



# JAPAN-KOREA HPC WINTER SCHOOL 2014

## Accelerated Computing 1: GPGPU Programming and Computing

**Hyungon Ryu 柳賢坤**

hryu@nvidia.com

**NVIDIA Korea | PSG SA**

# Agenda of Contents

- Heterogenous HPC
  - GPU vs. CPU
  - GPGPU
  - simple GPU architecture
- How to Accelerate GPU
  - CUDA Libraries
  - Directives (OpenACC)
  - CUDA
- CUDA Examples
  - CUDA Converting ( transpose)
  - pyCUDA ( image processing)
  - SAXPY/DGEMM (cuBLAS example)
  - BHTE(Bio-Heat Transfer Equation / Laplace )
- Tips for CUDA Optimization
  - Latency Reduction vs. Latency Hiding
  - Memory Bound vs. Arithmetic Bound (BHTE case)

# Heterogenous HPC (이기종 HPC)

- Antonym of Homogenous HPC

- Homogenous HPC
  - X86 CPU Cluster : Homogenous
  - Power Cluster : Homogenous

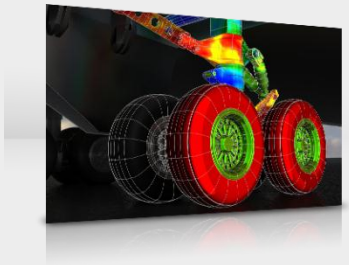
- Heterogenous HPC
  - Cluster with IBM Cell
  - Cluster with ASIC
  - Cluster with FPGA
  - Cluster with NVIDIA GPU ← GPGPU
  - Cluster with AMD GPU ← GPGPU
  - Cluster with Intel Xeon Phi ← GPGPU (Larrabee was GPU)

# Purpose of GPU( Graphic Processing Units)

## Real-Time Graphic Rendering

**Game with DirectX**

**CAD with OpenGL**




**Geforce**


**Quadro**

# GPGPU and CUDA


- GPGPU stands for *General-Purpose* computation *on Graphics Processing Units*, also known as *GPU Computing*. Graphics Processing Units (GPUs) are high-performance many-core processors capable of very high computation and data throughput. Once specially designed for computer graphics and difficult to program, today's GPUs are general-purpose parallel processors with support for accessible programming interfaces and industry-standard languages such as C.

- CUDA stands for *Compute Unifed Device Architecture.* CUDA™ is a parallel computing platform and programming model invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU)

- GPGPU with Cg, OpenGL , DirectX , sh, Brook, RapidMind, PeakStream, Brook++, CAL, CTM, CUDA, OpenGL Compute, DirectXCompute, MS AMP, OpenCL

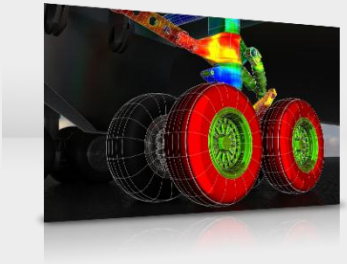
# Purpose of GPU ( Graphic Processing Units)

## Real-Time Graphic Rendering

### Game with DirectX


Geforce

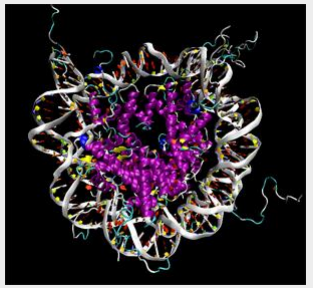
### CAD with OpenGL

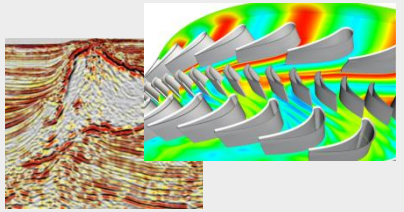



Quadro

## GPGPU

### HPC with CUDA

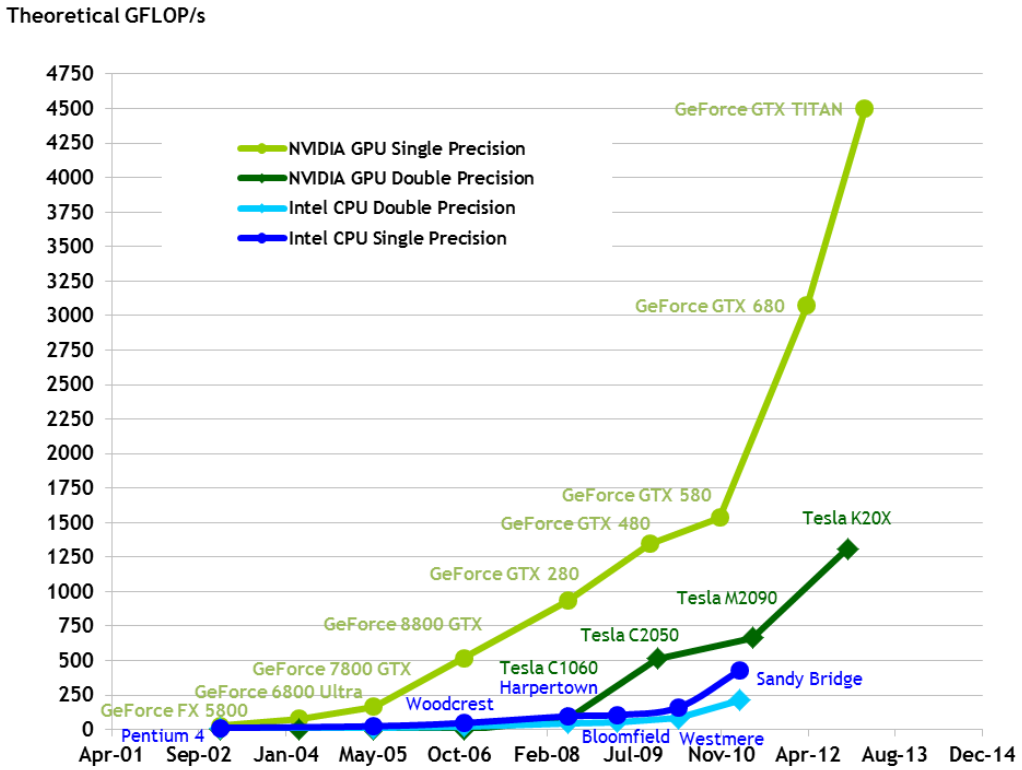




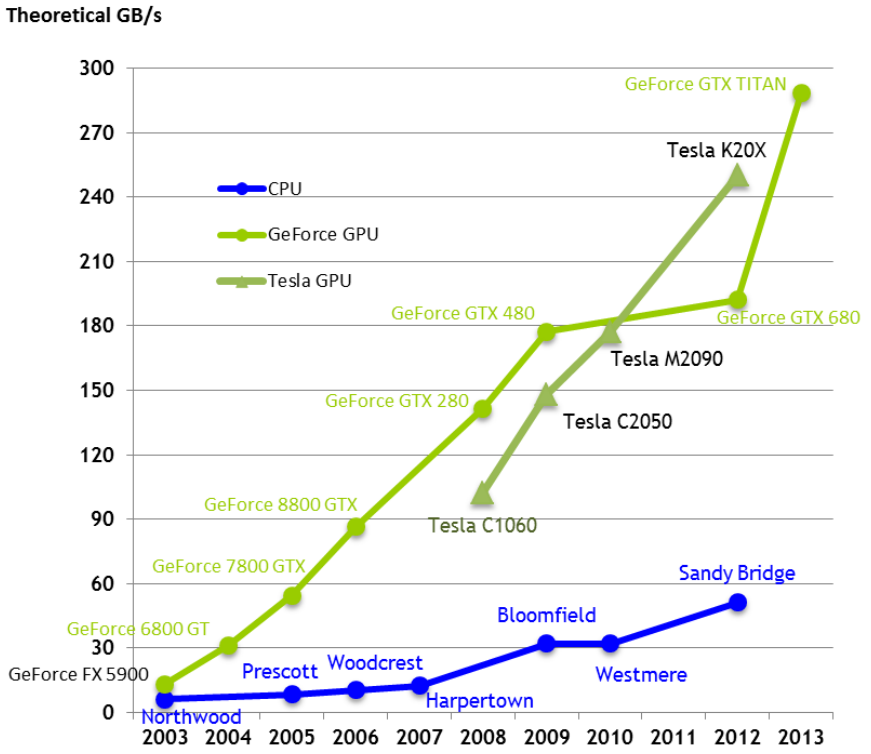

Tesla

# GPU vs CPU Version1

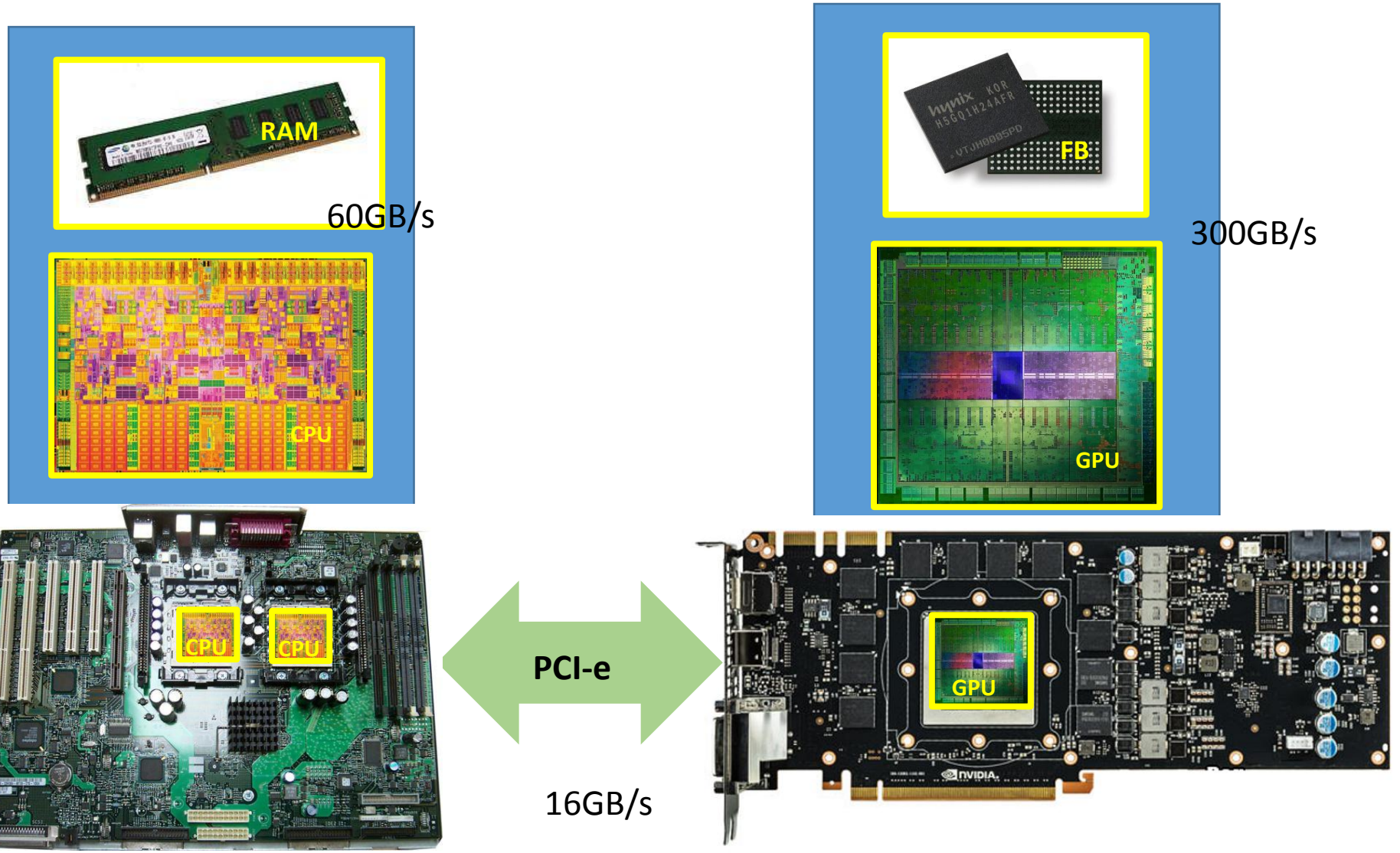
Horse Power



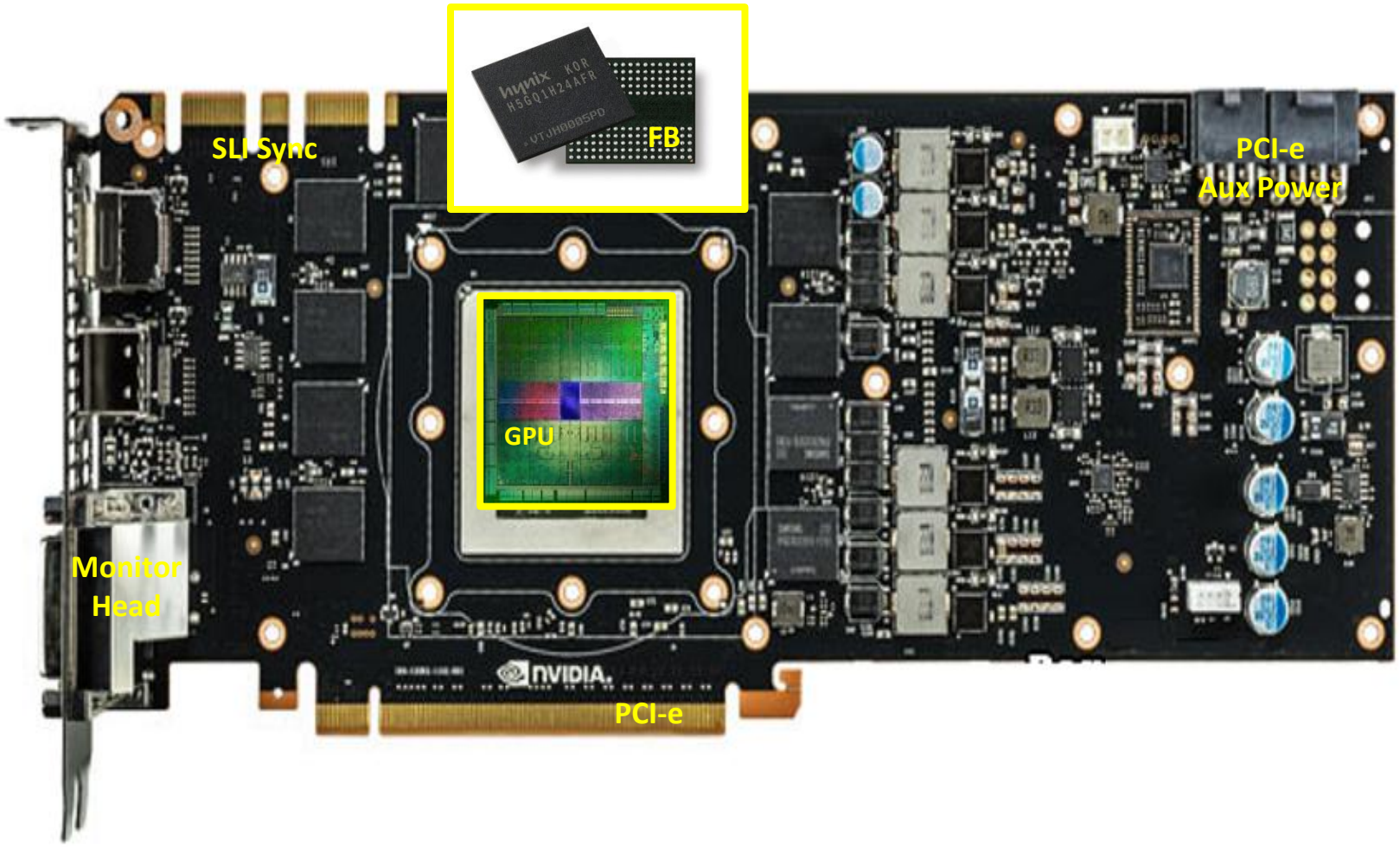

Memory Bandwidth



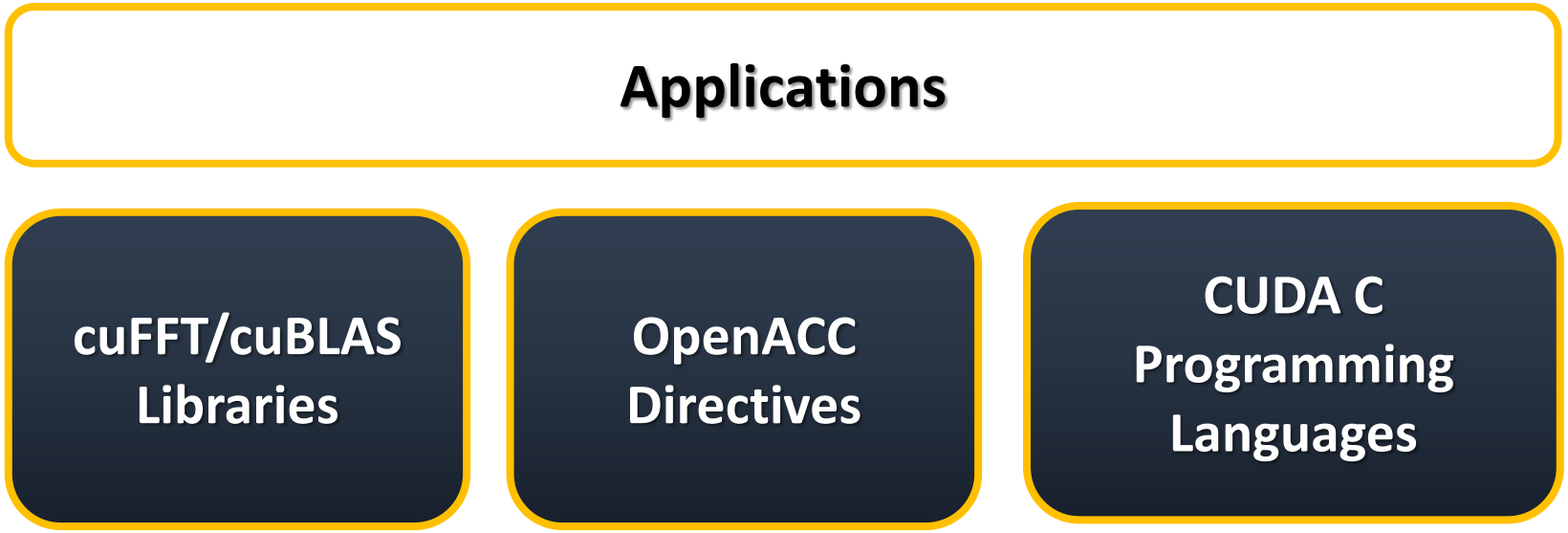
# GPU Accelerator



# GPU board



# How to  Parallelize with GPU



# Compare CUDA programming

## Serial

Algorithm

serial Programming
- Compile
- Debugging
- Optimize/CPU profile

Release

## MPI parallel

Algorithm

serial Programming
- Compile
- Debugging

MPI Parallel Programming
- CPU Profile
- Parallelize
- Compile
- Debugging [totalview]
- Optimize/MPI profile

