Accelerated Computing 2

GPGPU Programming

17/Feb/2016

presented by Seungmin Lee

KiSTi 한국과학기술정보연구원 Korea Institute of Science and Technology Information www.kisti.re.kr

Supercomputing Service Center

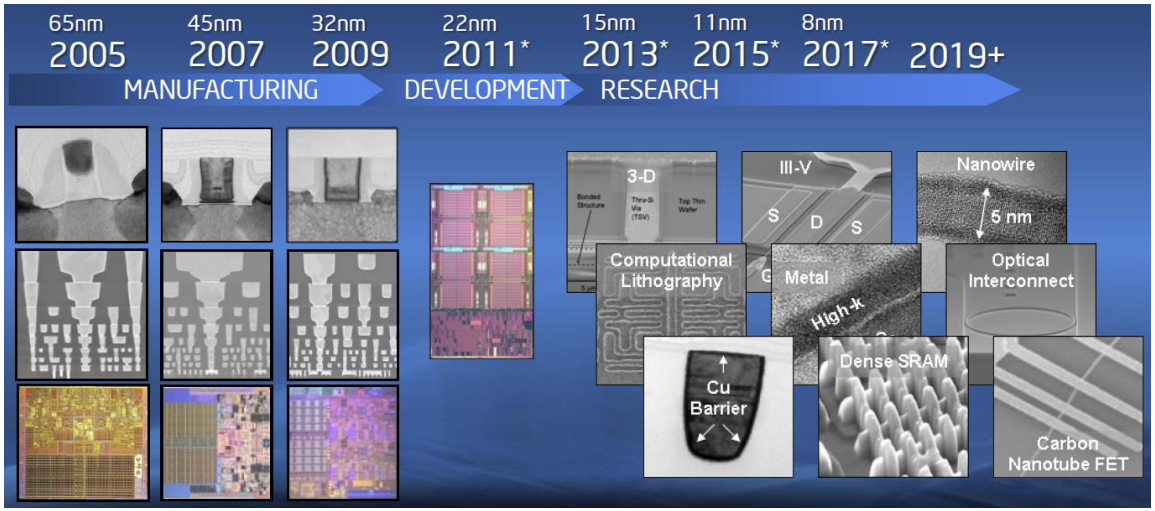
Outline

- Introduction
  - Evolution of Processor
  - History of GPU Computing
- GPGPU Programming
  - OpenACC
  - OpenMP
  - CUDA
  - OpenCL

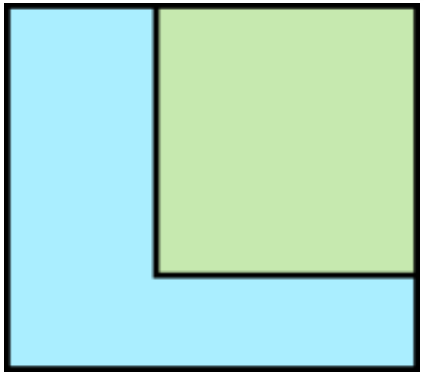
Evolution of Processor

Moore's Law

- The number of transistors on a chip will double about every 1.5 years

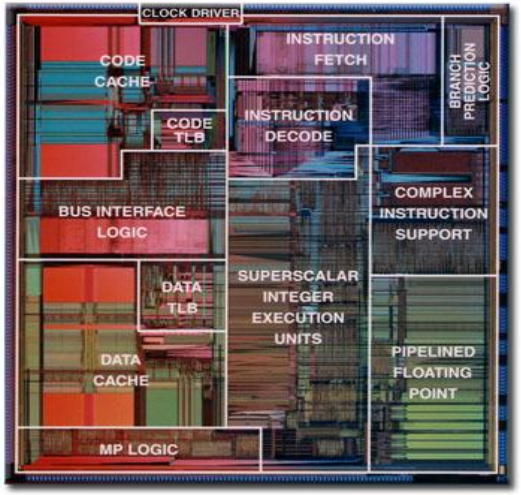


$$\frac{1}{\sqrt{2}} \cong 0.7$$

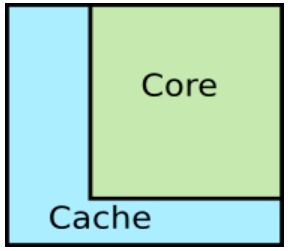


Evolution of Processor (Cont.)

