighter weight than CPU threads: they take very few cycles to generate and schedule thanks to efficient hardware support whereas CPU threads may require thousands of cycles.

Data Parallelism and Matrix Multiplication

1. Data Parallelism
   - matrix-matrix multiplication
   - CUDA program structure
2. Code for Matrix-Matrix Multiplication
   - linear address system for 2-dimensional array
   - defining the kernel
   - the main program
   - using `threadIdx.x` and `threadIdx.y`
3. Examining Performance
   - counting flops
4. using CUDA.jl and Metal.jl
   - a plain matrix matrix multiplication in Julia

CUDA program structure

A CUDA program consists of several phases, executed on

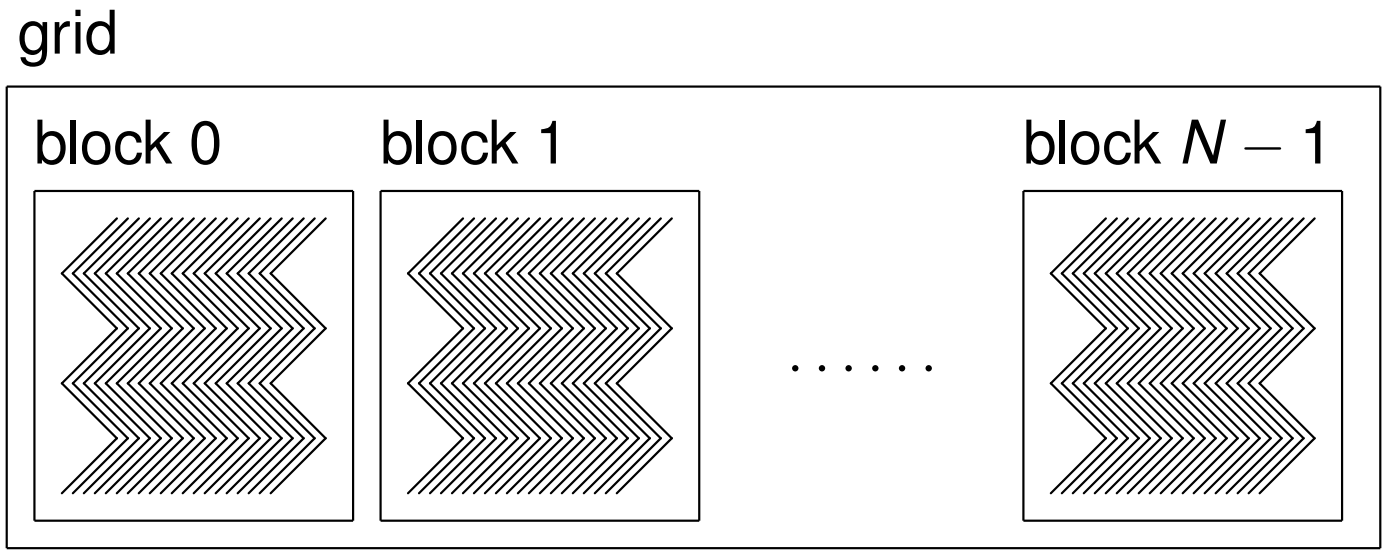
- the host: if no data parallelism,
- the device: for data parallel algorithms.

The NVIDIA C compiler `nvcc` separates phases at compilation:

- Code for the host is compiled on host's standard C compilers and runs as ordinary CPU process.
- The device code is written in C with keywords for data parallel functions and further compiled by `nvcc`.

execution of a CUDA program

```
CPU code
kernel<<<numb_blocks,numb_threads_per_block>>>(args)
CPU code
```



stages in a CUDA program

For the matrix multiplication $C = A \cdot B$:

1. Allocate device memory for A, B, and C.
2. Copy A and B from the host to the device.
3. Invoke the kernel to have device do $C = A \cdot B$.
4. Copy C from the device to the host.
5. Free memory space on the device.

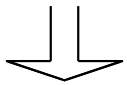
Data Parallelism and Matrix Multiplication

1. Data Parallelism
   - matrix-matrix multiplication
   - CUDA program structure
2. Code for Matrix-Matrix Multiplication
   - linear address system for 2-dimensional array
   - defining the kernel
   - the main program
   - using `threadIdx.x` and `threadIdx.y`
3. Examining Performance
   - counting flops
4. using CUDA.jl and Metal.jl
   - a plain matrix matrix multiplication in Julia

linear address system

Consider a 3-by-5 matrix stored row-wise (as in C):

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$



$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

We will store a matrix as a one dimensional array.