Release

## CUDA parallel

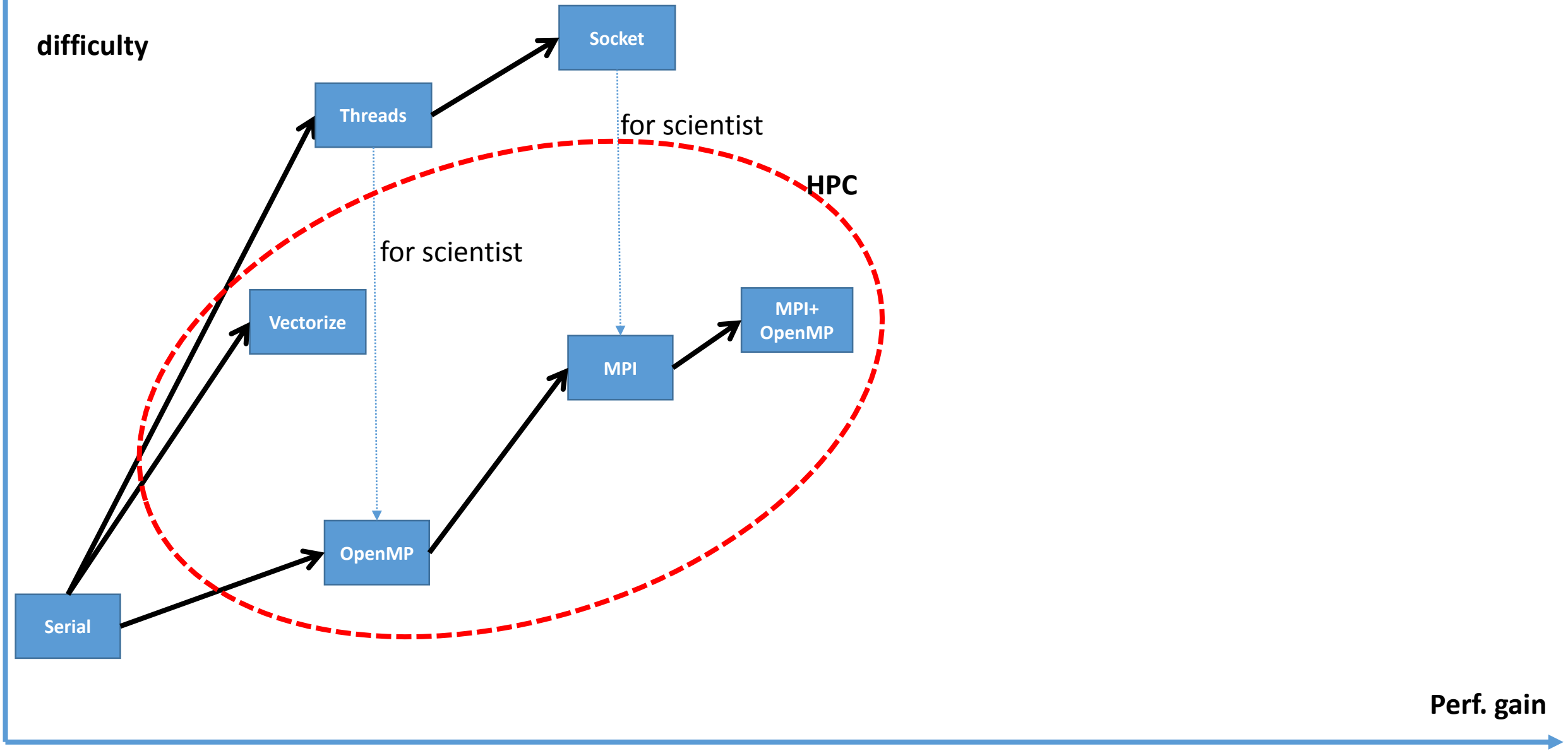
Algorithm

serial Programming
- Compile
- Debugging

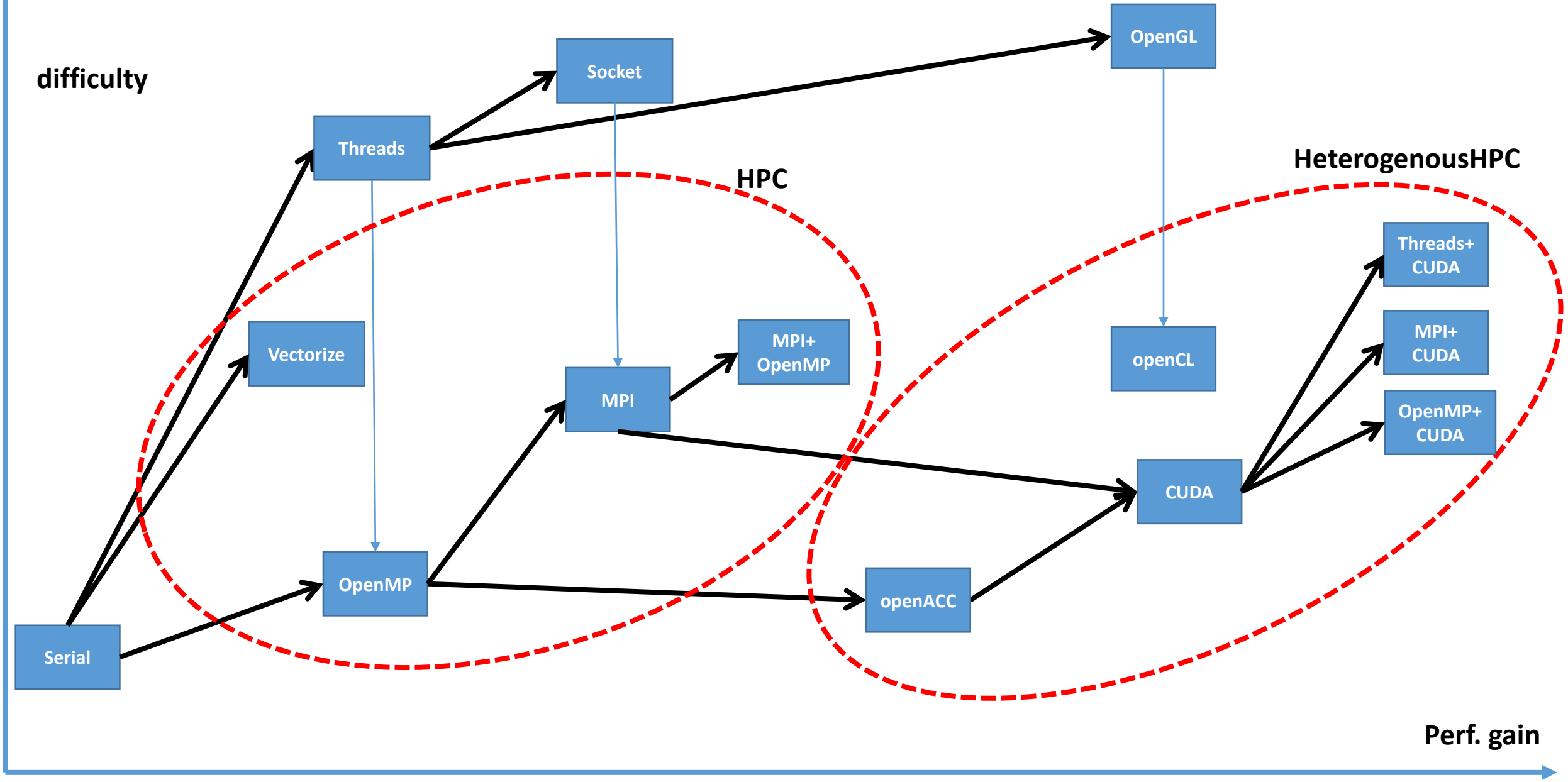
CUDA Parallel Programming
- CPU Profile
- Parallelize
- Compile
- Debugging [cuda-gdb]
- Optimize/CUDA profile