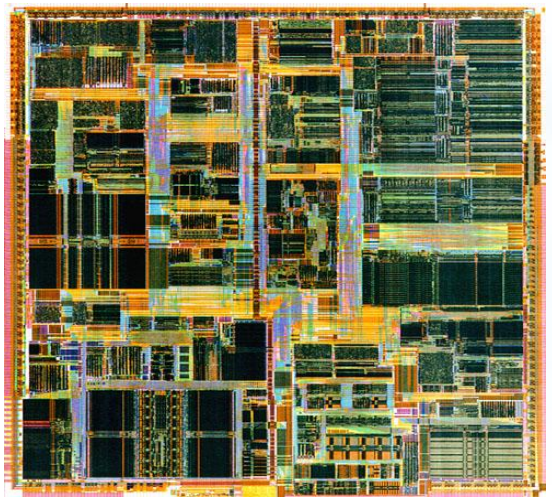
Pentium I

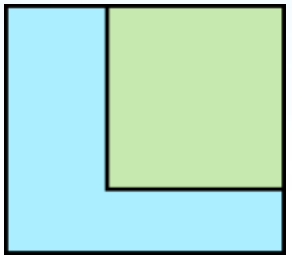


Chip area breakdown

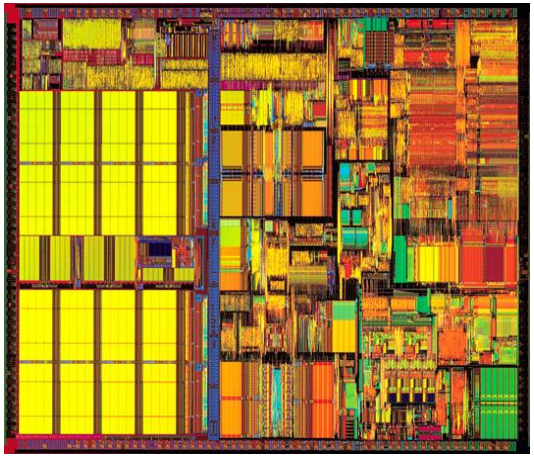


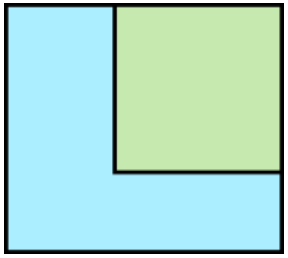
Pentium II



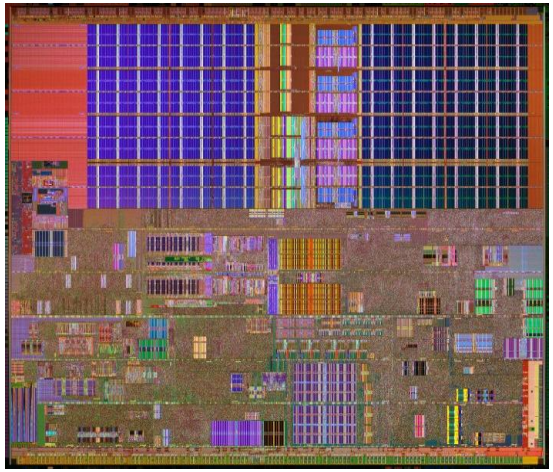


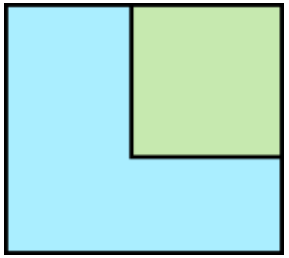
Pentium III





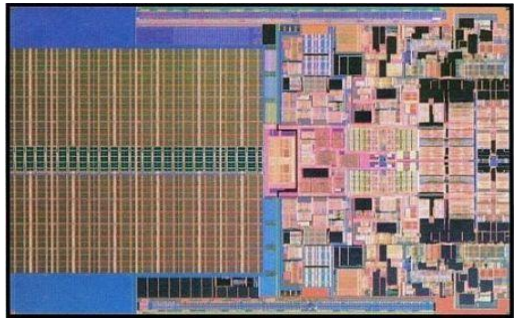
Pentium IV



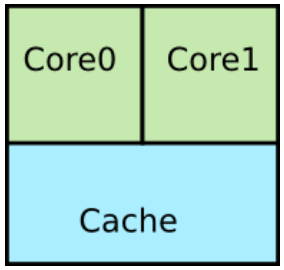


Evolution of Processor (Cont.)