generating a random matrix

```
#include <stdlib.h>

__host__ void randomMatrix ( int n, int m, float *x, int mode )
/*
 * Fills up the n-by-m matrix x with random
 * values of zeroes and ones if mode == 1,
 * or random floats if mode == 0. */
{
   int i,j,r;
   float *p = x;

   for(i=0; i<n; i++)
      for(j=0; j<m; j++)
      {
         if(mode == 1)
            r = rand() % 2;
         else
            r = ((float) rand())/RAND_MAX;
         *(p++) = (float) r;
      }
}
```

writing a matrix

```
#include <stdio.h>

__host__ void writeMatrix ( int n, int m, float *x )
/*
 * Writes the n-by-m matrix x to screen. */
{
   int i,j;
   float *p = x;

   for(i=0; i<n; i++,printf("\n"))
      for(j=0; j<m; j++)
         printf(" %d", (int)*(p++));
}
```

Data Parallelism and Matrix Multiplication

1. Data Parallelism
   - matrix-matrix multiplication
   - CUDA program structure
2. Code for Matrix-Matrix Multiplication
   - linear address system for 2-dimensional array
   - defining the kernel
   - the main program
   - using `threadIdx.x` and `threadIdx.y`
3. Examining Performance
   - counting flops
4. using CUDA.jl and Metal.jl
   - a plain matrix matrix multiplication in Julia

assigning inner products to threads

Consider a 3-by-4 matrix A and a 4-by-5 matrix B:

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$

$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$	$b_{0,4}$
$b_{1,0}$	$b_{1,1}$	$b_{1,2}$	$b_{1,3}$	$b_{1,4}$
$b_{2,0}$	$b_{2,1}$	$b_{2,2}$	$b_{2,3}$	$b_{2,4}$
$b_{3,0}$	$b_{3,1}$	$b_{3,2}$	$b_{3,3}$	$b_{3,4}$

$c_{0,0}$	$c_{0,1}$	$c_{0,2}$	$c_{0,3}$	$c_{0,4}$	$c_{1,0}$	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	$c_{1,4}$	$c_{2,0}$	$c_{2,1}$	$c_{2,2}$	$c_{2,3}$	$c_{2,4}$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

The `i = blockIdx.x*blockDim.x + threadIdx.x` determines what entry in $C = A \cdot B$ will be computed:

- the row index in C is `i` divided by 5 and
- the column index in C is the remainder of `i` divided by 5.

the kernel function

```
__global__ void matrixMultiply
  ( int n, int m, int p, float *A, float *B, float *C )
/*
 * Multiplies the n-by-m matrix A
 * with the m-by-p matrix B into the matrix C.
 * The i-th thread computes the i-th element of C. */
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   C[i] = 0.0;
   int rowC = i/p;
   int colC = i%p;
   float *pA = &A[rowC*m];
   float *pB = &B[colC];
   for(int k=0; k<m; k++)
   {
      pB = &B[colC+k*p];
      C[i] += (*(pA++))*(*pB);
   }
}
```

Data Parallelism and Matrix Multiplication

1. Data Parallelism
   - matrix-matrix multiplication
   - CUDA program structure
2. Code for Matrix-Matrix Multiplication
   - linear address system for 2-dimensional array
   - defining the kernel
   - the main program
   - using `threadIdx.x` and `threadIdx.y`
3. Examining Performance
   - counting flops
4. using CUDA.jl and Metal.jl
   - a plain matrix matrix multiplication in Julia

running the program

```
$ ./matmatmul 3 4 5 1
a random 3-by-4 0/1 matrix A :
 1 0 1 1
 1 1 1 1
 1 0 1 0
a random 4-by-5 0/1 matrix B :
 0 1 0 0 1
 0 1 1 0 0
 1 1 0 0 0
 1 1 0 1 0
the resulting 3-by-5 matrix C :
 2 3 0 1 1
 2 4 1 1 1
 1 2 0 0 1
$
```

the main program — command line arguments

```
int main ( int argc, char*argv[] )
{
   if(argc < 4)
   {
      printf("call with 3 arguments :\n");
      printf("dimensions n, m, and p\n");
   }
   else
   {
      int n = atoi(argv[1]);     /* number of rows of A */
      int m = atoi(argv[2]);     /* number of columns of A */
                                 /* and number of rows of B */
      int p = atoi(argv[3]);     /* number of columns of B */
      int mode = atoi(argv[4]); /* 0 no output, 1 show output */
      if(mode == 0)
         srand(20140331)
      else
         srand(time(0));
```