Release

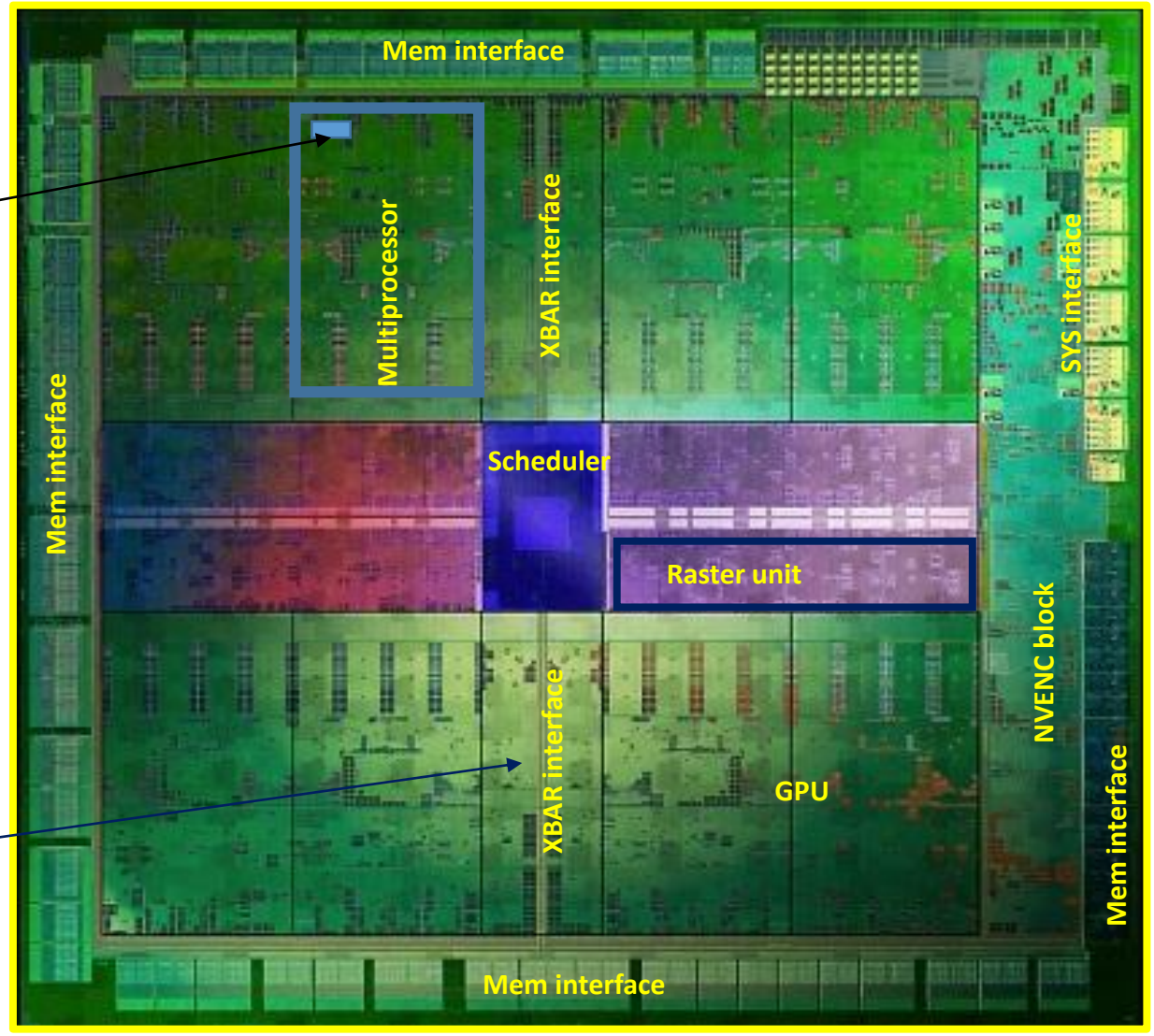
# Difficulty of Parallel Programming



# Difficulty of Parallel Programming



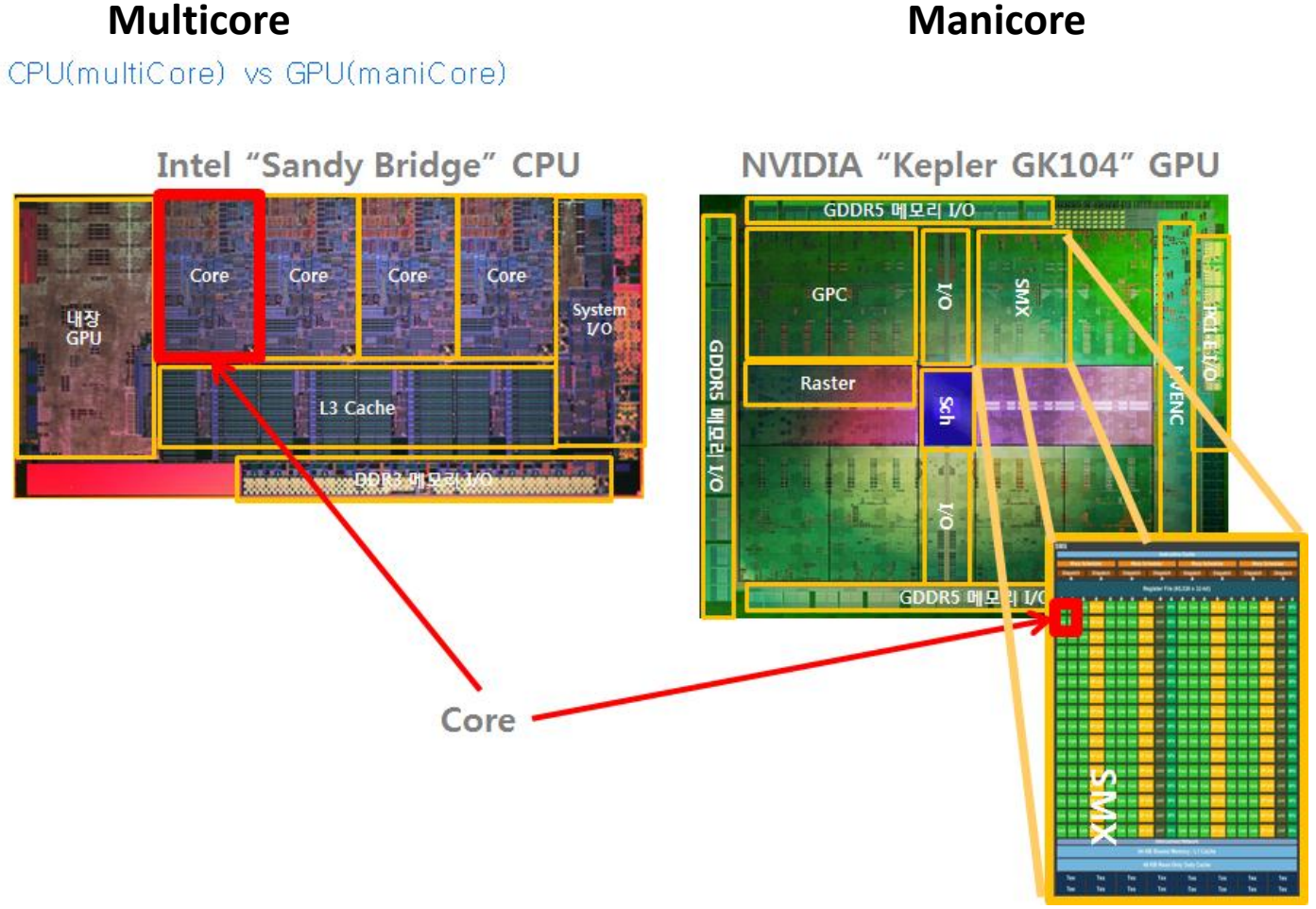
# Die photo of GK104 GPU



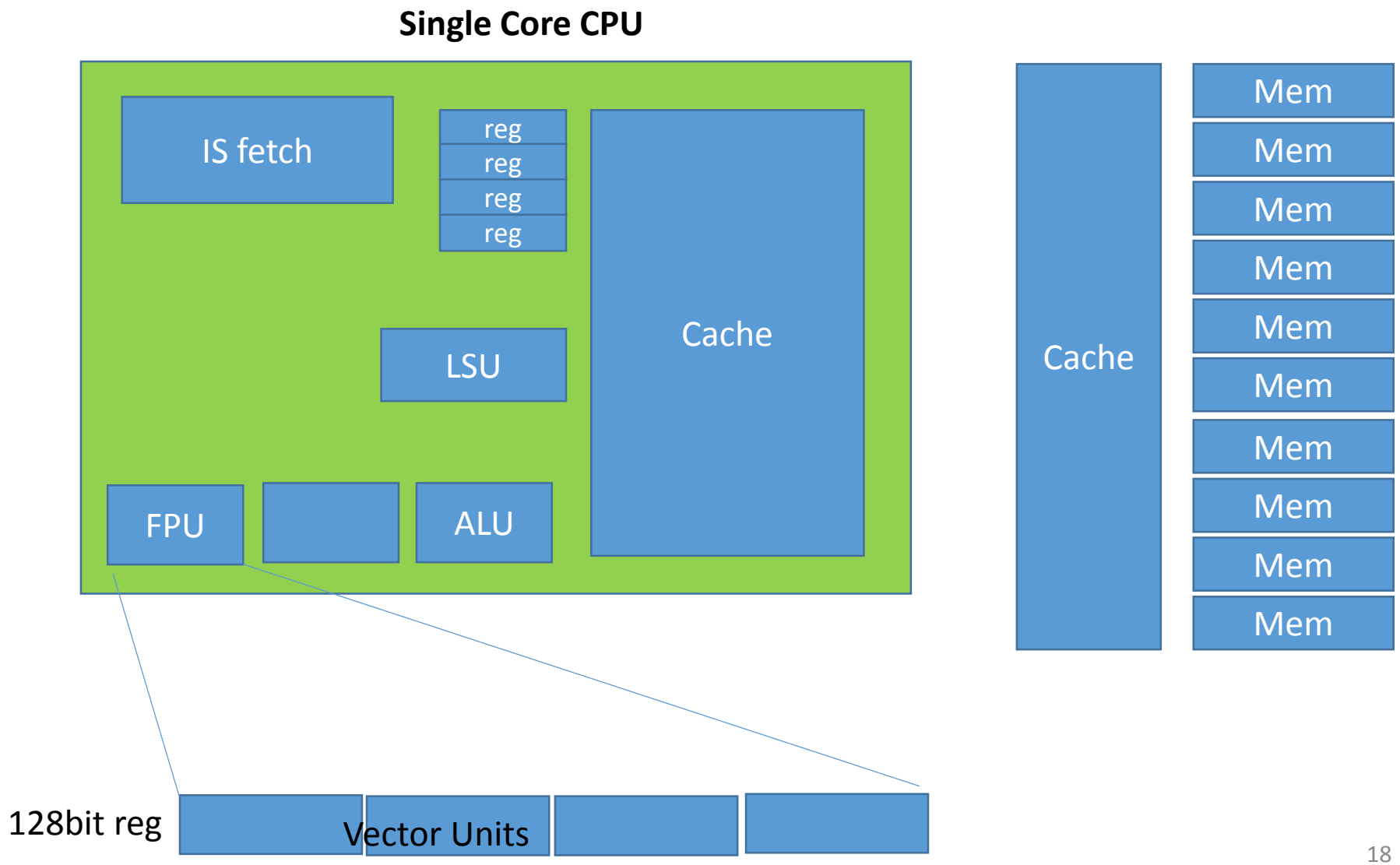

GK104

# CPU vs GPU version2

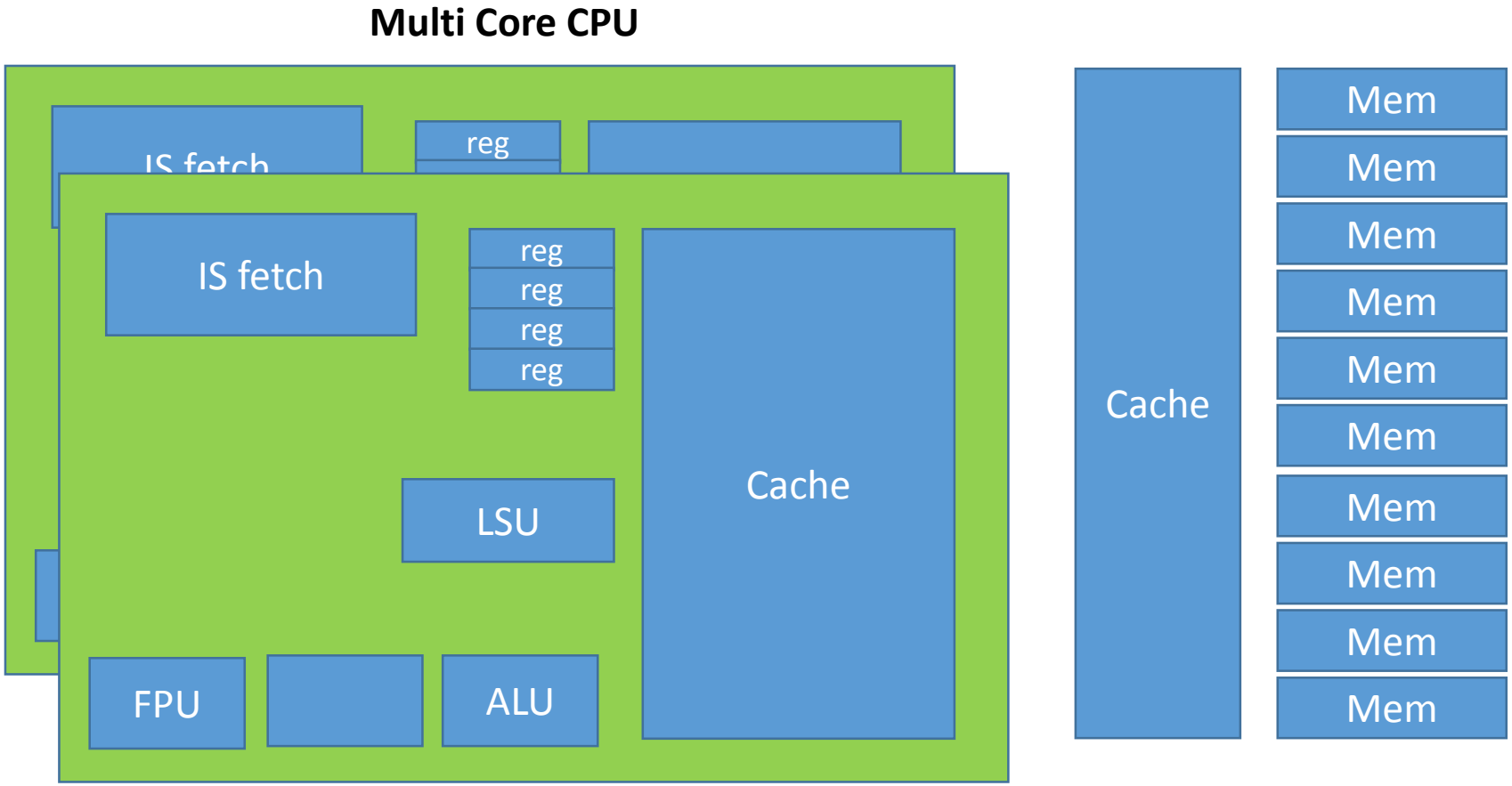
**Multicore** **Manicore**



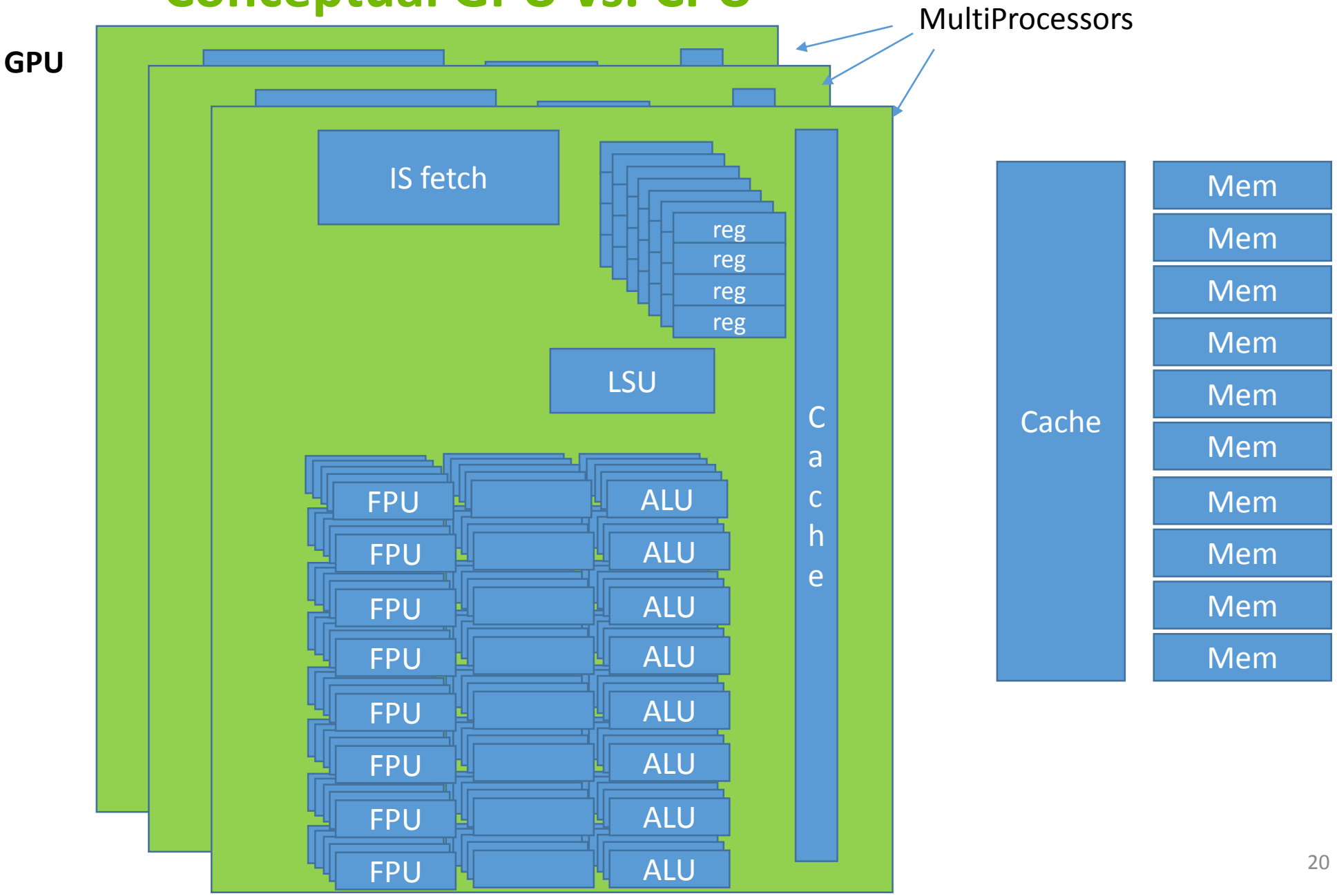
# Conceptual GPU vs. CPU



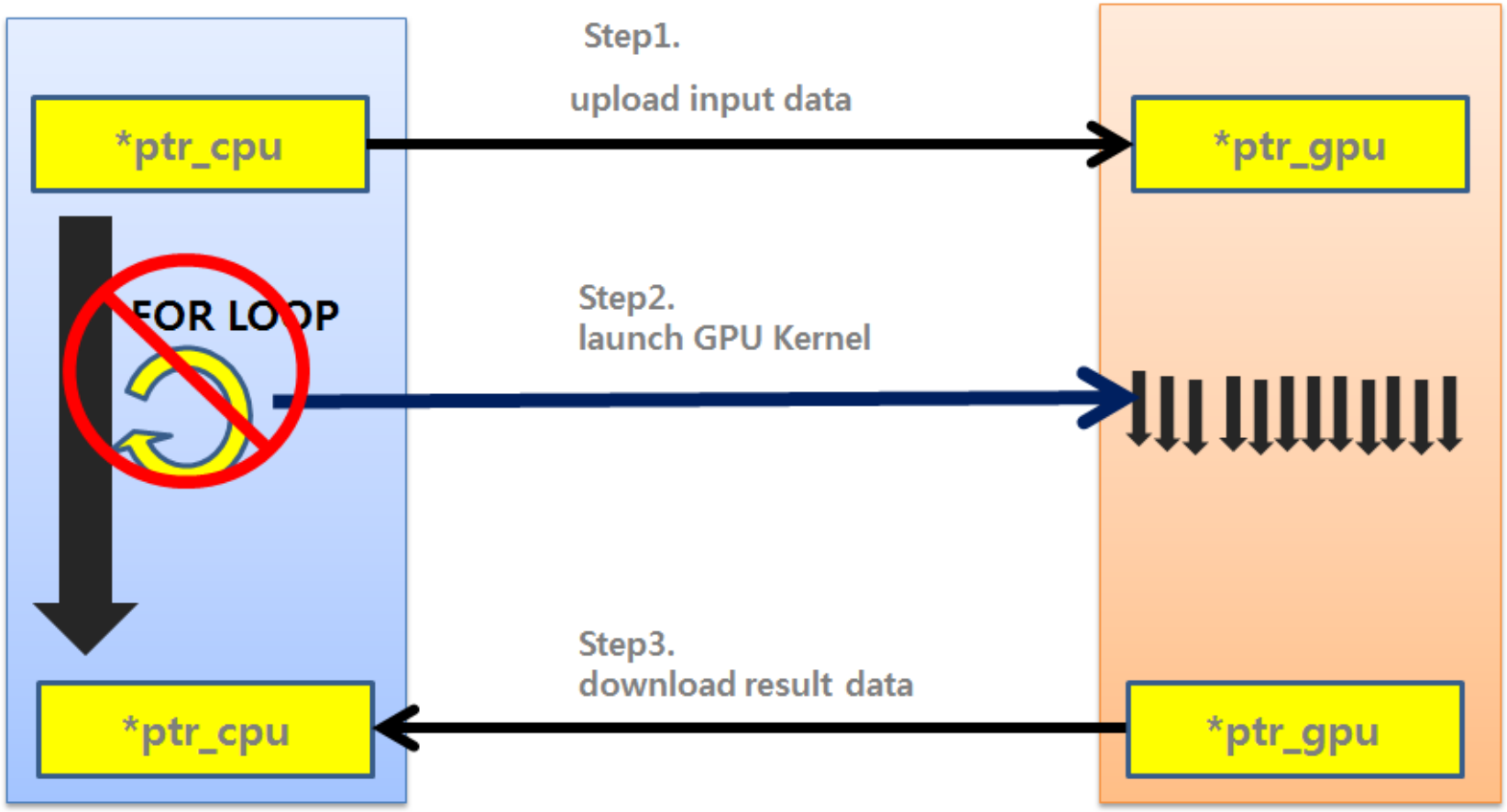
# Conceptual GPU vs. CPU



# Conceptual GPU vs. CPU

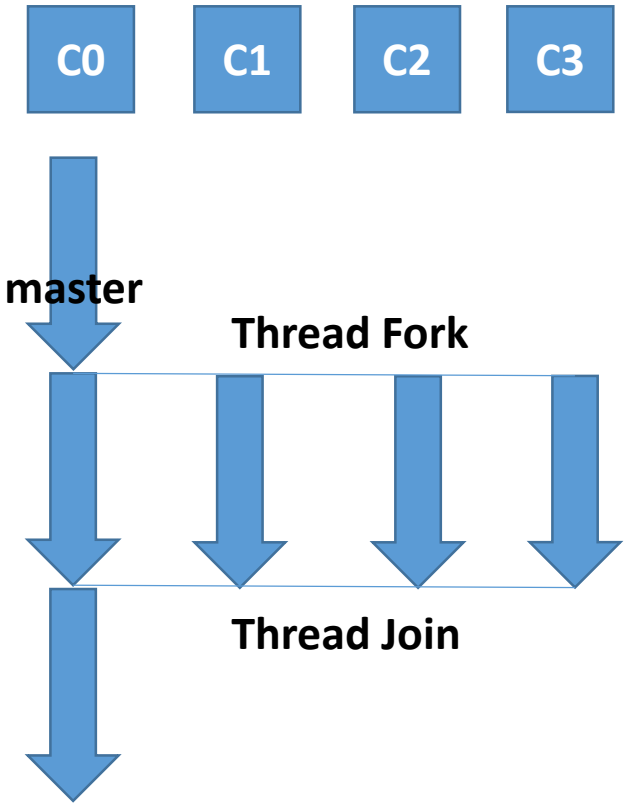


# Simple CUDA Programming Model

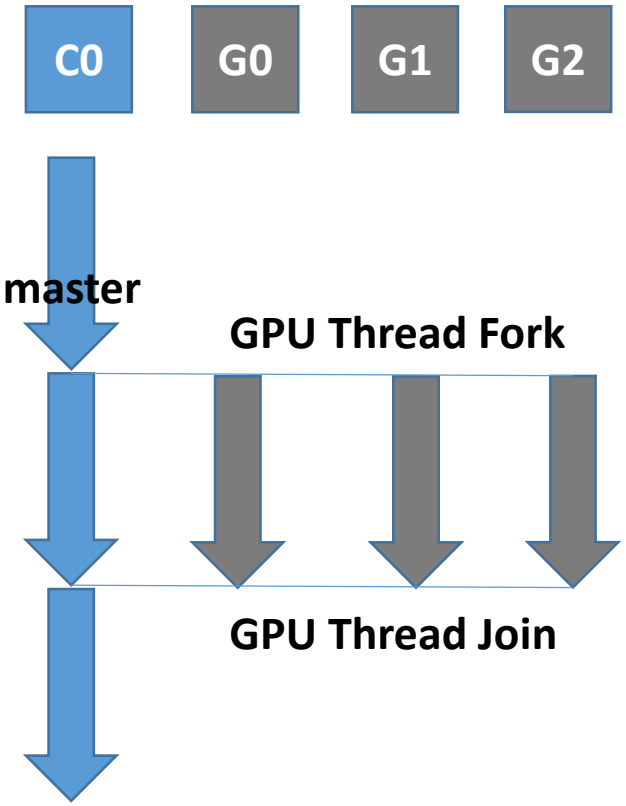


# Fork-Join Model

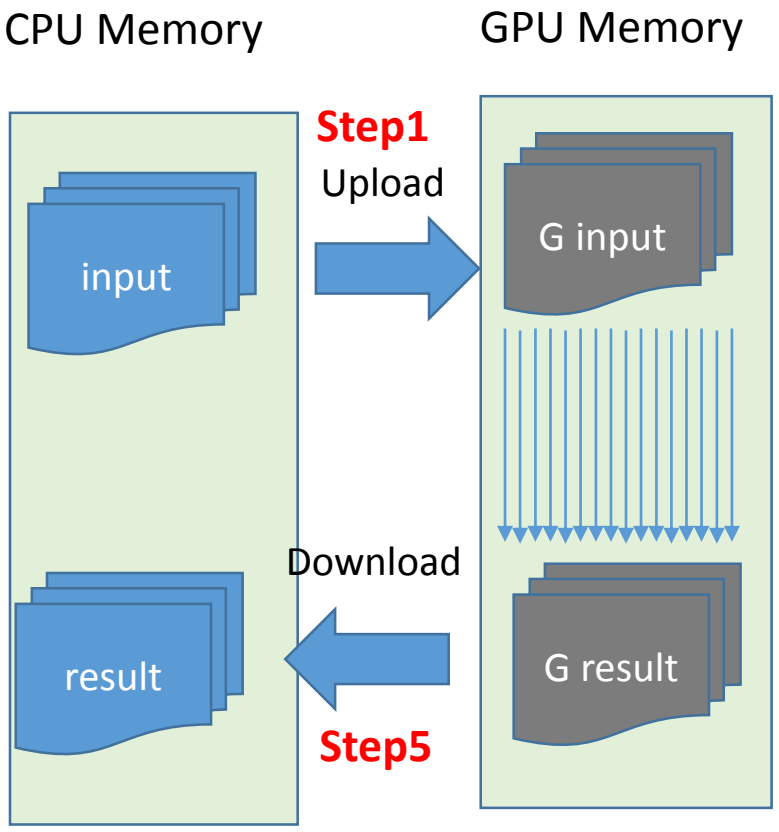
## OpenMP on CPU

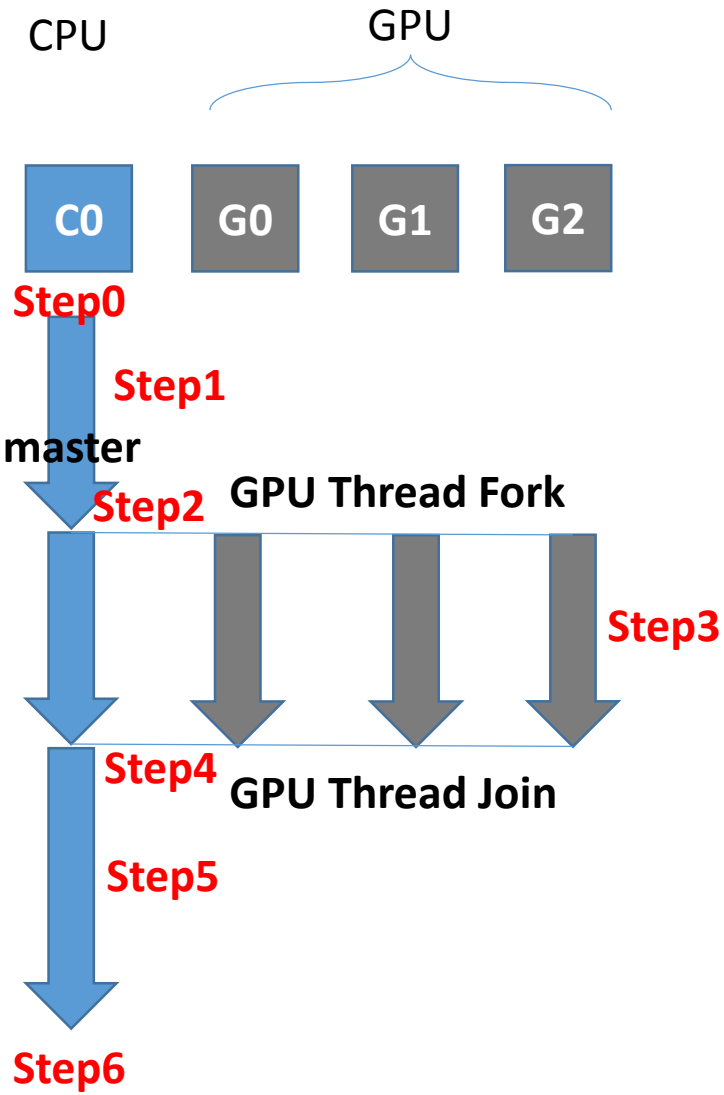



## CUDA/OpenACC on GPU



# Real Fork-Join Model in GPU



# OpenACC

- Http://www.openacc.org

**PGI compiler, Cray compiler and HMPP compiler support OpenACC**

**Current GCC and ICC compiler ignore OpenACC Directives**
**Without OpenACC option, PGCC also ignore OpenACC Directives**

## Fortran Keyword

**!$acc kernels**
**!$acc kernels loop**

**!$acc data**
**!$acc end data**

## C/C++ Keyword

**#pragma acc kernels**
**#pragma acc kernels for**

**#pragma acc data**

# OpenACC SAXPY Example

**Serial Code**

```
void saxpy_serial( int n, float a,
       float *x,
       float *restrict y )
{

    for (int i = 0; i < n; ++i)
      y[i] = a*x[i] + y[i];

}
```

pgcc ./saxpy.c

**GPU acceleration**

```
void saxpy_parallel( int n, float a,
       float *x,
       float *restrict y )
{
#pragma acc kernels
    for (int i = 0; i < n; ++i)
      y[i] = a*x[i] + y[i];

}
```

pgcc –acc -ta=nvidia ./saxpy.c
hmpp gcc –acc –ta=nvidia ./saxpy.c

# Quiz : how to accelerate CPI code with OpenACC?

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[])
{
    int nthreads, tid;
    int i, INTER;
    double n_1, x, pi = 0.0;
    INTER=100;
    n_1 = 1.0 / (double)INTER ;



        for (i = 0; i < INTER; i++)
        {
            x = n_1 * ((double)i - 0.5);
            pi += 4.0 / (1.0 + x * x);
        }
    pi *= n_1;
    printf ("Pi = %.12lf\n", pi);
    return 0;
}
```

# Quiz : how to accelerate CPI code with OpenACC?

```c
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[])
{
    int nthreads, tid;
    int i, INTER;
    double n_1, x, pi = 0.0;
    INTER=100;
    n_1 = 1.0 / (double)INTER ;



        for (i = 0; i < INTER; i++)
        {
            x = n_1 * ((double)i - 0.5);
            pi += 4.0 / (1.0 + x * x);
        }
    pi *= n_1;
    printf ("Pi = %.12lf\n", pi);
    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[])
{
    int nthreads, tid;
    int i, INTER;
    double n_1, x, pi = 0.0;
    INTER=100;
    n_1 = 1.0 / (double)INTER ;

#pragma acc kernels for

        for (i = 0; i < INTER; i++)
        {
            x = n_1 * ((double)i - 0.5);
            pi += 4.0 / (1.0 + x * x);
        }
    pi *= n_1;
    printf ("Pi = %.12lf\n", pi);
    return 0;
}
```



# Ex1-1 HelloCUDA

# Step3 NVCC .cu file

**source :** ex03-hello_nvcc.cu

```
#include <stdio.h>

int main(){

    printf("hello nvcc\n");
    return 0;

}
```

**result**

```
$./a.out
hello nvcc
$
```

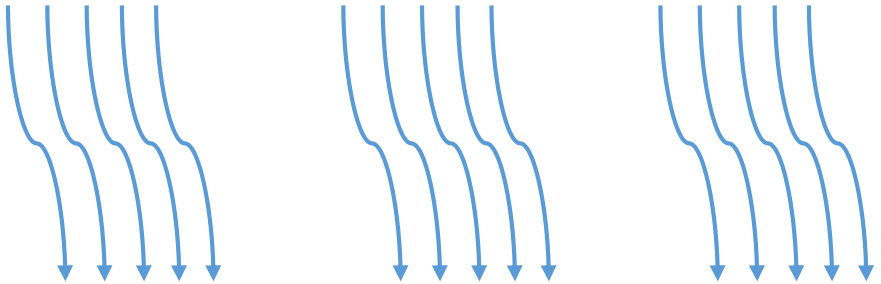