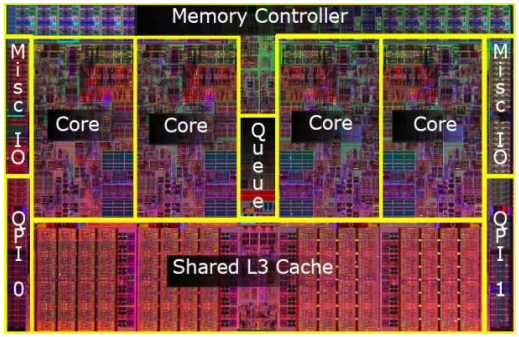
Penryn

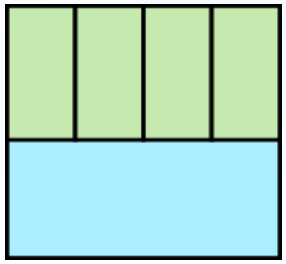


Multi-core

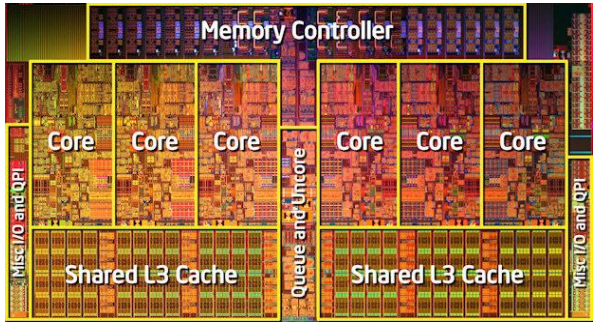


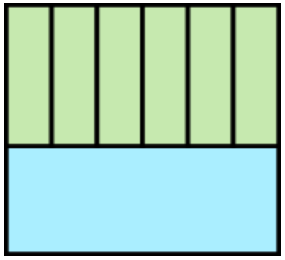
Bloomfield



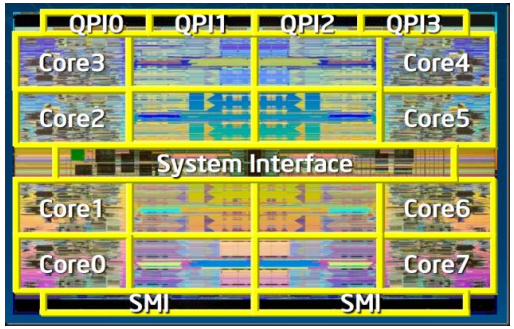


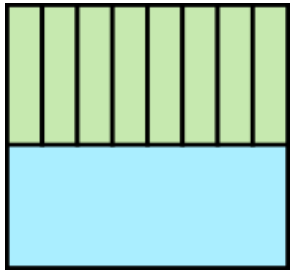
Gulftown





Beckton





HPC vs HTC

- HPC (High Performance Computing)
  - Metric : FLOPS
- HTC (High Throughput Computing)
  - Metric : jobs/day or month

- An extreme example
  - Two processors
    - A : 4 cores, 10 GFLOPS/core
    - B : 50 cores, 1 GFLOPS/core
  - 100 jobs : 100 GFLOP / job
  - Execution time of 1 job
    - 2.5 seconds for A, 2 seconds for B
  - Execution time of 100 jobs
    - 250 seconds for A : 0.4 jobs/s
    - 200 seconds for B : 0.5 jobs/s

Calculation FLOPS

- FLOPS : Floating-point Operations Per Second
  - Clock (Hz), # of cores, SIMD, FMA(MAD)

