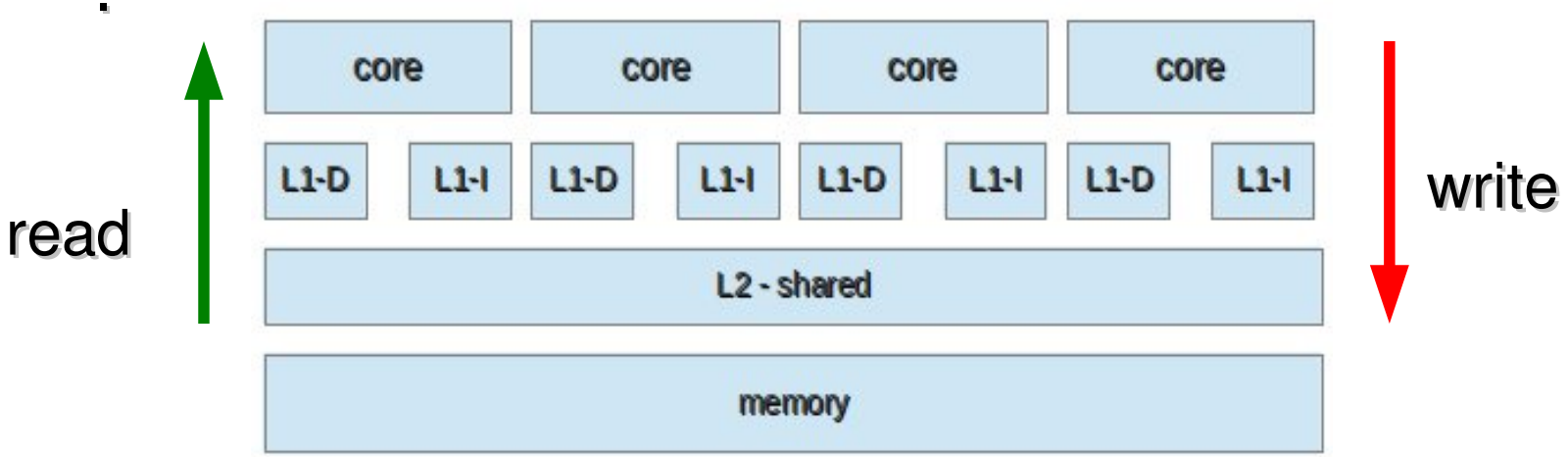
Multi and Many-core parallel programming exercises

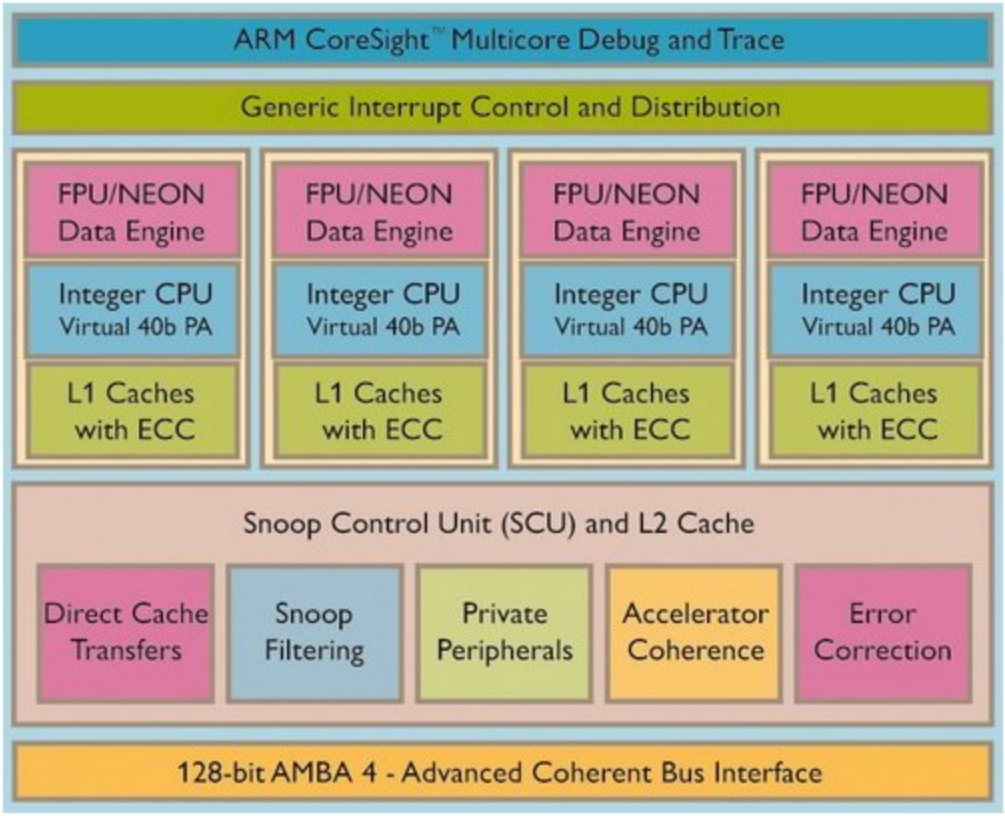
ARM: multi-core SMP architecture



Discussion topic: shared memory coherence (snooping)