**command**

```
$cp ex03-hello_nvcc.c  ex03-hello_nvcc.cu
$vi ex03-hello_nvcc.cu
$nvcc ex03-hello_nvcc.cu
```

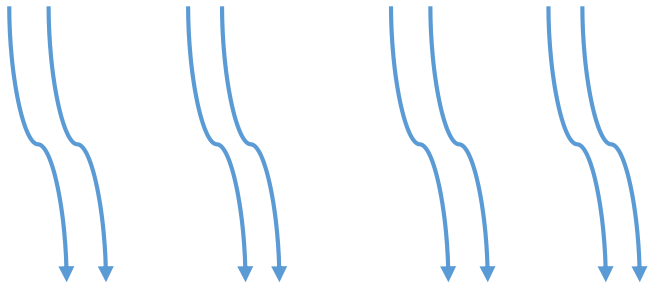
# GPU threads fork

```
__global__  void functionGPU;

functionGPU <<< A , B >>> ( );


__global__  void functionGPU() {

        //TODO

        Return ;
}
```

<<< 3 , 5 >>>



<<< 4 , 2 >>>

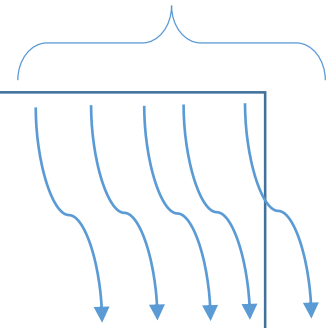


# Step4 Kernel (multiple blocks)

**source :** ex04-first_kernel.cu

```
#include <stdio.h>

__global__ void fA(){
  printf("hello from %d \n", blockIdx.x);
  return;
}

int main(){
   printf("hello nvcc\n");
   fA <<< 5 , 1 >>>();
   cudaDeviceReset();
   return 0;
}
```

**result**



```
$./a.out
hello nvcc
hello from 0
hello from 1
hello from 4
hello from 2
hello from 3
$
```

**command**

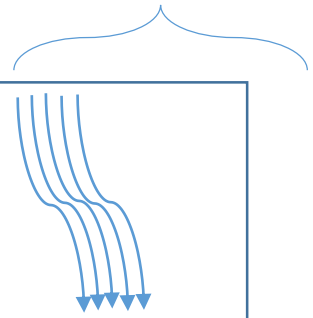
```
$ vi ex04-first_kernel.cu
$ nvcc -arch=sm_20 ex04-first_kernel.cu
```

# Step5 Kernel (Multiple threads)

**source :** ex05-kernel.cu

```
#include <stdio.h>

__global__ void fA(){
  printf("hello from %d \n", threadIdx.x);
  return;
}

int main(){
   printf("hello nvcc\n");
   fA <<< 1 , 5 >>>();
   cudaDeviceReset();
   return 0;
}
```

**result**



```
$./a.out
hello nvcc
hello from 0
hello from 1
hello from 2
hello from 3
hello from 4
$
```

**command**

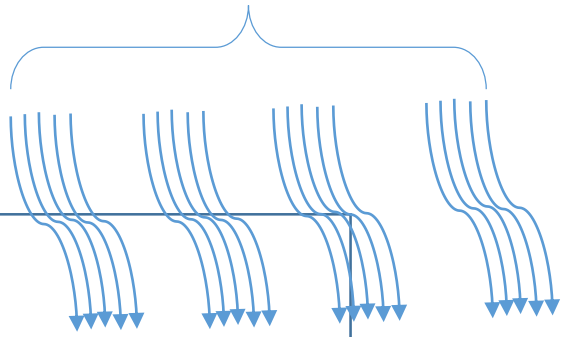
```
$ cp ex04-first_kernel.cu ex05-kernel.cu
$vi ex05-kernel.cu
$ nvcc -arch=sm_20 ex05-kernel.cu
```

# Step6 Kernel (Mutiple block and thread)

**source :** ex06-kernel.cu

```
#include <stdio.h>

__global__ void fA(){
  printf("hello from %d %d \n",
            blockIdx.x , threadIdx.x);
  return;
}

int main(){
   printf("hello nvcc\n");
   fA <<< 4 , 5 >>>();
   cudaDeviceReset();
   return 0;
}
```

**result**



```
$./a.out
hello nvcc
hello from 1 0
hello from 1 1
hello from 1 2
hello from 1 3
hello from 1 4
hello from 0 0
hello from 0 1
hello from 0 2
hello from 0 3
hello from 0 4
hello from 3 0
hello from 3 1
hello from 3 2
hello from 3 3
hello from 3 4
hello from 2 0
hello from 2 1
hello from 2 2
hello from 2 3
hello from 2 4
$
```

**command**

```
$ cp ex05-kernel.cu ex06-kernel.cu
$vi ex06-kernel.cu
$ nvcc -arch=sm_20 ex06-kernel.cu
```

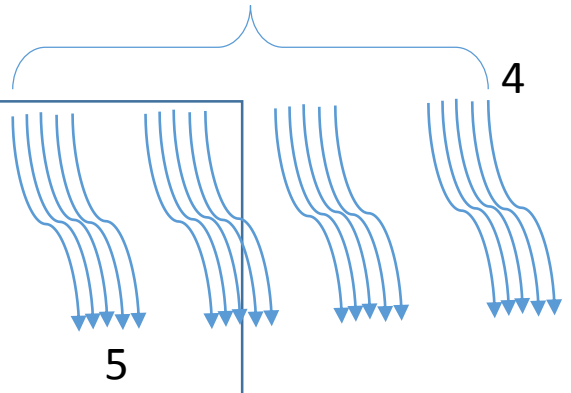
# Step7 Kernel (info)

**source :** ex07-kernel.cu

```
#include <stdio.h>

__global__ void fA(){
  printf("hello from %d %d %d %d\n",
        blockIdx.x, threadIdx.x,
        gridDim.x, blockDim.x );
  return;
}

int main(){
   printf("hello nvcc\n");
   fA <<< 4 , 5 >>>();
   cudaDeviceReset();
   return 0;
}
```

**result**

```
$./a.out
hello nvcc
hello from 1 0 4 5
hello from 1 1 4 5
hello from 1 2 4 5
hello from 1 3 4 5
hello from 1 4 4 5
hello from 0 0 4 5
hello from 0 1 4 5
hello from 0 2 4 5
hello from 0 3 4 5
hello from 0 4 4 5
hello from 3 0 4 5
hello from 3 1 4 5
hello from 3 2 4 5
hello from 3 3 4 5
hello from 3 4 4 5
hello from 2 0 4 5
hello from 2 1 4 5
hello from 2 2 4 5
hello from 2 3 4 5
hello from 2 4 4 5
$
```




**command**

```
$ cp ex06-third_kernel.cu ex07-kernel.cu
$vi ex07-kernel.cu
$ nvcc -arch=sm_20 ex07-kernel.cu
```

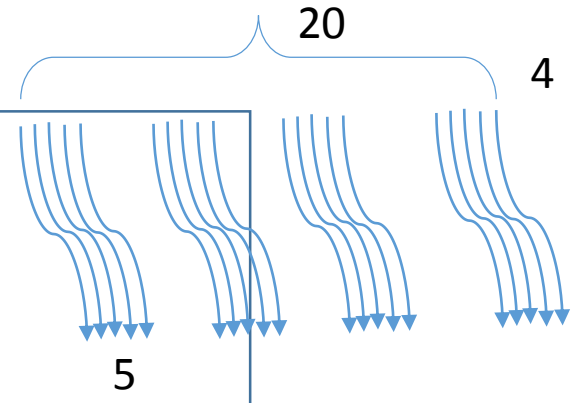
# Step8 Kernel (unique ID)

**source :** ex08-kernel.cu