    ex) KISTI TachyonII System

    $2.93 \times 10^9 \times 25{,}408 \times 2 \times 2$

    $= 297{,}781.76 \times 10^9 \fallingdotseq 300 \times 10^{12}$

1st Supercomputer (1988) : 2 GFlops

2nd Supercomputer (1993) : 16 GFlops

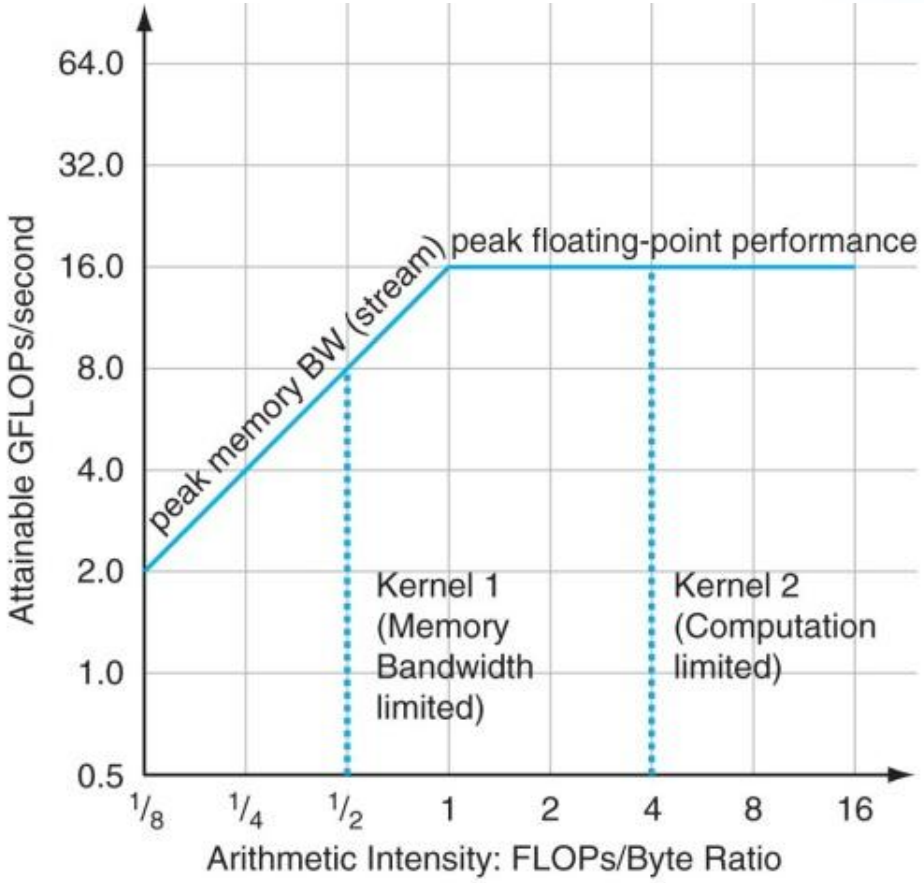
3rd Supercomputer (2004) : 4.3 TFlops

Rpeak	300TFLOPS
Nodes(#)	3,176
Processor	Intel Xeon X5570 2.93GHz
CPU(#)	25,408
Memory	76.8 TB
Storage	1,061 TB

Notebook(3GHz, Quad-core) : $3 \times 10^9 \times 4 \times 4 \times 2 = 96$ GFlops

Arithmetic Intensity (AI)

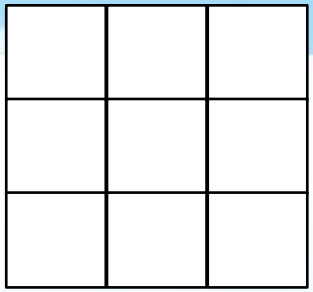
- The number of float-point operations to run the program divided by the number of bytes accessed in main memory



Roofline Model [Williams,Patterson, 2008]

Arithmetic Intensity (AI)

Intel Xeon E5690 (6 cores, 3.73 GHz, 32GB/s)
FLOPS : 3.73 * 6 * 4 * 2 = 179.04 GFLOPS