ARM: multi-core SMP architecture

ARM Cortex-15 MPCore



OpenMP – operational mode

openMP works with threads that share the same memory address space with the other threads created for a single program.

```
export OMP_NUM_THREADS=8

#pragma omp ...
{
...code... // parallel region
}
```

Ex1: OpenMP – operational mode

Write an **openMP** based program that prints the numbers of the threads.
Use **tid = omp_get_thread_num();**
to get the thread number.

```
#include "stdio.h"  #include <omp.h>
int main(int argc, char *argv[])
{
  #pragma omp parallel
  { ..
```

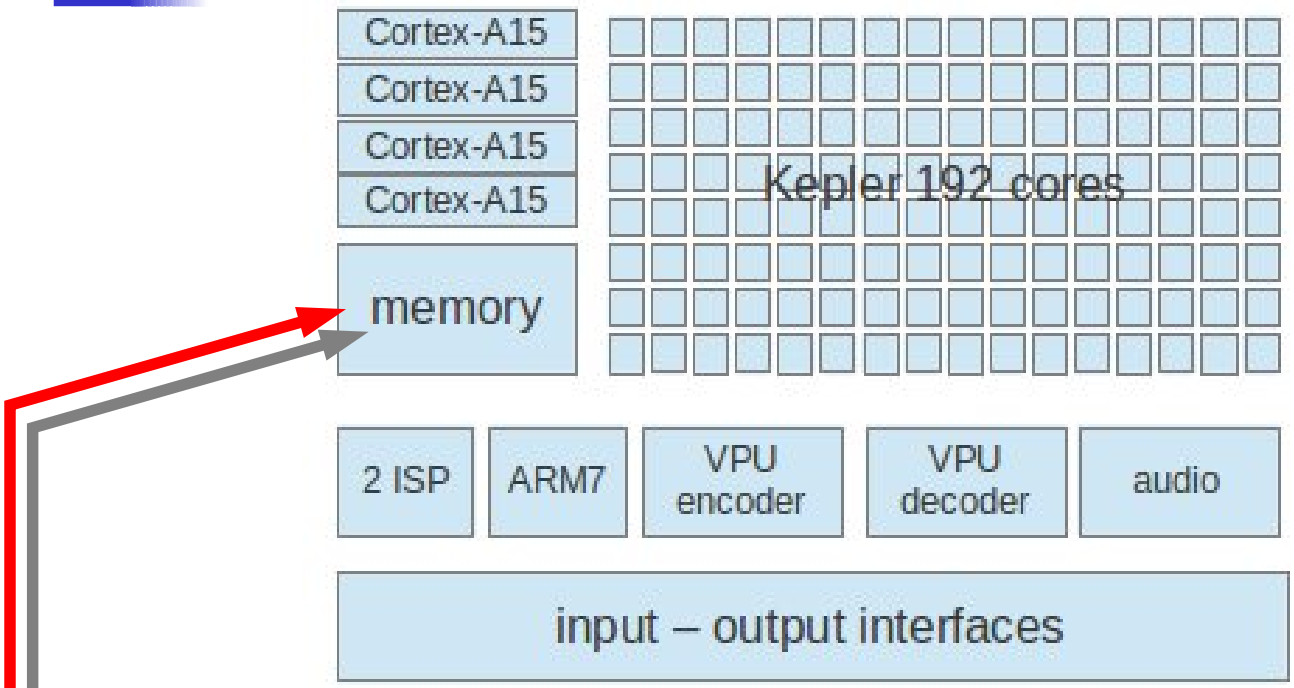
Ex2: OpenMP – operational mode

Write an **openMP** based program that adds two 16-element vectors. Each element is an integer number. Initialise the vectors by their index values:

```
v1[i]=i; v2[i]=i;
```

```
#include "stdio.h"  #include <omp.h>
int main(int argc, char *argv[])
{
  #pragma omp parallel
  { ..
```