allocating memories

```
float *Ahost = (float*)calloc(n*m,sizeof(float));
float *Bhost = (float*)calloc(m*p,sizeof(float));
float *Chost = (float*)calloc(n*p,sizeof(float));
randomMatrix(n,m,Ahost,mode);
randomMatrix(m,p,Bhost,mode);
if(mode == 1)
{
   printf("a random %d-by-%d 0/1 matrix A :\n",n,m);
   writeMatrix(n,m,Ahost);
   printf("a random %d-by-%d 0/1 matrix B :\n",m,p);
   writeMatrix(m,p,Bhost);
}
/* allocate memory on the device for A, B, and C */
float *Adevice;
size_t sA = n*m*sizeof(float);
cudaMalloc((void**)&Adevice,sA);
float *Bdevice;
size_t sB = m*p*sizeof(float);
cudaMalloc((void**)&Bdevice,sB);
float *Cdevice;
size_t sC = n*p*sizeof(float);
cudaMalloc((void**)&Cdevice,sC);
```

copying and kernel invocation

```
      /* copy matrices A and B from host to the device */
      cudaMemcpy(Adevice,Ahost,sA,cudaMemcpyHostToDevice);
      cudaMemcpy(Bdevice,Bhost,sB,cudaMemcpyHostToDevice);

      /* kernel invocation launching n*p threads */
      matrixMultiply<<<n*p,1>>>(n,m,p,
                                Adevice,Bdevice,Cdevice);

      /* copy matrix C from device to the host */
      cudaMemcpy(Chost,Cdevice,sC,cudaMemcpyDeviceToHost);
      /* freeing memory on the device */
      cudaFree(Adevice); cudaFree(Bdevice); cudaFree(Cdevice);
      if(mode == 1)
      {
         printf("the resulting %d-by-%d matrix C :\n",n,p);
         writeMatrix(n,p,Chost);
      }
   }
   return 0;
}
```

Data Parallelism and Matrix Multiplication

1. Data Parallelism
   - matrix-matrix multiplication
   - CUDA program structure
2. Code for Matrix-Matrix Multiplication
   - linear address system for 2-dimensional array
   - defining the kernel
   - the main program
   - using `threadIdx.x` and `threadIdx.y`
3. Examining Performance
   - counting flops
4. using CUDA.jl and Metal.jl
   - a plain matrix matrix multiplication in Julia

using `threadIdx.x` and `threadIdx.y`

Instead of a one dimensional organization of the threads in a block we can make the (i, j)-th thread compute $c_{i,j}$.

The main program is then changed into

```
/* kernel invocation launching n*p threads */
dim3 dimGrid(1,1);
dim3 dimBlock(n,p);
matrixMultiply<<<dimGrid,dimBlock>>>
   (n,m,p,Adevice,Bdevice,Cdevice);
```

The above construction creates a grid of one block.
The block has $n \times p$ threads:

- `threadIdx.x` will range between 0 and $n - 1$, and
- `threadIdx.y` will range between 0 and $p - 1$.

the new kernel

```
__global__ void matrixMultiply
  ( int n, int m, int p, float *A, float *B, float *C )
/*
 * Multiplies the n-by-m matrix A
 * with the m-by-p matrix B into the matrix C.
 * The (i,j)-th thread computes the (i,j)-th element of C.
{
   int i = threadIdx.x;
   int j = threadIdx.y;
   int ell = i*p + j;
   C[ell] = 0.0;
   float *pB;
   for(int k=0; k<m; k++)
   {
      pB = &B[j+k*p];
      C[ell] += A[i*m+k]*(*pB);
   }
}
```

Data Parallelism and Matrix Multiplication

1. Data Parallelism
   - matrix-matrix multiplication
   - CUDA program structure
2. Code for Matrix-Matrix Multiplication
   - linear address system for 2-dimensional array
   - defining the kernel
   - the main program
   - using `threadIdx.x` and `threadIdx.y`
3. Examining Performance
   - counting flops
4. using CUDA.jl and Metal.jl
   - a plain matrix matrix multiplication in Julia

performance analysis

Performance is often expressed in terms of flops.

- 1 flops = one floating-point operation per second;
- use `perf`: Performance analysis tools for Linux
- run the executable, with `perf stat`

```
$ perf stat ./matmatmul0 1024 1024 1024 0
```

- with the events following the `-e` flag
  we count the floating-point operations.

```
$ perf stat -e fp_arith_inst_retired.scalar_single \
  ./matmatmul0 1024 1024 1024 0
```

Executables are compiled with the option `-O2`.

on one Intel Xeon E5-2699v4 Broadwell core

on pascal

```
$ perf stat -e fp_arith_inst_retired.scalar_single \
  ./matmatmul0 768 768 768 0

 Performance counter stats for './matmatmul0 768 768 768 0':

       905,969,664      fp_arith_inst_retired.scalar_single:u

       1.039681742 seconds time elapsed

       1.036818000 seconds user
       0.002999000 seconds sys

$
```

Did 905,969,664 operations in 1.037 seconds:

$$\Rightarrow (905,969,664/1.037)/(2^{30}) = 0.81\text{GFlops}.$$

performance on the P100

```
$ perf stat -e fp_arith_inst_retired.scalar_single
  ./matmatmul1 768 768 768 0

 Performance counter stats for './matmatmul1 768 768 768 0':

             6,123      fp_arith_inst_retired.scalar_single:u

       0.207871212 seconds time elapsed

       0.039441000 seconds user
       0.167880000 seconds sys


$
```

Drop from 1.037 seconds to 0.28 seconds is not impressive.