Flop:byte ratio : 5.595



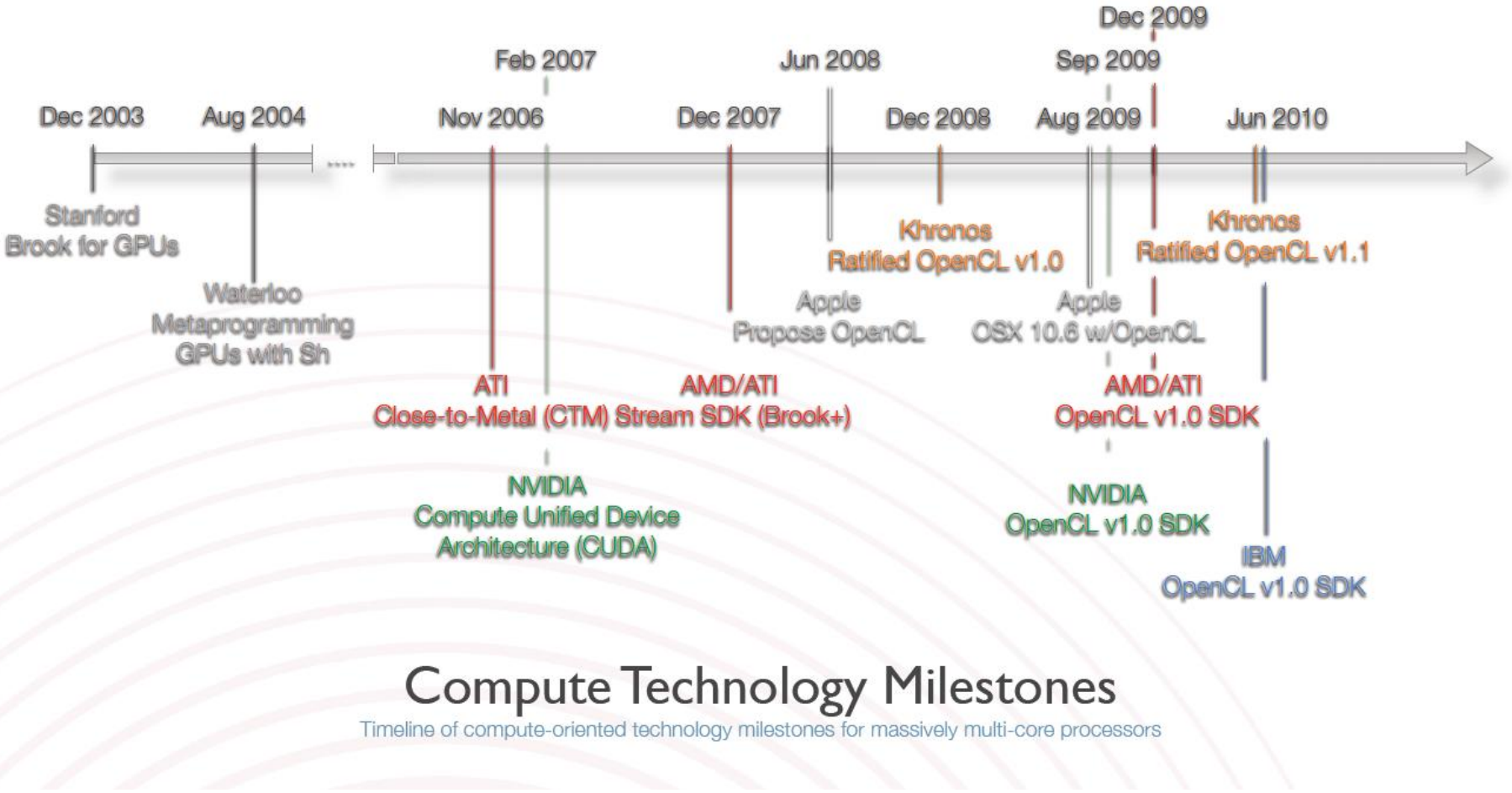

```
for( i = 0;  i  < N;  i++)                          3N loads, 1N stores, 2N flop
      A[i] =  B[i] + C[i] * D[i];                   (6 * 2 * N) / (4 * 8 * N) = 0.375


for( i = 0;  i  < N;  i++)                          N^2 loads, N^2 stores, 2N^3 loads, 2N^3 flop
       for( j = 0;  j  <  N;  j++)                  (6 * 2N^3) / (8 * (2N^3 + 2N^2))  ≈ 0.75
             for( k = 0;  k  < N;  k++)
                   C[i][j] +=  A[i][k] * B[k][j];


for( ii=0; ii< N; ii+= NB)
for( jj=0; jj < N; jj+=NB)
for( i = ii;  i  < NB+ii;  i++)                     3NB^2 loads, NB^2 stores , 2NB^3 flop
       for( j = jj;  j  <  NB+jj;  j++)             (6 * 2NB^3) / (8 * 4NB^2)  =  0.375*NB
             for( k = 0;  k  < NB;  k++)
                   C[i][j] +=  A[i][jj+k] * B[ii+k][j];
```