Tegra K1: many-core architecture



CPU Cortex-15 MPCore + GPU Kepler-192 cores

shared Global memory

CUDA – operational mode

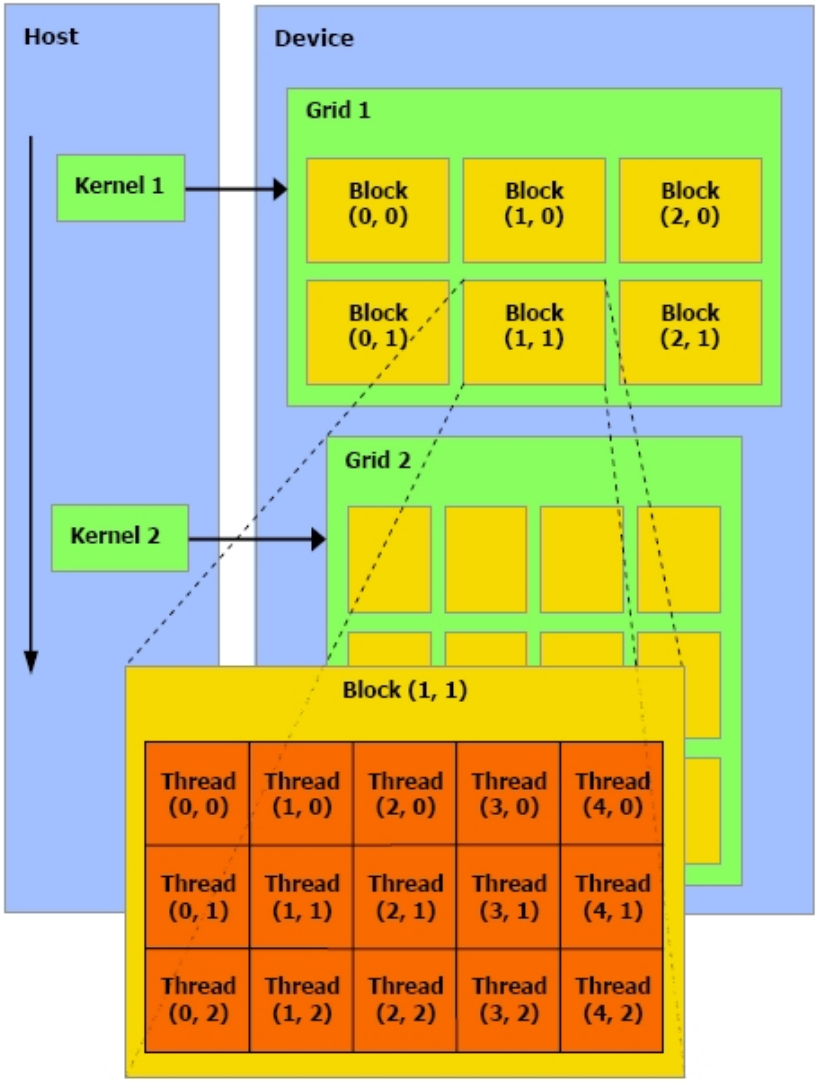
The **grid** is built from **blocks**.

Each **block** is built from **threads**.

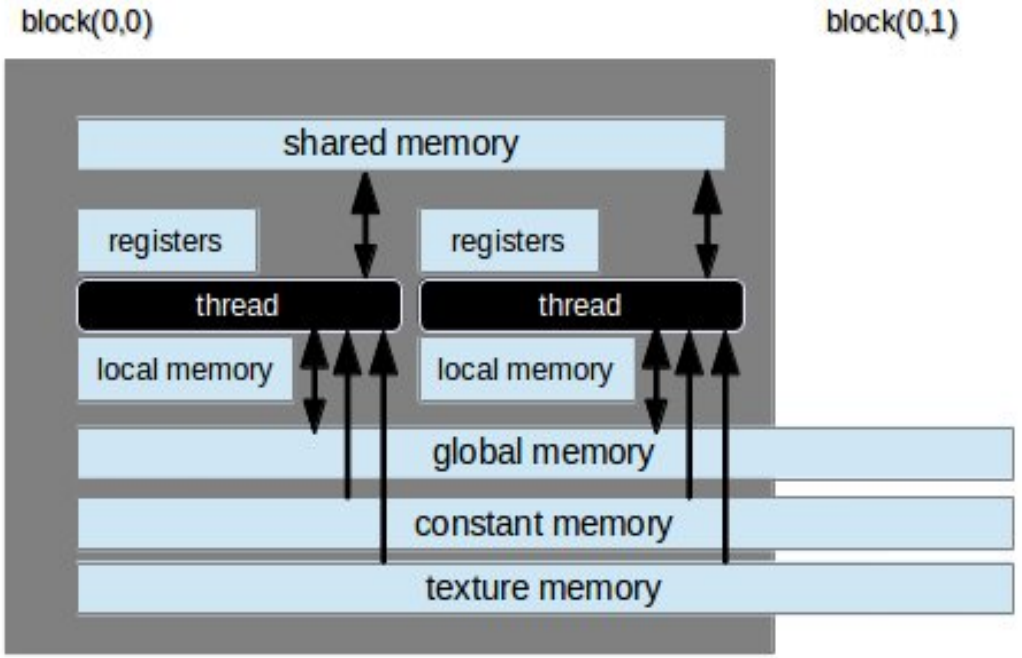
Questions :