The dimension 768 is too small for the GPU to be able to improve much.

running for larger dimensions

```
$ perf stat -e fp_arith_inst_retired.scalar_single
  ./matmatmul0 4096 4096 4096 0

 Performance counter stats for './matmatmul0 4096 4096 4096 0':

   137,438,953,472      fp_arith_inst_retired.scalar_single:u

     416.494934403 seconds time elapsed

     416.466205000 seconds user
       0.047003000 seconds sys


$ perf stat  ./matmatmul1 4096 4096 4096 0
```

shows

```
   0.569705088 seconds time elapsed
```

speedup and performance

Seconds time elapsed on CPU: 416.495.
Seconds time elapsed on GPU: 0.570.

Speedup: 416.495/0.570 = 730.

Counting flops, f = 137,438,953,472, the performance is:

- $t_{\text{cpu}} = 415.495$: $f/t_{\text{cpu}}/(2^{30}) = 0.3$ GFlops.
- $t_{\text{gpu}} = 0.570$: $f/t_{\text{gpu}}/(2^{30}) = 224.5$ GFlops.

The performance is far from optimal, both for CPU and GPU.

To be continued ...

Data Parallelism and Matrix Multiplication

1. Data Parallelism
   - matrix-matrix multiplication
   - CUDA program structure
2. Code for Matrix-Matrix Multiplication
   - linear address system for 2-dimensional array
   - defining the kernel
   - the main program
   - using `threadIdx.x` and `threadIdx.y`
3. Examining Performance
   - counting flops
4. using CUDA.jl and Metal.jl
   - a plain matrix matrix multiplication in Julia

a plain matrix matrix multiplication

```
using CUDA

function matmul!(C, A, B)
    i = threadIdx().x
    j = threadIdx().y
    for k=1:size(A, 2)
        @inbounds C[i, j] = C[i, j] + A[i, k]*B[k, j]
    end
end

dim = 2^2
A_h = rand(dim, dim)
B_h = rand(dim, dim)
C_h = A_h * B_h
A_d = CuArray(A_h)
B_d = CuArray(B_h)
C_d = CuArray(zeros(dim, dim))

@cuda threads=(dim, dim) matmul!(C_d, A_d, B_d)

println(C_h)
println(C_d)
```

on a macOS GPU using Apple's Metal framework

The equivalent code for execution on an M1 MacBook air:

```
using Metal

function matmul!(C, A, B)
    threadpos = thread_position_in_grid_2d()
    i = threadpos[1]
    j = threadpos[2]
    for k=1:size(A, 2)
        @inbounds C[i, j] = C[i, j] + A[i, k]*B[k, j]
    end
end
```

Observe the `threadpos` to work
with the two dimensional grid of threads.

the code continues

The GPU in an M1 MacBook Air does not support 64-bit floats.
Use `Float32` instead of the default `Float64`:

```
dim = 2^2
A_h = rand(Float32, dim, dim)
B_h = rand(Float32, dim, dim)
C_h = A_h * B_h
A_d = MtlArray(A_h)
B_d = MtlArray(B_h)
C_d = MtlArray(zeros(Float32, dim, dim))

@metal threads=(dim, dim) matmul!(C_d, A_d, B_d)

println(C_h)
println(C_d)
```

Observe the launching of the kernel.

summary and exercises

We covered more of chapter 3 in the book of Kirk & Hwu.

1. The `perf` was illustrated on on older computer.
   Redo the illustrations on `ampere`.
2. Modify `matmatmul0.c` and `matmatmul1.cu` to work with doubles instead of floats. Examine the performance.
3. Modify `matmatmul2.cu` to use double indexing of matrices, e.g.: `C[i][j] += A[i][k]*B[k][j]`.
4. Compare the performance of `matmatmul1.cu` and `matmatmul2.cu`, taking larger and larger values for n, m, and p. Which version scales best?
5. Compare the performance of `matmatmul1.cu` and `mmmulcuda2.jl`. Does the Julia code achieve the same performance as the C CUDA program?