32KB L1 cache
3 matrices, 8 bytes(double)

$$NB = \sqrt{\frac{32 \times 2^{10}}{3 \times 8}} \cong 36.9$$

$$0.375 \times 32 = 12$$

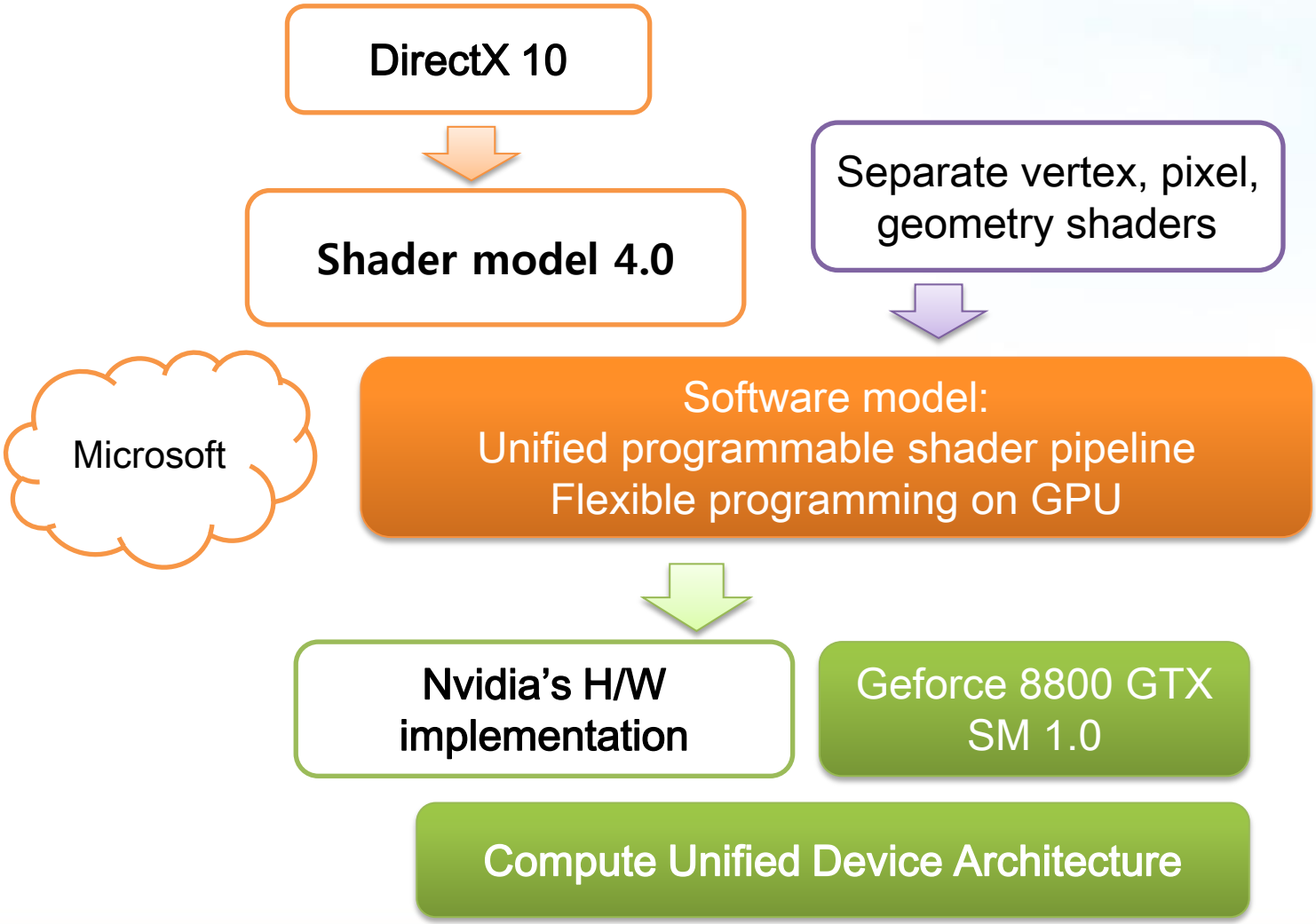
Brief History of GPU Computing



Source: SIGGRAPH Asia 2010 OpenCL Overview tutorial

History of GPU Computing (Cont.)

Development 2006 ~ 2007

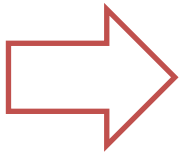


History of GPU Computing (Cont.)

C for Graphics(Cg)

Brook+

DirectX 10



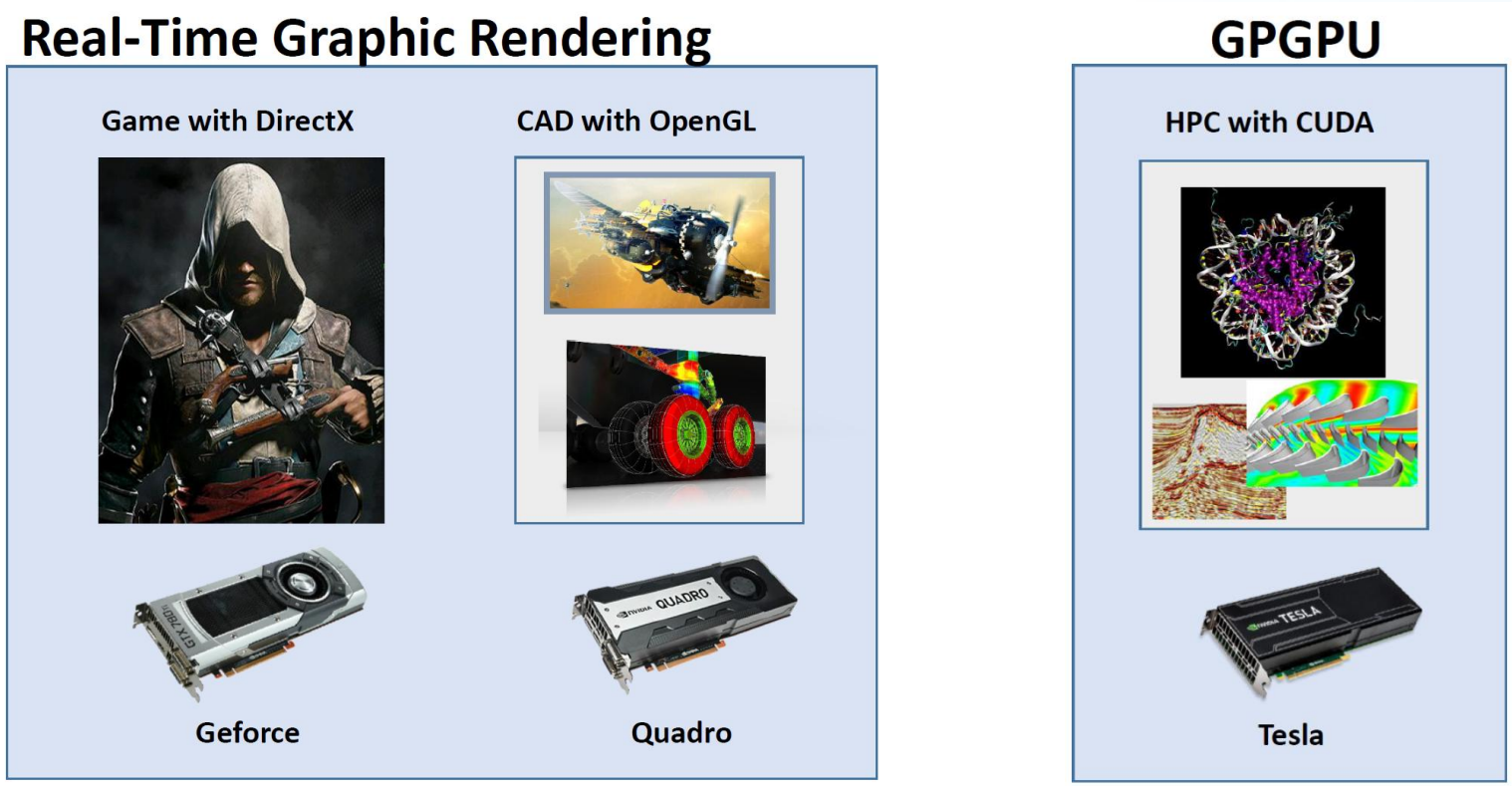
CUDA
Compute Unified
Device Architecture