How many threads are illustrated in this figure ?

What are their dimmensions ?



CUDA – memories

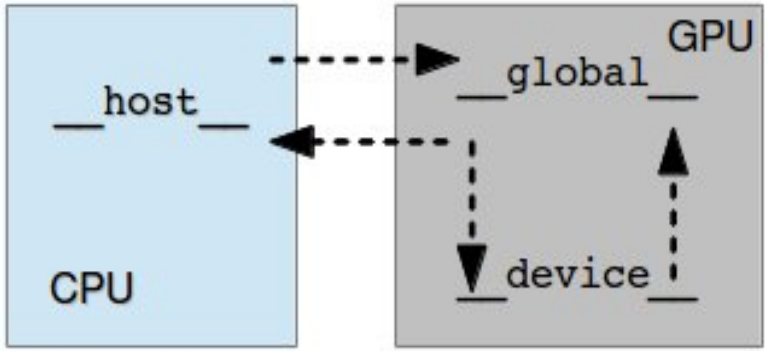


Questions :

How do the threads share memories ?

What is the difference between the independent and the embedded GPU concerning the use of Global memory ?

CUDA – kernels



Kernel automatic variables :

`threadIdx, blockIdx, blockDim, gridDim`

In corresponding dimension : **`.x`**, **`.y`** or **`.z`**

```
int tix ;
tix = threadIdx.x+blockIdx.x*blockDim.x
```

CUDA – kernel call

A kernel call includes the threads and blocks parameters:

```
kernel <<<nb, nt>>> (arguments);
```

Example 1 (give the number of threads per block):

```
#define vsize 64
MAXaddVect<<<2,vsize/2>>>(Cv1,Cv2,Cres);
```

Example 2 (give the number of threads per block):

```
#define MAX 512
const int BLOCKSIZE = 16;
dim3 dimBlock(BLOCKSIZE,BLOCKSIZE);
dim3 dimGrid(MAX/dimBlock.x,MAX/dimBlock.y);
kernel<<<dimGrid,dimBlock>>>(devPtr,MAX);
```

Ex3: CUDA – adding vectors

Given the following **`main()`** program, **write** the **`CUDA`** kernel :

```
#include <stdio.h>
int main()
{ int i=0;
float v1[]={1,2,3,4,5,6,7,8,9,10};
float v2[]={1.0,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8,1.9};
int memsize = sizeof(v1);
int vsize = memsize/sizeof(float);
float res[vsize];
float* Cv1; cudaMalloc((void **)&Cv1,memsize);
float* Cv2; cudaMalloc((void **)&Cv2,memsize);
float* Cres; cudaMalloc((void **)&Cres,memsize);
cudaMemcpy(Cv1,v1,memsize,cudaMemcpyHostToDevice);
cudaMemcpy(Cv2,v2,memsize,cudaMemcpyHostToDevice);

addVect<<<2,vsize/2>>>(Cv1,Cv2,Cres);  // 2 blocks

cudaMemcpy(res,Cres,memsize,cudaMemcpyDeviceToHost);
printf("res= { ");
for(i=0;i<vsize;i++)  { printf("%2.2f ", res[i]); } printf("}\n");
}
```