```
#include <stdio.h>

__global__ void fA(){

int N = gridDim.x * blockDim.x; //20
int JS = blockDim.x; // block 1+
int i = blockIdx.x * JS  + threadIdx.x;

   printf("hello from %d %d : %d %d: %d %d\n",
         blockIdx.x, threadIdx.x,
         gridDim.x, blockDim.x,
         i, N );
  return;
}

int main(){
   printf("hello nvcc\n");
   fA <<< 4 , 5 >>>();
   cudaDeviceReset();
   return 0;
}
```

**result**

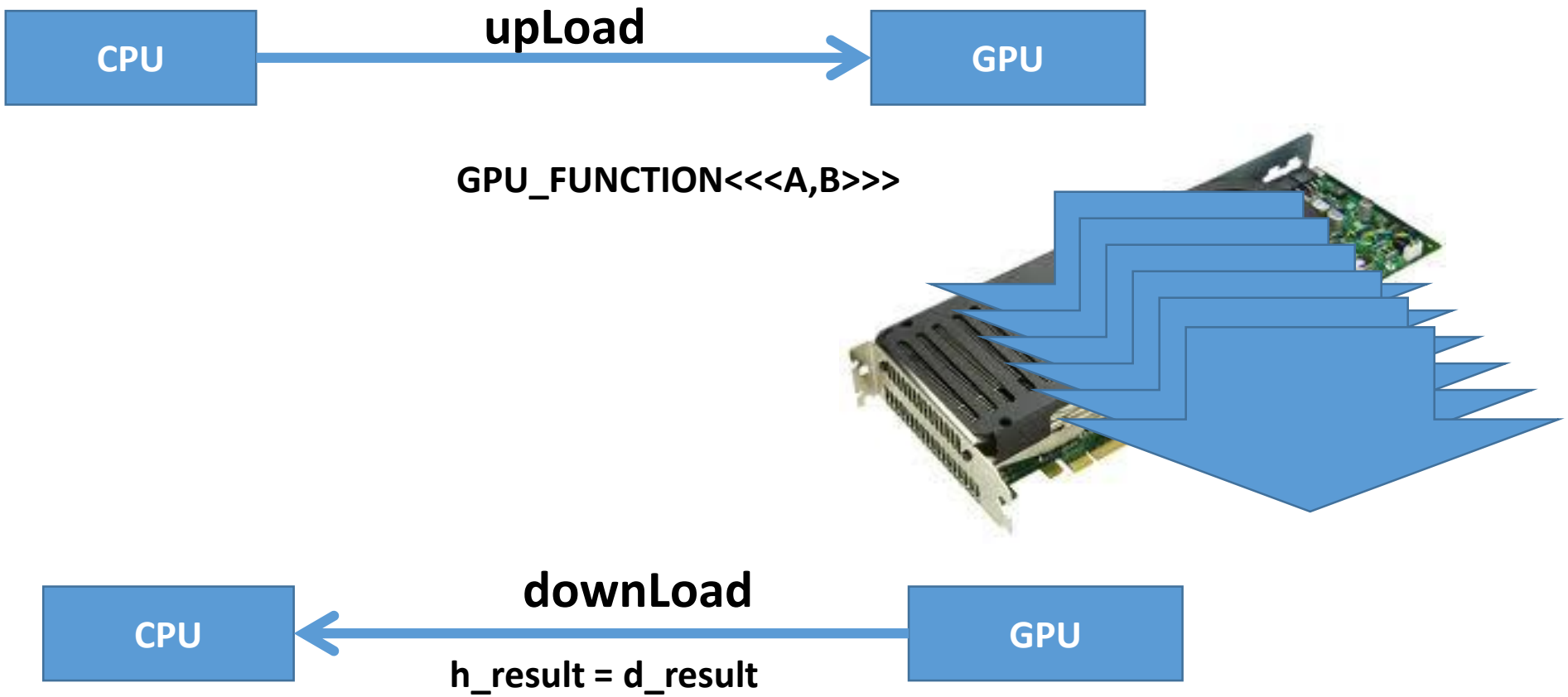
```
$./a.out
hello nvcc
hello from 1 0 : 4 5: 5 20
hello from 1 1 : 4 5: 6 20
hello from 1 2 : 4 5: 7 20
hello from 1 3 : 4 5: 8 20
hello from 1 4 : 4 5: 9 20
hello from 0 0 : 4 5: 0 20
hello from 0 1 : 4 5: 1 20
hello from 0 2 : 4 5: 2 20
hello from 0 3 : 4 5: 3 20
hello from 0 4 : 4 5: 4 20
hello from 2 0 : 4 5: 10 20
hello from 2 1 : 4 5: 11 20
hello from 2 2 : 4 5: 12 20
hello from 2 3 : 4 5: 13 20
hello from 2 4 : 4 5: 14 20
hello from 3 0 : 4 5: 15 20
hello from 3 1 : 4 5: 16 20
hello from 3 2 : 4 5: 17 20
hello from 3 3 : 4 5: 18 20
hello from 3 4 : 4 5: 19 20
$
```




**command**

```
$ cp ex07-kernel.cu ex08-kernel.cu
$vi ex08-kernel.cu
$ nvcc -arch=sm_20 ex08-kernel.cu
```

# CUDA Template



# CUDA Code Template



CUDA Malloc

CUDA Memset

CUDA Memcpy(upload input)

Launch GPU Kernel

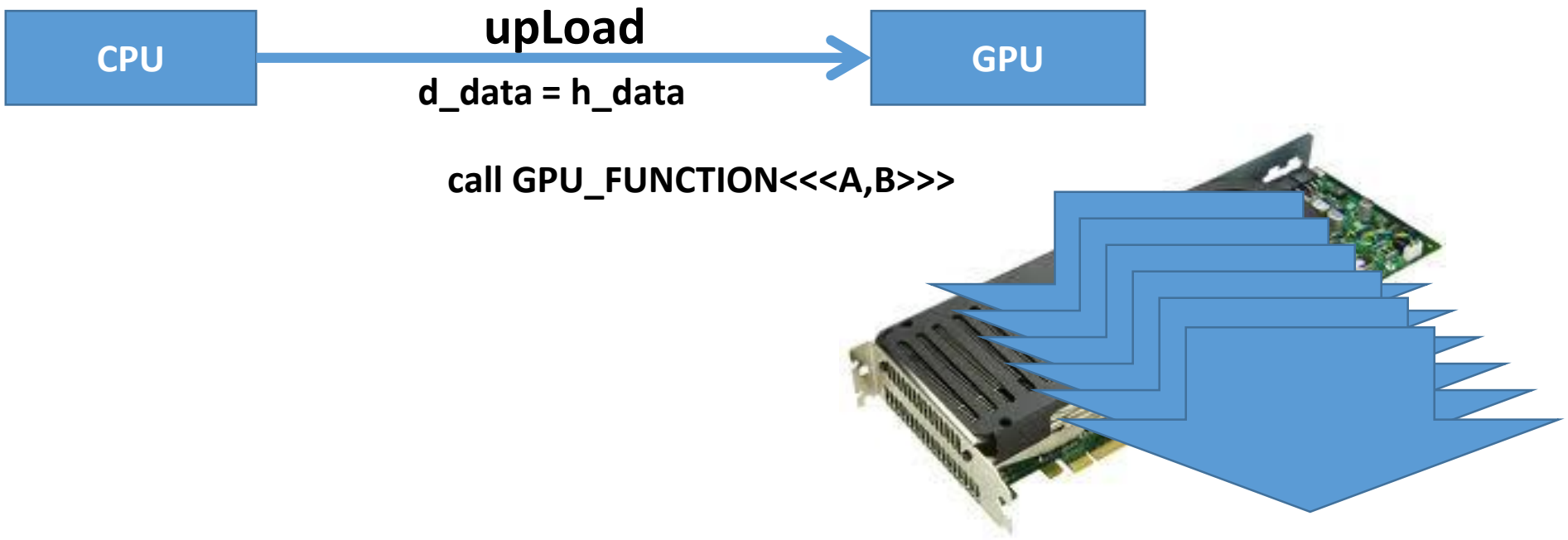
CUDA Memcpy(get result)

Define CUDA Kernel

```
#include <stdio.h>
#define  A 10
#define  B 511
#define N 10
__global__ void functionG(float *input, float *output);

main(){
printf("hello CUDA\n");
float *x_h, *y_h, *x_d, *y_d;
size_t memSize = sizeof(float) * N;

x_h = (float *)malloc(memSize);
y_h = (float *)malloc(memSize);

cudaMalloc( (void**)&x_d, memSize );
cudaMalloc( (void**)&y_d, memSize );

cudaMemset( x_d, 0.0, memSize);
cudaMemset( y_d, 0.0, memSize);
for( int i =0; i<N; i++){x_h[i]=i; y_h[i]=0.0;


cudaMemcpy( x_d, x_h, memSize, cudaMemcpyHostToDevice);

functionG <<< A , B >>> (x_d, y_d);

cudaMemcpy( y_h, y_d, memSize, cudaMemcpyDeviceToHost);
for( int i =0; i<N; i++){printf("%d, %f %f \n",i, x_h[i], y_h[i] ); } return ;
}
void __global__ functionG(float *input, float *output)
{
          int idx  = threadIdx.x + blockIdx.x *blockDim.x;
          if ( idx < N) {
                    output[idx]= input[idx] + 0.001 * idx;
          }
}
```

# CUDA FORTRAN Model

```
integer :: h_data(n), h_result(n), b
integer, device :: d_data(n)
```





# Example pyCUDA

# pyCUDA Template

```
import pycuda.driver as drv
import pycuda.tools
import pycuda.autoinit
import numpy
import numpy.linalg as la
from pycuda.compiler import SourceModule

mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(4).astype(numpy.float32)
b = numpy.random.randn(4).astype(numpy.float32)

dest = numpy.zeros_like(a)

multiply_them(
  drv.Out(dest), drv.In(a), drv.In(b),
  block=(400,1,1))

print a*b
print dest

print dest-a*b
```

pyCUDA initialize

define CUDA kernel

define python function

init on CPU memory

Launch Kernel

Define cudaMemcpy

Define Job index

# Python Image Libraray

```
from __future__ import division
import numpy
import matplotlib.pyplot as plt
from PIL import Image


# read image file
img = Image.open("6x3-pixel.png")
arr=numpy.array(img)

print arr

# plot the numpy array
plt.imshow(arr)
plt.show()
```

Image library initialize

Image File Read

array for image processing

Show plot

# pyCUDA import

```
############ for image processing  #################
from __future__ import division
import numpy
import matplotlib.pyplot as plt
from PIL import Image
from scipy import misc



############ for pyCUDA  ################
import pycuda.driver as drv
import pycuda.tools
import pycuda.autoinit
import numpy.linalg as la
from pycuda.compiler import SourceModule
```

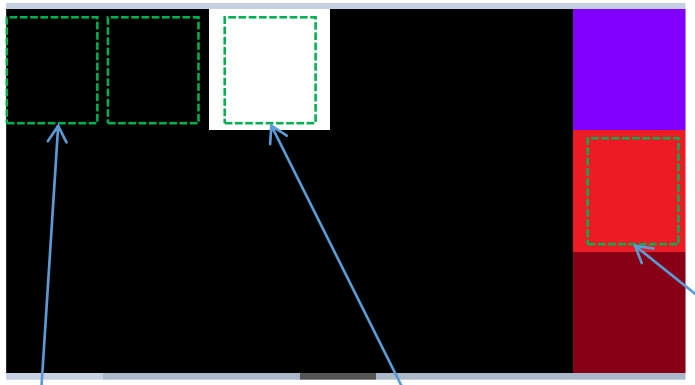
```
########### for image processing  ################
from __future__ import division
import numpy
import matplotlib.pyplot as plt
from PIL import Image
from scipy import misc


########### for pyCUDA  ###############
import pycuda.driver as drv
import pycuda.tools
import pycuda.autoinit
import numpy.linalg as la
from pycuda.compiler import SourceModule
```

```
########### read image file    ##################
img = Image.open("Fisheye-Nikkor 10.5mm-sample4-building.jpg")

# convert image to numpy array
arr = numpy.array(img)

#upload to GPU
#TODO kernel
#download to CPU



# result value
tmp = numpy.empty_like(arr)
tmp = arr

# plot the numpy array
plt.subplot(121)
plt.title("original")
plt.imshow(arr)

plt.subplot(122)
plt.title("defished")
plt.imshow(tmp)

plt.show()
```

# Kernel Launch

```
bright = mod.get_function("bright")

# read image file
img = Image.open("papua.jpg")
# convert image to numpy array
arr = numpy.array(img)
tmp = numpy.empty_like(arr)

bright(
      drv.In(arr), drv.Out(tmp) ,
      grid=(40,60,1), block=(40,20,1)
      )
```

# Numpy Array Data Structure



**arr=np.array (img)**

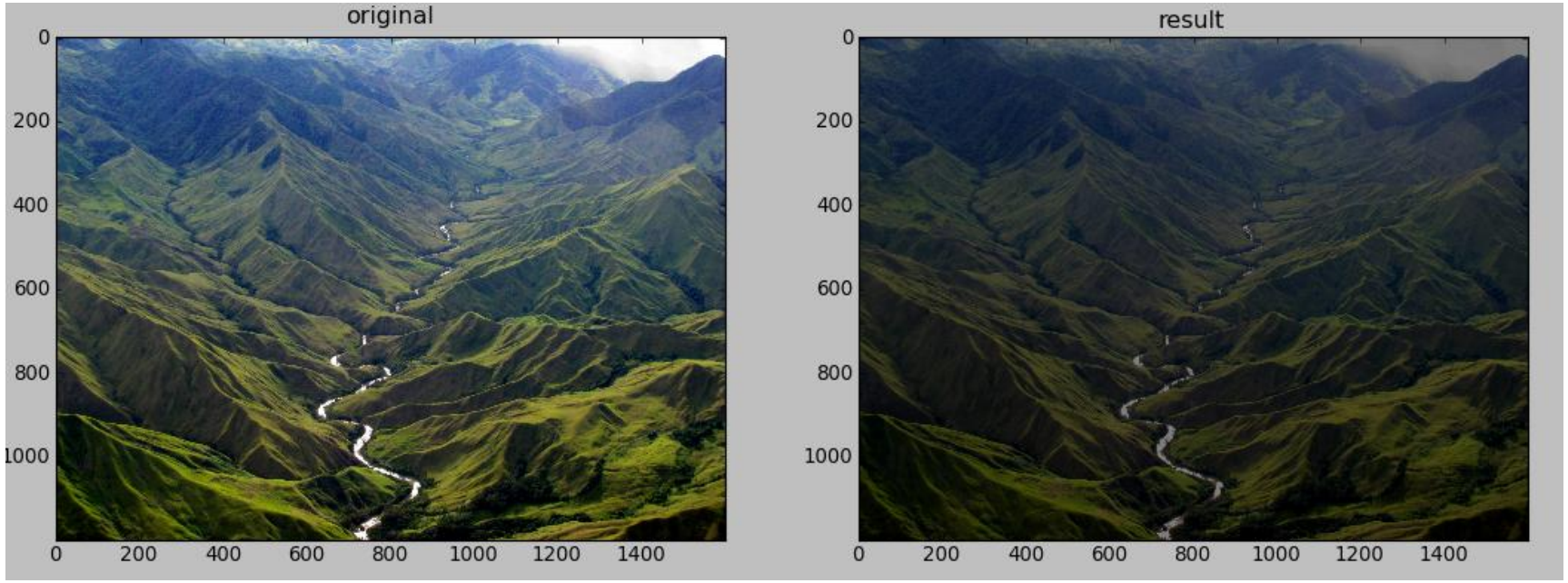
**PNG image example**

```
[
[[ 0  0  0] [ 0  0  0] [255 255 255] [ 0  0  0] [ 0  0  0] [128  0 255]]

[[ 0  0  0] [ 0  0  0] [ 0  0  0] [ 0  0  0] [ 0  0  0] [237 28 36]]

[[ 0  0  0] [ 0  0  0] [ 0  0  0] [ 0  0  0] [ 0  0  0] [136  0 21]]
]
```

# pyCUDA module (RGB control)

```
mod = SourceModule("""
__global__ void bright(unsigned char *in, unsigned char *out){
  int ix = blockIdx.x * blockDim.x + threadIdx.x; //1600   16* 5 , 20
  int iy = blockIdx.y * blockDim.y + threadIdx.y; //1200   12* 5 , 20
  int it = ix * ( gridDim.y * blockDim.y  ) + iy;
  int px = ix *3;
  int py = iy *3;
  int pixel_R = (py * 1600) + px + 0;
  int pixel_G = (py * 1600) + px + 1;
  int pixel_B = (py * 1600) + px + 2;
  int R,G,B;
     R=in[pixel_R];
     G=in[pixel_G];
     B=in[pixel_B];
     out[pixel_R]=R/2;
     out[pixel_G]=G/2;
     out[pixel_B]=B/2;
```

**Consider relation with Numpy Array & CUDA job Index**




```
out[pixel_R]=R/2;
out[pixel_G]=G/2;
out[pixel_B]=B/2;
```



# Ex2-2 transpose

# Transpose

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 66 | 77 | 71 | 89 | 5 | 82 | 95 |
| 2 | 55 | 20 | 45 | 24 | 3 | 68 | 30 | 40 |
| 3 | 72 | 67 | 49 | 30 | 61 | 61 | 69 | 49 |
| 4 | 62 | 33 | 55 | 20 | 25 | 43 | 52 | 91 |
| 5 | 42 | 57 | 18 | 14 | 37 | 42 | 78 | 48 |
| 6 | 68 | 70 | 85 | 68 | 68 | 23 | 93 | 1 |
| 7 | 56 | 39 | 38 | 73 | 59 | 22 | 72 | 98 |
| 8 | 79 | 64 | 82 | 68 | 97 | 55 | 41 | 32 |

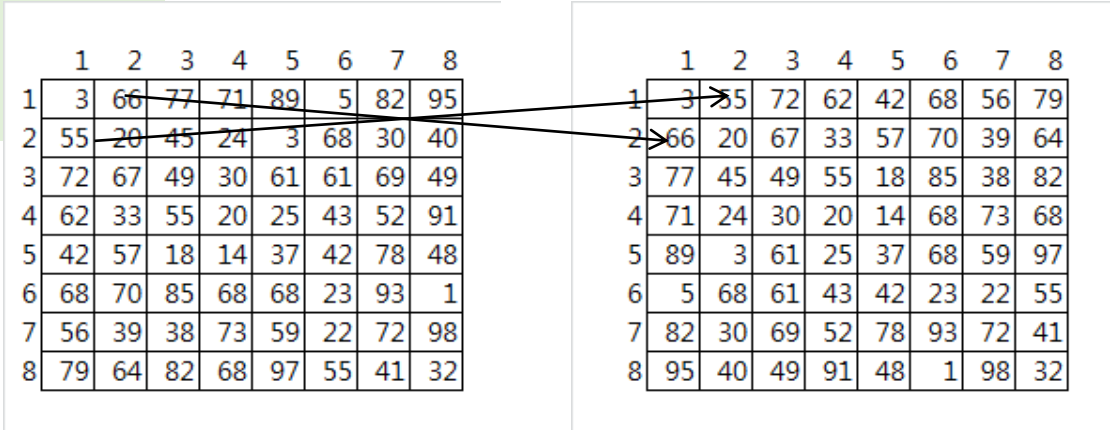
**A**

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 55 | 72 | 62 | 42 | 68 | 56 | 79 |
| 2 | 66 | 20 | 67 | 33 | 57 | 70 | 39 | 64 |
| 3 | 77 | 45 | 49 | 55 | 18 | 85 | 38 | 82 |
| 4 | 71 | 24 | 30 | 20 | 14 | 68 | 73 | 68 |
| 5 | 89 | 3 | 61 | 25 | 37 | 68 | 59 | 97 |
| 6 | 5 | 68 | 61 | 43 | 42 | 23 | 22 | 55 |
| 7 | 82 | 30 | 69 | 52 | 78 | 93 | 72 | 41 |
| 8 | 95 | 40 | 49 | 91 | 48 | 1 | 98 | 32 |

**A’**

# CPU code1 [ update same array]