CAL
Compute Abstraction Layer

OpenCL
Open Computing Lanaguage

GPGPU (General Purpose Graphic Processing Unit)

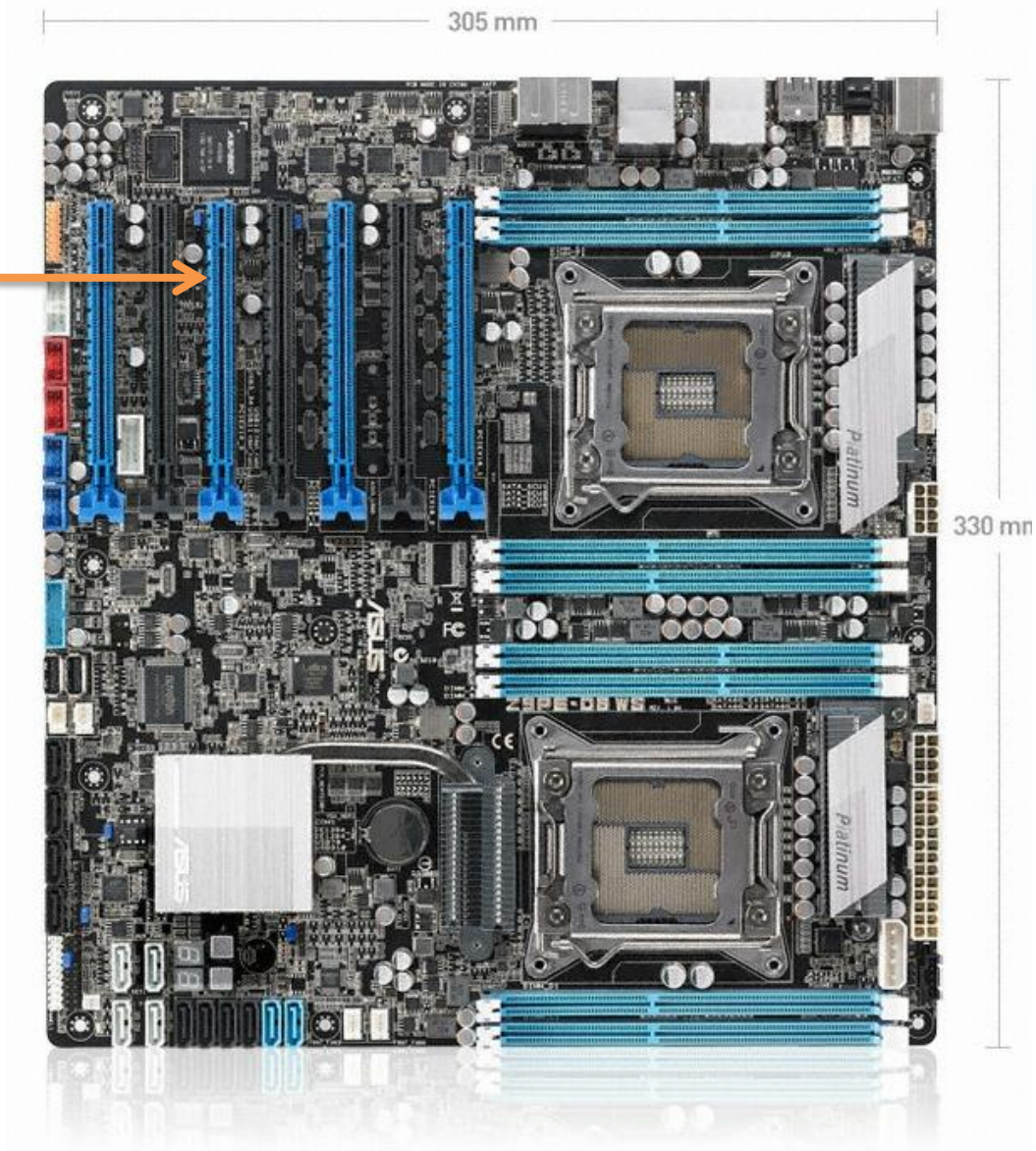
- GPGPU stands for General-Purpose computation on Graphics Processing Units, also known as GPU Computing
- GPGPU with Cg, OpenGL, DirectX, sh, Brook, RapidMind, PeakStream, Brook++, CAL, CTM, CUDA, OpenGL Computing, DirectXCompute, MS AMP, OpenCL