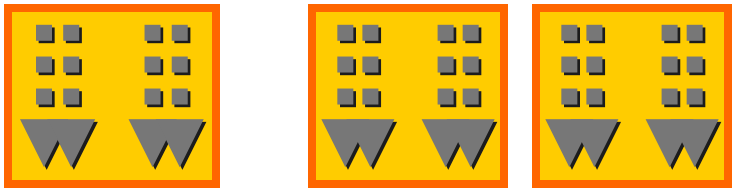
Ex4: CUDA – adding vectors

Rewrite the **`main()`** program using **Zero Copy** :

```
cudaSetDeviceFlags(cudaDeviceMapHost); // enable Zero Copy

float* h_in  = NULL; // Host Arrays
float* h_out = NULL;

// Allocate host memory using CUDA allocation calls
cudaHostAlloc((void **)&h_in,  sizeIn,  cudaHostAllocMapped);
cudaHostAlloc((void **)&h_out, sizeOut, cudaHostAllocMapped);

float *d_out, *d_in; // Device arrays

// Get device pointer from host memory. No allocation or memcpy
cudaHostGetDevicePointer((void **)&d_in,  (void *) h_in , 0);
cudaHostGetDevicePointer((void **)&d_out, (void *) h_out, 0);

kernel<<<blocks, threads>>>(d_out, d_in);
// No need to copy d_out back
```

Ex4: CUDA – adding vectors

```
__global__ void vecAdd(float *A,float *B,float *C)
{
int i = blockIdx.x * blockDim.x + threadIdx.x;
C[i] = A[i] + B[i];
}
int main()
{
   vecAdd<<<nBlocks, nThreadsPerBlocks>>>(A, B, C);
}
```



blockIdx.x

N blocs avec N threads

Ex5: CUDA – Matrix Multiplication

The steps of the program (with Zero Copy):

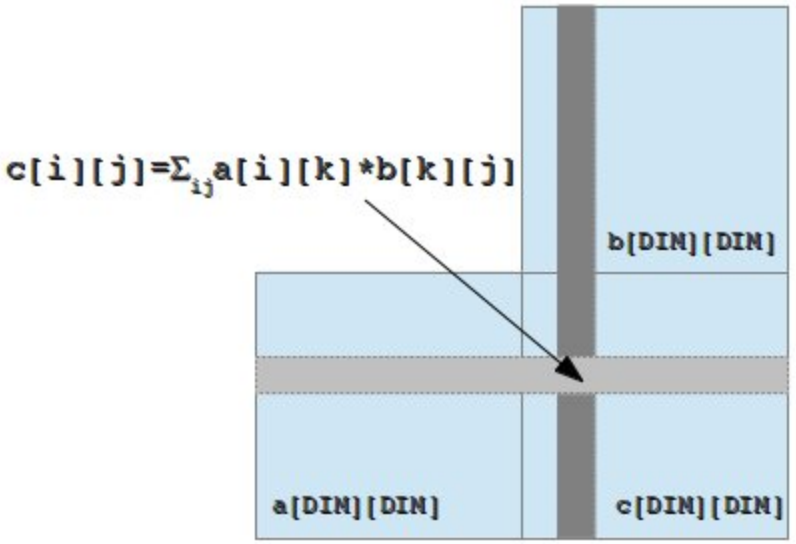
1. **CPU** : square matrices allocation M, N & P
2. **CPU** : square matrices initialization M, N
3. **GPU** : Calculation of M & N product, result in P
4. **CPU** : showing/verifying the result P
5. **CPU** : freeing the memory

$$\mathbf{C}_{ik} = (\mathbf{A}\,\mathbf{B})_{ik} = \sum_{j=1}^{N} A_{ij}B_{jk}$$

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 3 & 4 \\ 0 & 0 \end{bmatrix}$$

Ex5: CUDA – Matrix Multiplication

```
#define Width 512   // corresponds to DIM
__global__ void
  matrix_mul(float* dev_A,float* dev_B,float* dev_C,int Width)
  {   // 2D thread ID
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  float Pvalue =0;
  for(int k=0;k<Width;++k)
  {
    float Ael=dev_A[ty*Width+k];
    float Bel=dev_B[k*Width+tx];
    Pvalue += Ael*Bel;
  }
  dev_C[ty*Width+tx]=Pvalue;
}
```



Question: How many threads are deployed in this example and what are the calculation steps in each thread ?

CUDA events

The execution time of a task (eg. kernel) may be measured via the mechanism of **events**.

Below we define the **`start`** and **`stop`** events and we measure the **elapsed** time.

```
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start,0);

// the code to be evaluated

cudaEventRecord(stop,0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsedTime,start,stop);
```

Ex6 : MatMult execution time

To the previous example, add the evaluation of the execution time.

```
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start,0);

// MatMult kernel

cudaEventRecord(stop,0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsedTime,start,stop);
```

Summary

- **SMP** architecture - example of ARM Cortex-15 MPCore
- **openMP** operational mode
- **openMP** - adding vectors with multiple threads
- Tegra K1 (GPU) – massively parralel architecture
- **CUDA** operational mode
- **CUDA** kernels
- **CUDA** – adding vectors with many threads
- **CUDA** – matrix multiplication & performance evaluation