```
void transposeCPU1( float* a,  int W) {

   for (int i = 0; i < N; i++) {
      for (int j = i+1; j < N; j++) {
         float temp = a[i][j];
         a[i][j] = a[j][i];
         a[j][i] = temp;
      }
   }
}
```



| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 66 | 77 | 71 | 89 | 5 | 82 | 95 |
| 2 | 55 | 20 | 45 | 24 | 3 | 68 | 30 | 40 |
| 3 | 72 | 67 | 49 | 30 | 61 | 61 | 69 | 49 |
| 4 | 62 | 33 | 55 | 20 | 25 | 43 | 52 | 91 |
| 5 | 42 | 57 | 18 | 14 | 37 | 42 | 78 | 48 |
| 6 | 68 | 70 | 85 | 68 | 68 | 23 | 93 | 1 |
| 7 | 56 | 39 | 38 | 73 | 59 | 22 | 72 | 98 |
| 8 | 79 | 64 | 82 | 68 | 97 | 55 | 41 | 32 |

A

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 55 | 72 | 62 | 42 | 68 | 56 | 79 |
| 2 | 66 | 20 | 67 | 33 | 57 | 70 | 39 | 64 |
| 3 | 77 | 45 | 49 | 55 | 18 | 85 | 38 | 82 |
| 4 | 71 | 24 | 30 | 20 | 14 | 68 | 73 | 68 |
| 5 | 89 | 3 | 61 | 25 | 37 | 68 | 59 | 97 |
| 6 | 5 | 68 | 61 | 43 | 42 | 23 | 22 | 55 |
| 7 | 82 | 30 | 69 | 52 | 78 | 93 | 72 | 41 |
| 8 | 95 | 40 | 49 | 91 | 48 | 1 | 98 | 32 |

A’

# CPU code2 [different array]

```
void transposeCPU2( float* a, float* b,  int W) {

   for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
         b[i][j] = a[j][i];
      }
   }
}
```

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 66 | 77 | 71 | 89 | 5 | 82 | 95 |
| 2 | 55 | 20 | 45 | 24 | 3 | 68 | 30 | 40 |
| 3 | 72 | 67 | 49 | 30 | 61 | 61 | 69 | 49 |
| 4 | 62 | 33 | 55 | 20 | 25 | 43 | 52 | 91 |
| 5 | 42 | 57 | 18 | 14 | 37 | 42 | 78 | 48 |
| 6 | 68 | 70 | 85 | 68 | 68 | 23 | 93 | 1 |
| 7 | 56 | 39 | 38 | 73 | 59 | 22 | 72 | 98 |
| 8 | 79 | 64 | 82 | 68 | 97 | 55 | 41 | 32 |

A

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 55 | 72 | 62 | 42 | 68 | 56 | 79 |
| 2 | 66 | 20 | 67 | 33 | 57 | 70 | 39 | 64 |
| 3 | 77 | 45 | 49 | 55 | 18 | 85 | 38 | 82 |
| 4 | 71 | 24 | 30 | 20 | 14 | 68 | 73 | 68 |
| 5 | 89 | 3 | 61 | 25 | 37 | 68 | 59 | 97 |
| 6 | 5 | 68 | 61 | 43 | 42 | 23 | 22 | 55 |
| 7 | 82 | 30 | 69 | 52 | 78 | 93 | 72 | 41 |
| 8 | 95 | 40 | 49 | 91 | 48 | 1 | 98 | 32 |

A’

# CPU code5 [1 D Array]

```
void transposeCPU5( float* input, float* output,  int W) {

   for(  int i = 0; i < W; i++) {
            for(  int j = 0; j < W; j++) {
               output[ (i * W) + j ] = input[ ( i * W) + j ];
            }
   }
}
```

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 66 | 77 | 71 | 89 | 5 | 82 | 95 |
| 2 | 55 | 20 | 45 | 24 | 3 | 68 | 30 | 40 |
| 3 | 72 | 67 | 49 | 30 | 61 | 61 | 69 | 49 |
| 4 | 62 | 33 | 55 | 20 | 25 | 43 | 52 | 91 |
| 5 | 42 | 57 | 18 | 14 | 37 | 42 | 78 | 48 |
| 6 | 68 | 70 | 85 | 68 | 68 | 23 | 93 | 1 |
| 7 | 56 | 39 | 38 | 73 | 59 | 22 | 72 | 98 |
| 8 | 79 | 64 | 82 | 68 | 97 | 55 | 41 | 32 |

A

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 66 | 77 | 71 | 89 | 5 | 82 | 95 | 55 | 20 | 45 | 24 | 3 | 68 | 30 | 40 | 72 | 67 | 49 | 30 | 61 | 61 | 69 | 49 |

**2D Array: A[i][j]**

**1D Array : A[i*width + j ]**

# GPU code1 : naive

Index Control
Job index
Data index

```
__global__ void transposeGPU1(float *input, float* output, int W)
{
  int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
  int yIndex = blockIdx.y * blockDim.y + threadIdx.y;

  int index_total     = xIndex * W + yIndex;  //  [i][j]=i*w+j
  int index_total_rev = yIndex * W + xIndex;  //[j][i]=j*w+i

  int index_in   = index_total;
  int index_out = index_total_rev;

output[index_out] = input[index_in];
}
```

Define Job Size
Kernel Launch

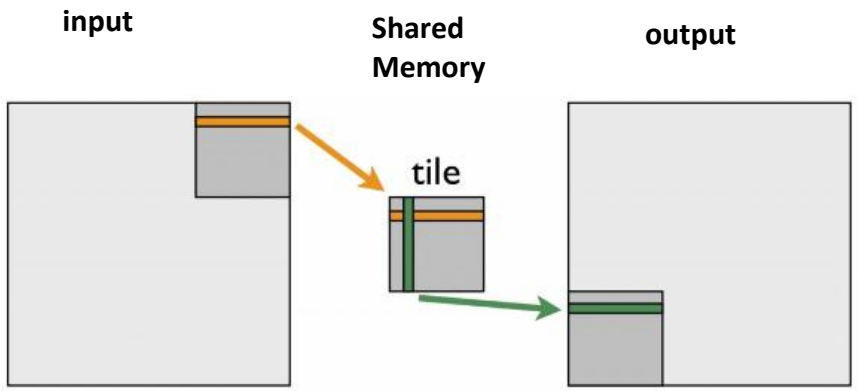
```
A.x=10000; A.y=10000; B.x=20, B.y=20;
transposeGPU<<<A,B>>>(input, output, width);
```

Matrix with 200000 x 200000

# GPU code2 : Shared Memory Version

Same technique as Blocked Matrix Multiplication

```
__global__ void transposeGPU2(float *input,   float *output, int W)
{
  __shared__ float tile[TILE_DIM * TILE_DIM];

  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;  // same as W

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
    tile[(threadIdx.y+j)*TILE_DIM + threadIdx.x] = input[(y+j)*width + x];

  __syncthreads();

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
    output[(y+j)*width + x] = tile[(threadIdx.y+j)*TILE_DIM + threadIdx.x];
}
```





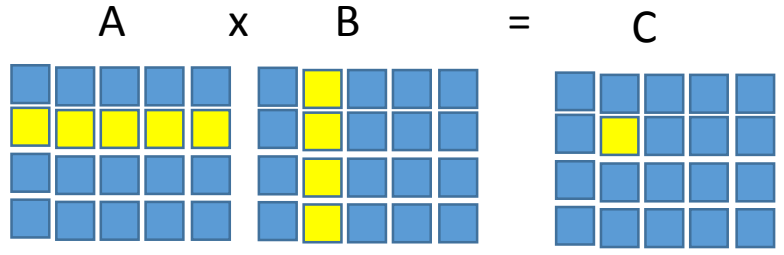
# DGEMM and cuBLAS libraries

# CPU version DGEMM routine

BLAS (Basic Linear Algebra Subprograms)
- BLAS Level 1 VV
- BLAS Level 2 MV
- BLAS Level 3 MM

Double Precision General Matrix Matrix Multiplication

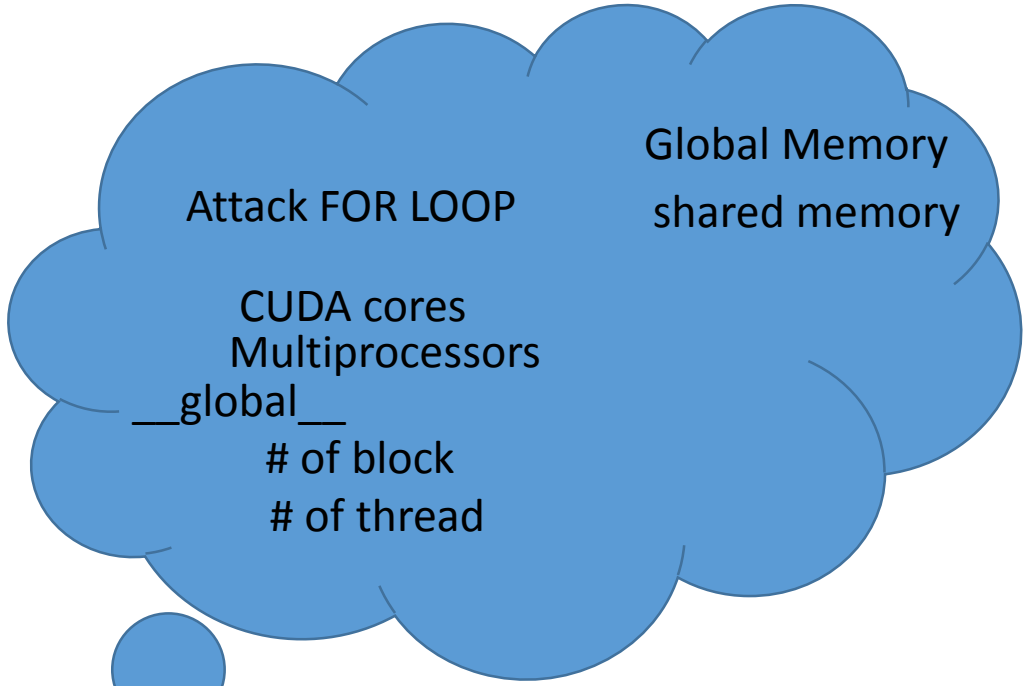
```
void simple_dgemm(int n, double alpha, const double *A,
                  const double *B,
                  double beta, double *C)
{
    int i, j, k;
    for (i = 0; i < n; ++i) {
        for (j = 0; j < n; ++j) {
            double inprod = 0;
            for (k = 0; k < n; ++k) {
                inprod += A[k * n + i] * B[j * n + k];
            }
            C[j * n + i] = alpha * inprod + beta * C[j * n + i];
        }
    }
}
```



# DGEMM example in MKL

```
#include <stdio.h>
#include <mkl.h>

int main(void) {
 double *a, *b, *c;  int n, i;  double alpha, beta;  MKL_INT64 AllocatedBytes;  int N_AllocatedBuffers;

 alpha = 1.1; beta = -1.2;  n = 1000;
 a = (double*)mkl_malloc(n*n*sizeof(double),128);
 b = (double*)mkl_malloc(n*n*sizeof(double),128);
 c = (double*)mkl_malloc(n*n*sizeof(double),128);
 for (i=0;i<(n*n);i++) {     a[i] = (double)(i+1);     b[i] = (double)(-i-1);     c[i] = 0.0;  }

 dgemm("N","N",&n,&n,&n,&alpha,a,&n,b,&n,&beta,c,&n);

 AllocatedBytes = mkl_mem_stat(&N_AllocatedBuffers);
 printf("\nDGEMM uses %ld bytes in %d buffers",(long)AllocatedBytes,N_AllocatedBuffers);
  mkl_free_buffers();
```

# Brainstorming for CUDA




**CUDAMalloc**

**CUDA Memcpy**

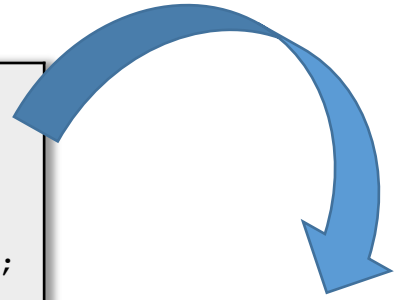
**Kernel Launch**

**CUDA Memcpy**

```
for ( int row=0; row<n; row++ ) {
   for ( int col=0; col<n; col++ ) {
      double val = 0;
      for ( int k=0; k<n; k++ ) {
        val += a[row*n+k] * b[k*n+col];
      }
      c[row*n+col] = val;
    }
```

# OpenACC : easiest way to accelerate with GPU

```
for ( int row=0; row<n; row++ ) {
   for ( int col=0; col<n; col++ ) {
      double val = 0;
      for ( int k=0; k<n; k++ ) {
          val += a[row*n+k] * b[k*n+col];
      }
      c[row*n+col] = val;
    }
```



```
#pragma acc kernels
for ( int row=0; row<n; row++ ) {
   for ( int col=0; col<n; col++ ) {
      double val = 0;
      for ( int k=0; k<n; k++ ) {
        val += a[row*n+k] * b[k*n+col];
      }
      c[row*n+col] = val;
    }
```

# CUDA DGEMM (Naïve)

```
__global__  void simple_dgemm_GPU(int n, double alpha, const double *A, const double *B,
                                  double beta, double *C)
{
  int  i = blockIdx.x * (gridDim.x) * threadIdx.x; // row
  int  j = blockIdx.y * (gridDim.y) * threadIdx.y;  //rol

    int  k;

          double prod = 0;
          for (k = 0; k < n; ++k) {
             prod += A[k * n + i] * B[j * n + k];
          }
          C[j * n + i] = alpha * prod + beta * C[j * n + i];

}
```

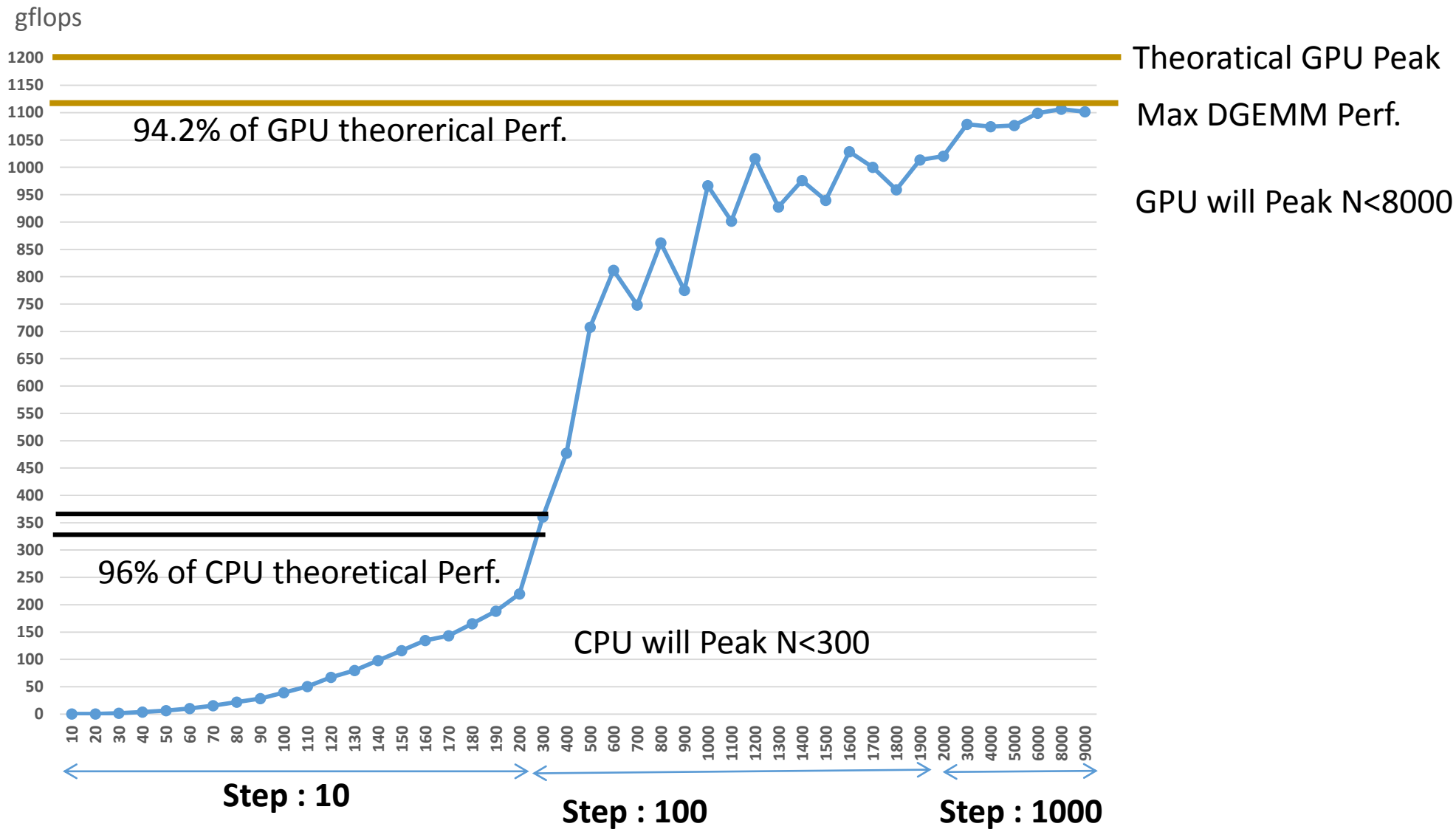
# Main code in cuBLAS

```
h_A = (float*) malloc(n2 * sizeof(h_A[0]));
h_B = (float *) malloc(n2 * sizeof(h_B[0]));
h_C = (float *) malloc(n2 * sizeof(h_C[0]));

cublasAlloc (n2, sizeof(d_A[0]),  (void**)&d_A );
cublasAlloc (n2, sizeof(d_B[0]),  (void **)&d_B );
cublasAlloc (n2, sizeof(d_C[0]),  (void **)&d_C );

 cublasSetVector (n2, sizeof(h_A[0]), h_A, 1, d_A, 1 );
cublasSetVector (n2, sizeof(h_B[0]), h_B, 1, d_B, 1 );
cublasSetVector (n2, sizeof(h_C[0]), h_C, 1, d_C, 1 );


cublasSgemm('n', 'n', N, N, N, alpha, d_A, N, d_B, N, beta, d_C, N );


cublasGetVector(n2, sizeof(h_C[0]), d_C, 1, h_C, 1 );
```

**gcc benchmark.c -lcuda -lcublas**

# cuBLAS DGEMM Perf.

NxN Matrix on Tesla K20M
NVIDIA Korea PSG Cluster

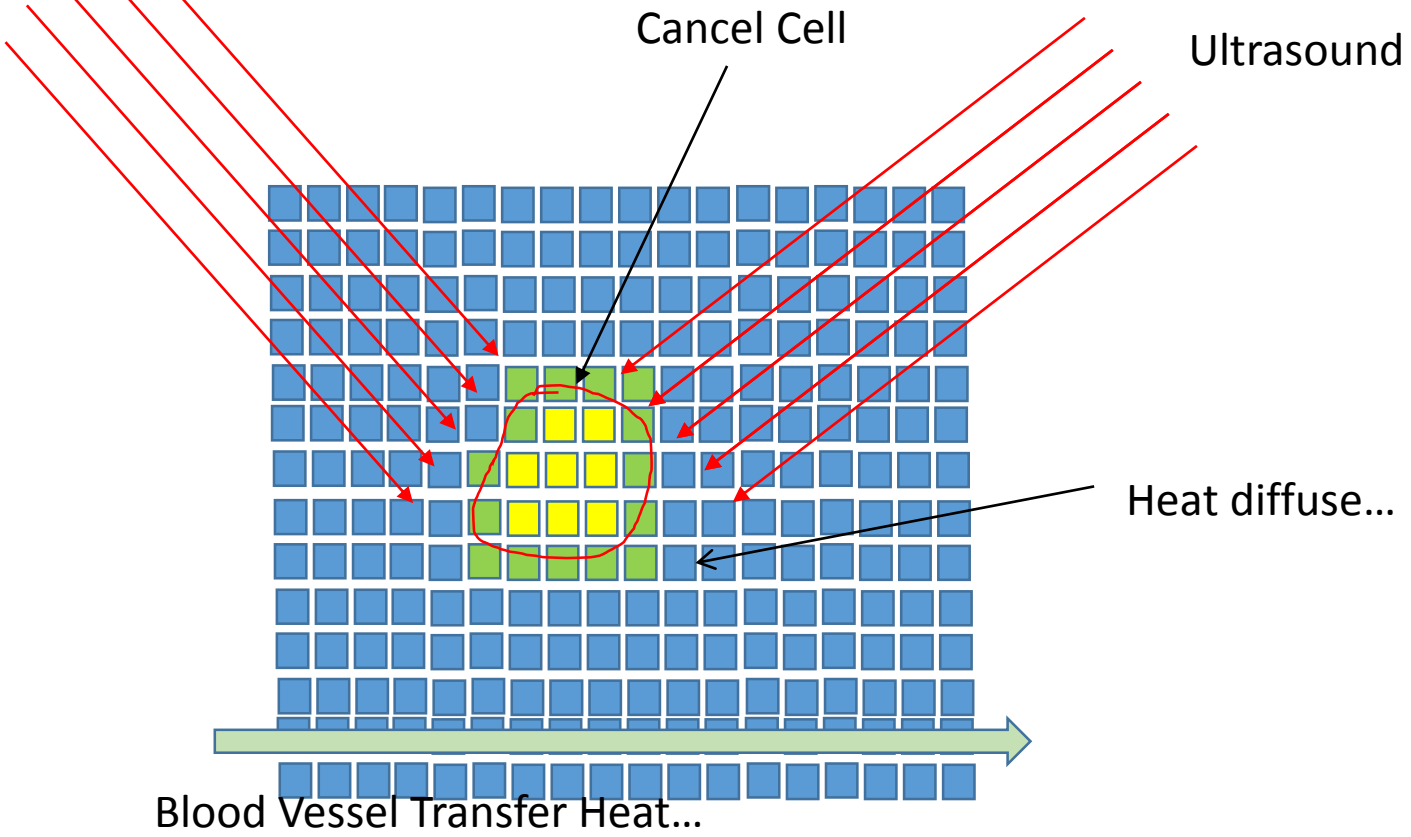




# CUDA example for BHTE(Bio Heat Transer Equation)

# 1D BHTE

Ultrasound provide External Thermal Source
Ultrasound




Cancer Therapy, Thermal Dose affects the tissue
- kill the cancer cell or not
- kill the healthy cell or not

BHTE

Heat diffusion
Heat Transfer
External Heat Source

Thermal Dose :
- expose time : 10~60min
- temperature : 40~46C

# BHTE(Bio Heat Transfer Equation)

$$\rho_t C_t \frac{\partial T(\mathbf{x},t)}{\partial t} = k_t \nabla^2 T(\mathbf{x},t) + V\rho_b C_b (T_b - T(\mathbf{x},t)) + Q(\mathbf{x},t)$$

time

diffusion

transfer

source

Heat equation

Heat transfer

# Descretization(이산화)

Forward Time Central Stencil FDM

$$T^{n+1}(\mathbf{x}) = T^{n}(\mathbf{x}) + \frac{\delta t}{\rho_t C_t} \cdot \left(k_t \nabla^2_{discret} T^{n}(\mathbf{x})\right) + V\,\rho_b C_b (T_b - T^{n}(\mathbf{x})) + Q^{n}(\mathbf{x}))$$

**time** | **diffusion** | **transfer** | **source**

**Heat equation** | **Heat transfer**

# BHTE Algorithm

**Computational Intensity : SAPXY level computation**

Forward Time Central Stencil FDM

$$T^{n+1}(\mathbf{x}) = T^{n}(\mathbf{x}) + \frac{\delta t}{\rho_t C_t} \cdot \left(k_t \nabla^2_{discret} T^{n}(\mathbf{x})\right) + V \rho_b C_b (T_b - T^{n}(\mathbf{x})) + Q^{n}(\mathbf{x}))$$

**time** | **diffusion** | **transfer** | **source**

**Heat equation** | **Heat transfer**

T_n+1 = T_n + dt / ( rhot + Ct) * ( kt * L < T_n > + Vb * rhob* Cb * ( Tb - T_n ) + Q_n )

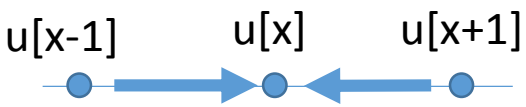
```
L < . > = 1 / (dx)^2 * (
            T_n( x-1,y,z) + T_n(x+1,y,z) +
            T_n(x,y-1,z) + T_n(x,y+1,z) +
            T_n(x,y,z-1) +T_n(x,y,z+1)
            - 6*T_n(x,y,z)
            )
```

**define A(x,y,z) ~ A[x][y][z]**

# Analysis of BHTE

We can count he arithmetic computation for each 3D voxel

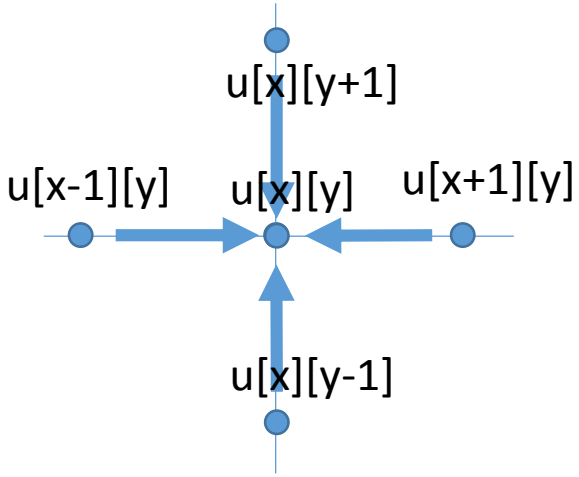
1D




$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{x-1} - 2u_x + u_{x+1}}{\Delta x^2}$$

```
1DLaplace = 1/(dx*dx)*(
    T[x-1]+T[x+1]-2*T[x][y][z]);
```

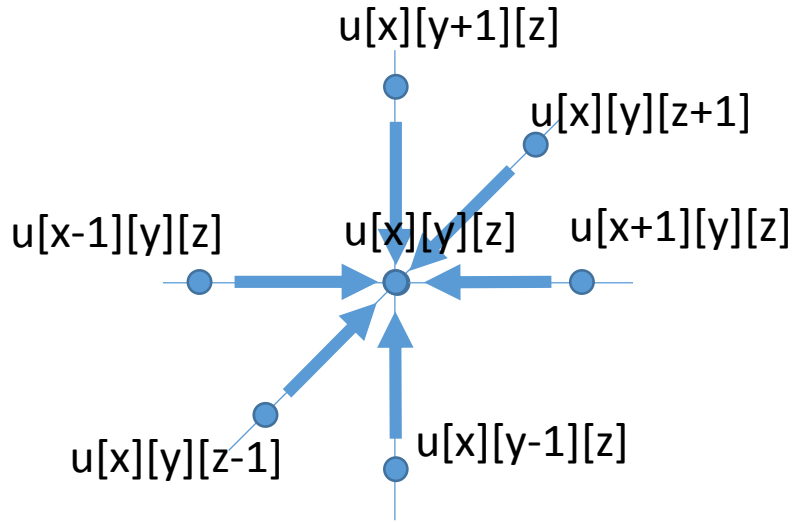
2D




$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{x-1} - 2u_x + u_{x+1}}{\Delta x^2}$$
$$\frac{\partial^2 u}{\partial y^2} = \frac{u_{y-1} - 2u_y + u_{y+1}}{\Delta y^2}$$

```
2DLaplace = 1/(dx*dx)*(
    T[x-1][y]+T[x+1][y]+
    T[x][y-1]+T[x][y+1]-4*T[x][y][z]);
```

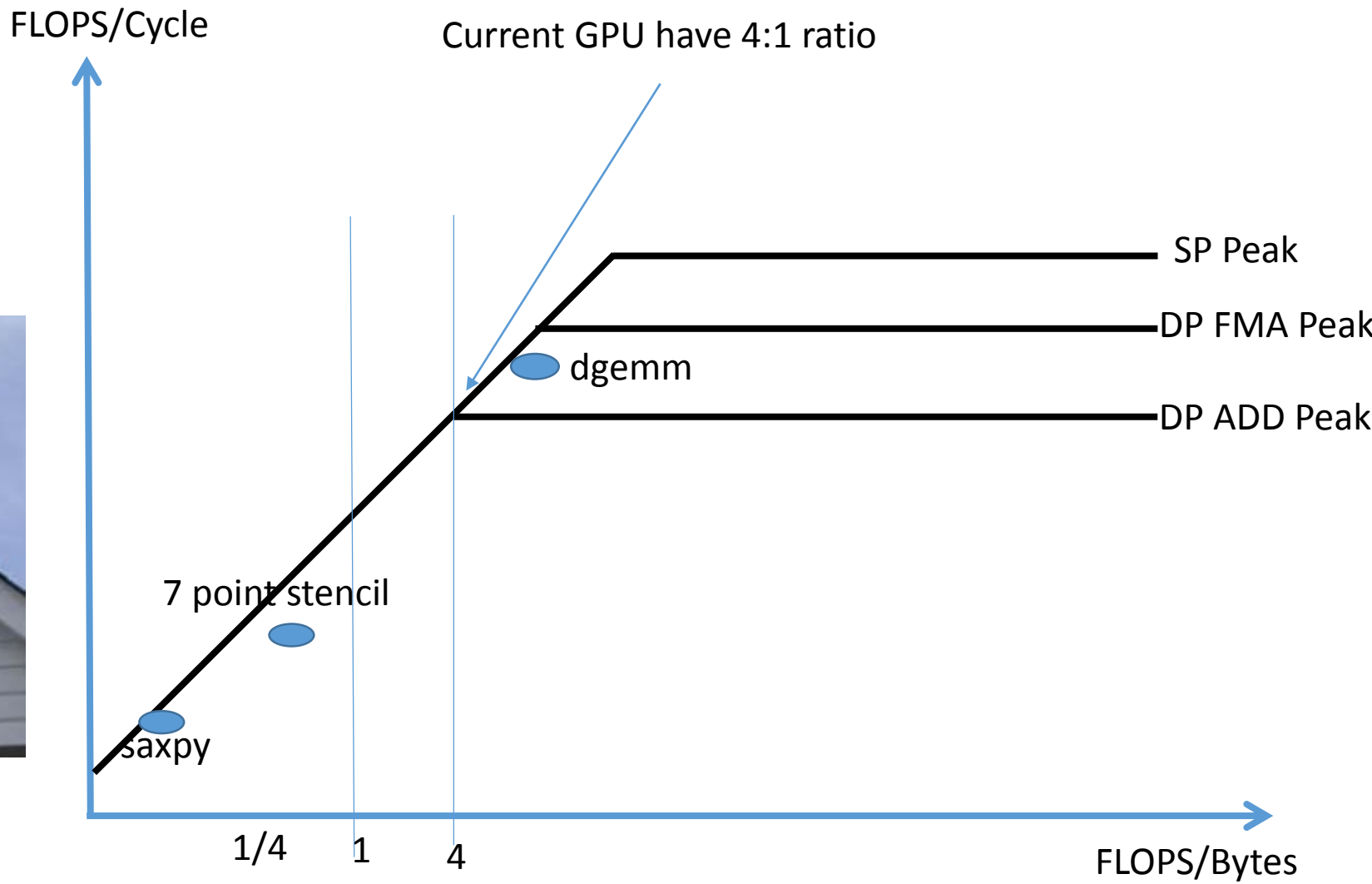
3D




$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{x-1} - 2u_x + u_{x+1}}{\Delta x^2}$$
$$\frac{\partial^2 u}{\partial y^2} = \frac{u_{y-1} - 2u_y + u_{y+1}}{\Delta y^2}$$
$$\frac{\partial^2 u}{\partial z^2} = \frac{u_{z-1} - 2u_z + u_{z+1}}{\Delta z^2}$$

```
3DLaplace = 1/(dx*dx)*(
    T[x-1][y][z]+T[x+1][y][z]+
    T[x][y-1][z]+T[x][y+1][z]+
    T[x][y][z-1]+T[x+1][y][z+1]-6*T[x][y][z]);
```

# Roofline Method (arithematic Intensity)



# Analysis of BHTE

To simulate the real world,
Ideal simulation ( N >1024, dt<10ms, duration > 1hour )

| Dim | Operation | | | | | | | steps need | | | | Total Operation | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | L | etc | N | KOP | MOP | GOP | data | dt(ms) | min | sec | step | GOP | TOP | POP |
| 1D | 3 | 4 | 1024 | 7.2 | 0.0 | 0.0 | n^1 | 10 | 60 | 3600 | 360,000 | 2.58 | 0.00 | 0.00 |
| 2D | 5 | 4 | 1024 | 9437.2 | 9.4 | 0.0 | n^2 | 10 | 60 | 3600 | 360,000 | 3,397.39 | 3.40 | 0.00 |
| 3D | 6 | 4 | 1024 | 10737418.2 | 10737.4 | 10.7 | n^3 | 10 | 60 | 3600 | 360,000 | 3,865,470.57 | 3,865.47 | 3.87 |

Reduce (N = 256, dt = 100ms, duration <15minutes) on CPU

| Dim | Operation | | | | | | | steps need | | | | Total Operation | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | L | etc | N | KOP | MOP | GOP | data | dt(ms) | min | sec | step | GOP | TOP | POP |
| 1D | 3 | 4 | 256 | 1.8 | 0.0 | 0.0 | n^1 | 100 | 15 | 900 | 9,000 | 0.02 | 0.00 | 0.00 |
| 2D | 5 | 4 | 256 | 589.8 | 0.6 | 0.0 | n^2 | 100 | 15 | 900 | 9,000 | 5.31 | 0.01 | 0.00 |
| 3D | 6 | 4 | 256 | 167772.2 | 167.8 | 0.2 | n^3 | 100 | 15 | 900 | 9,000 | 1,509.95 | 1.51 | 0.00 |

Want to see (N = 1024, dt = 100ms, duration <15minutes)

| Dim | Operation | | | | | | | steps need | | | | Total Operation | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | L | etc | N | KOP | MOP | GOP | data | dt(ms) | min | sec | step | GOP | TOP | POP |
| 1D | 3 | 4 | 1024 | 7.2 | 0.0 | 0.0 | n^1 | 100 | 15 | 900 | 9,000 | 0.06 | 0.00 | 0.00 |
| 2D | 5 | 4 | 1024 | 9437.2 | 9.4 | 0.0 | n^2 | 100 | 15 | 900 | 9,000 | 84.93 | 0.08 | 0.00 |
| 3D | 6 | 4 | 1024 | 10737418.2 | 10737.4 | 10.7 | n^3 | 100 | 15 | 900 | 9,000 | 96,636.76 | 96.64 | 0.10 |

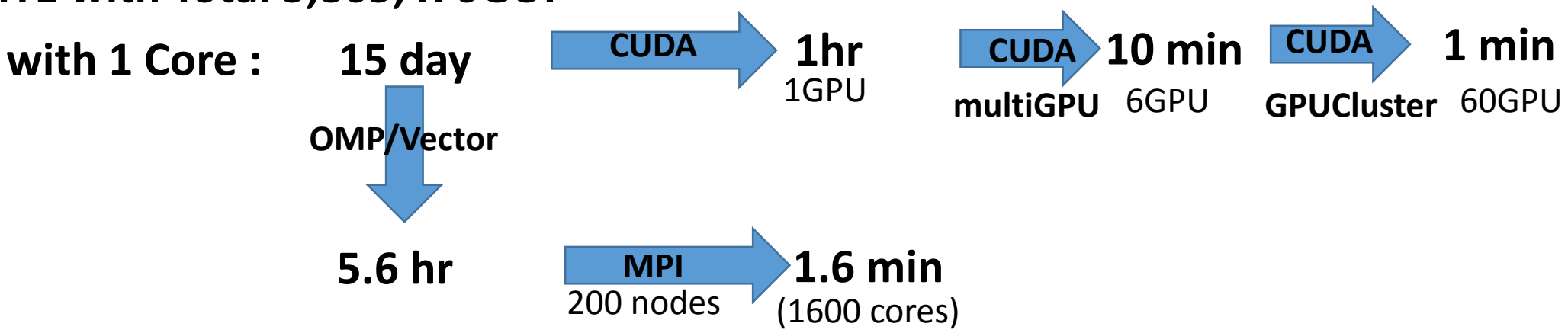
# Expectation of Computation Time

- **Computation Time = Total Operation / Peak Perf of Arch**

**1 Core CPU without vector : 3Ghz, 3GFLOPS**

**8 Core CPU with vector, HT : 3Ghz, 192GFLOPS ( x64 = x8,x4,x2)**

**3D BHTE with Total 3,865,470GOP**



# Naïve CPU Code for FDM of BHTE

```
void BHTE_FDM_Gold( double *T, double* RHO, double* CT,
                    double* Q, double dx, double dt, double kt, double Vb ,
                    double rhob, double Cb, double Tb ){

    for(int x=0; x< x_size; x++){
        for(int y=0; y< y_size; y++){
            for(int z=0; z< x_size; z++){

                Laplace = 1/(dx*dx)*(
                        T[x-1][y][z]+T[x+1][y][z]+
                        T[x][y-1][z]+T[x][y+1][z]-
                        T[x][y][z-1]+T[x+1][y][z+1]-6 * T[x][y][z]);
              ALPHA = dt/(RHO[x][y][z] + CT[X][Y][z]) ;
               BETA = Vb * rhob* Cb ;
        Temp_result = T[x][y][z]+ALPHA *( kt * Laplace + BETA*(Tb-T[x][y][z])+Q[x][y][z]) ;
         TN(x,y,z)  = Temp_result;

            }
        }
    }
```

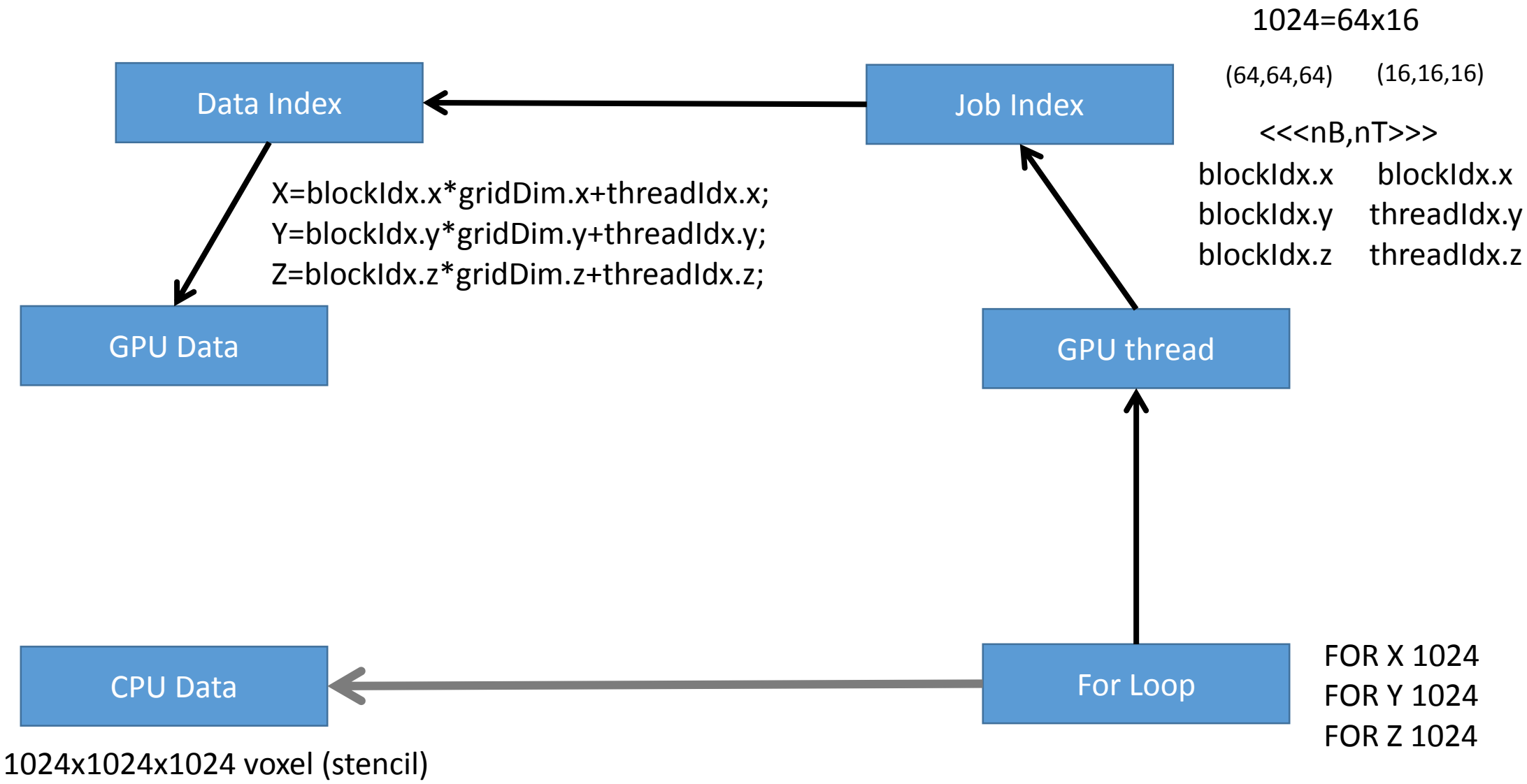
7 point Stencil

# CPU Code with Array Call function

```
void BHTE_FDM_Gold( double *data_heat, double* data_density, double* *data_SPECIFIC_HEAT,,
                    double* data_internalheat, double dx, double dt, double kt, double Vb, ,
                    double rhob, double Cb, double Tb ){

    for(int x=0; x< x_size; x++){
        for(int y=0; y< y_size; y++){
            for(int z=0; z< x_size; z++){
// TODO for each stencil

                L_tn = 1 / (dx*dx) * (
                       T_n( x-1, y  , z  ) + T_n( x+1, y   , z  )
                 +     T_n(   x, y-1, z  ) + T_n(   x, y+1 , z  )
                 +     T_n(   x, y  , z-1) + T_n( x  , y   , z+1)
                 - 6 * T_n( x  , y  , z  )  );

         Temp_result = TN(x,y,z) + dt /( RHOT(x,y,z) + CT(x,y,z) ) *
                       (   kt * L_tn + Vb * rhob* Cb * ( Tb - T_n(x,y,z)  ) +  QN(x,y,z) ) ;
          TN(x,y,z)  = Temp_result;

            }
        }
    }
```

Define fetch function
A_n(x,y,z) ~ A[x][y][z]

# GPU coding [ data idx, thread idx]



# GPU Code [ naïve index control]

**CPU code**

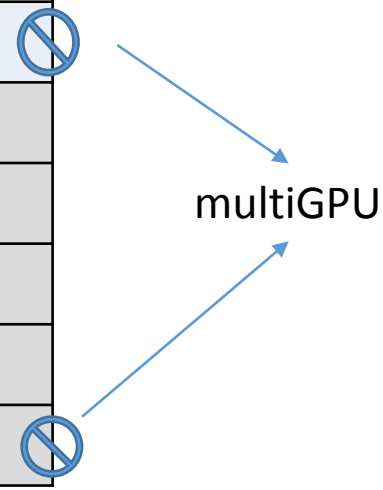
```
void BHTE_FDM_Gold( double *data_heat, double* data_density, double* *data_SPECIFIC_HEAT,
                    double* data_internalheat, double dx, double dt, double kt, double Vb,
                    double rhob, double Cb, double Tb ){
    for(int x=0; x< x_size; x++){
        for(int y=0; y< y_size; y++){
            for(int z=0; z< x_size; z++){
                    // TODO for each stencil
            }
        }
    }
}
```

**GPU code**

```
__global__ void BHTE_FDM_GPU( double * data_heat, double* data_density, double * data_SPECIFIC_HEAT,
                              double* data_internalheat,  double dx, double dt,  double kt,  double Vb,
                              double rhob ,   double Cb, double Tb ){
                int x =  blockIdx.x * (gridDim.x)+ threadIdx.x;
                int y =  blockIdx.y * (gridDim.y)+ threadIdx.y;
                int z =  blockIdx.z * (gridDim.z)+ threadIdx.z;
                  // TODO for each stencil

}
```

# GPU Memory

| | W_size | elemets | MB | GB | units | Total Memory |
|---|---|---|---|---|---|---|
| float | 64 | 2.62E+05 | 1 | 0.00 | 5 | 0.00 |
| | 128 | 2.10E+06 | 8 | 0.01 | 5 | 0.04 |
| | 256 | 1.68E+07 | 64 | 0.06 | 5 | 0.31 |
| | 512 | 1.34E+08 | 512 | 0.50 | 5 | 2.50 |
| | 1,024 | 1.07E+09 | 4,096 | 4.00 | 5 | 20.00 |
| double | 64 | 2.62E+05 | 2 | 0.00 | 5 | 0.01 |
| | 128 | 2.10E+06 | 16 | 0.02 | 5 | 0.08 |
| | 256 | 1.68E+07 | 128 | 0.13 | 5 | 0.63 |
| | 512 | 1.34E+08 | 1,024 | 1.00 | 5 | 5.00 |
| | 1,024 | 1.07E+09 | 8,192 | 8.00 | 5 | 40.00 |



# GPU Coding [ algorithm]

**// TODO for each stencil is Nothing Special!!! Same as CPU code with Array function Call**

```
L_tn = 1 / (dx*dx) * (   T_n( x-1, y   , z   ) + T_n( x+1, y    , z   )
                     +    T_n(   x, y-1, z   ) + T_n(   x, y+1 , z   )
                     +     T_n(   x, y   , z-1) + T_n( x   , y    , z+1)
                     - 6 * T_n( x   , y   , z   )
                        );

Temp_result = TN(x,y,z)    dt /(  RHOT(x,y,z) + CT(x,y,z) )
              * ( kt * L_tn  + Vb * rhob* Cb * ( Tb - T_n(x,y,z)  )
               + QN(x,y,z) ) ;
TN(x,y,z)   = Temp_result;
}
```

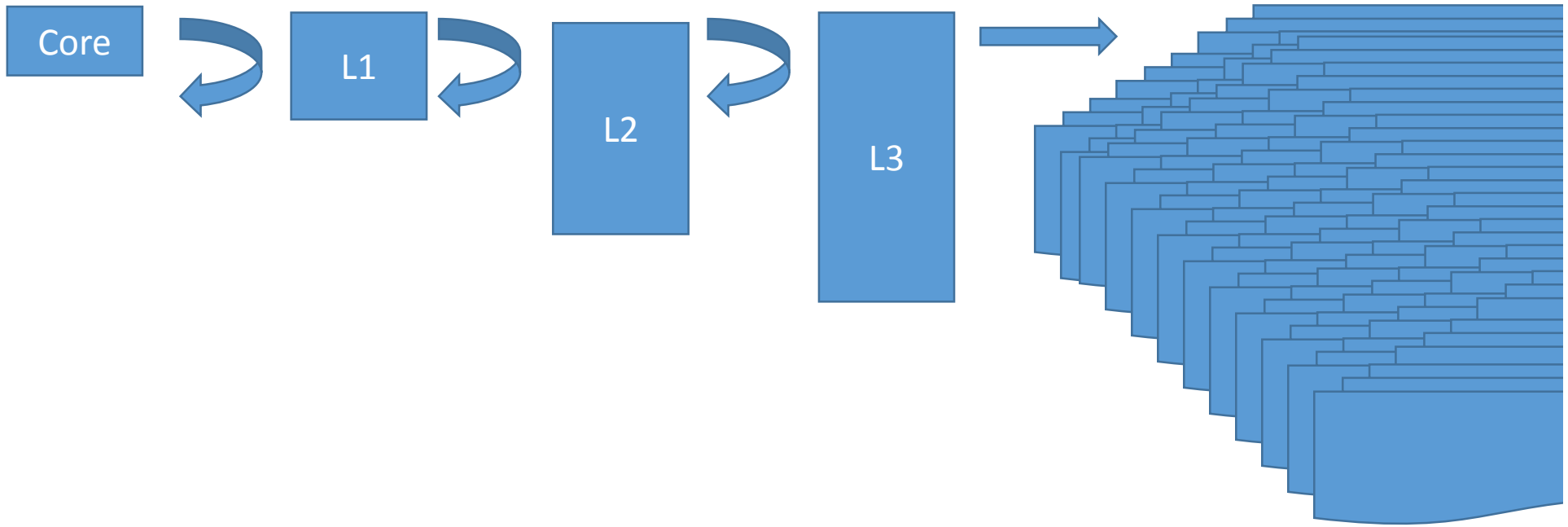


# Optimization Tips

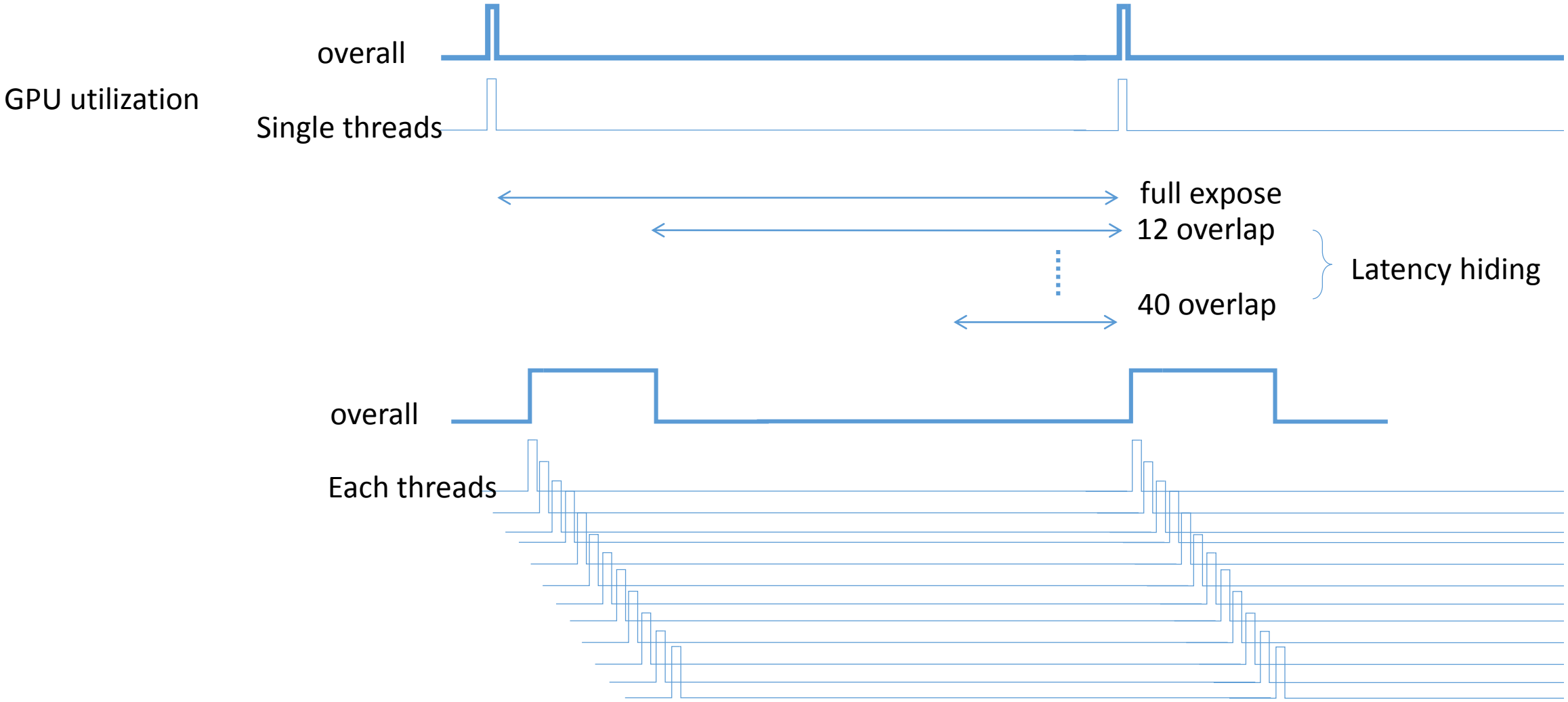
**More Optimization tip,**
**Use latest CUDA profiler and check CUDA Best Practice Guide**

# Concept of Latency Reduction

Cache hit will reduce the memory latency



# Simple Concept of Latency Hiding



# Summary

- Heterogenous HPC
  - GPU H/W vs CPU H/W
  - OpenACC 101
  - CUDA 101
- CUDA Examples
  - helloCUDA
  - pyCUDA
  - Transpose Example
  - 3D BHTE Example
- CUDA Optimization tip
  - Example of Latency Reduction technique
  - CUDA profiler and CUDA Best Practice Guide

# Thank You
# ありがとうございます
# 감사합니다.