Ref. Acclerated Computing 1: GPGPU Programming and Computing, Korea-Japan HPC Winter School 2014

GPGPU (Cont.)

GPUs are installed here.



GPGPU (Cont.)

- ORNL(Oak Ridge National Laboratory)
  - TITAN => SUBMIT (> 150 PFLOPS)
- LLNL (Lawrence Livermore National Laboratory)
  - SEQUOIA => SIERRA (> 100 PFLOPS)

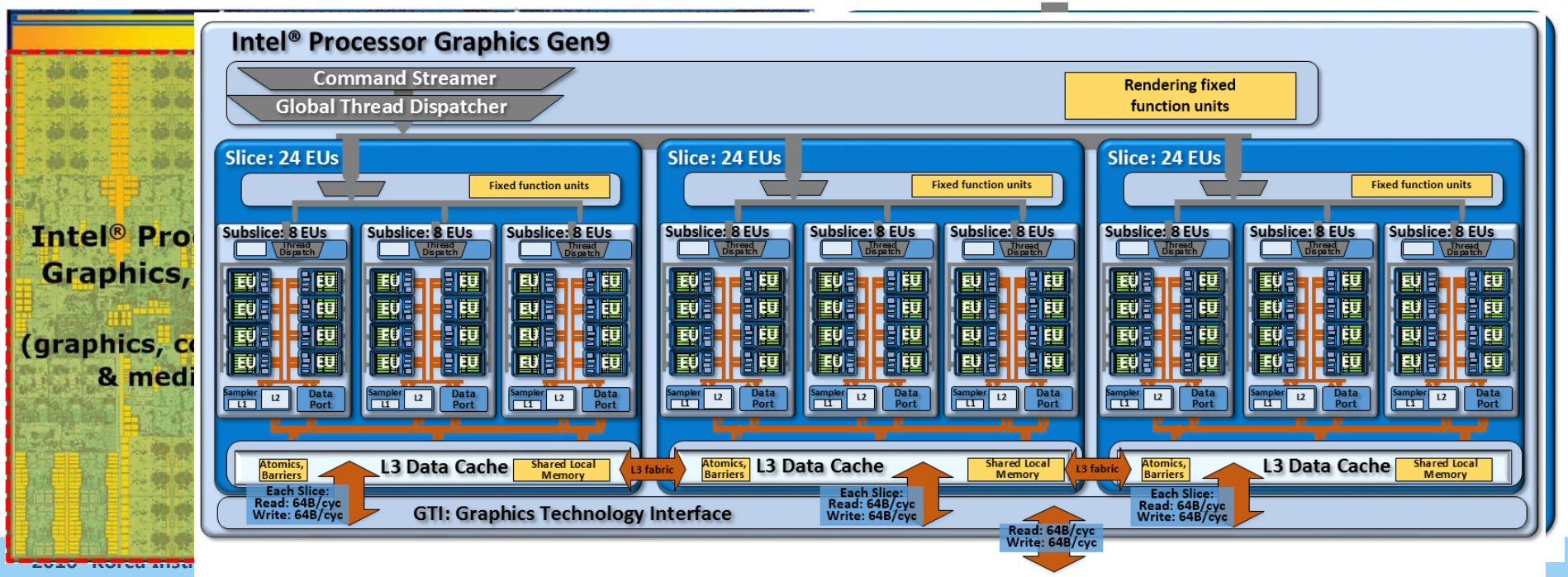
Ref. http://www.teratec.eu/actu/calcul/Nvidia_Coral_White_Paper_Final_3_1.pdf

GPGPU (Cont.)

- Intel
  - Intel Skylake Gen9 GT4/e
  - 1152 GFlops (GPU only)
    - 72 x 2 x 8 x 1 = 1152

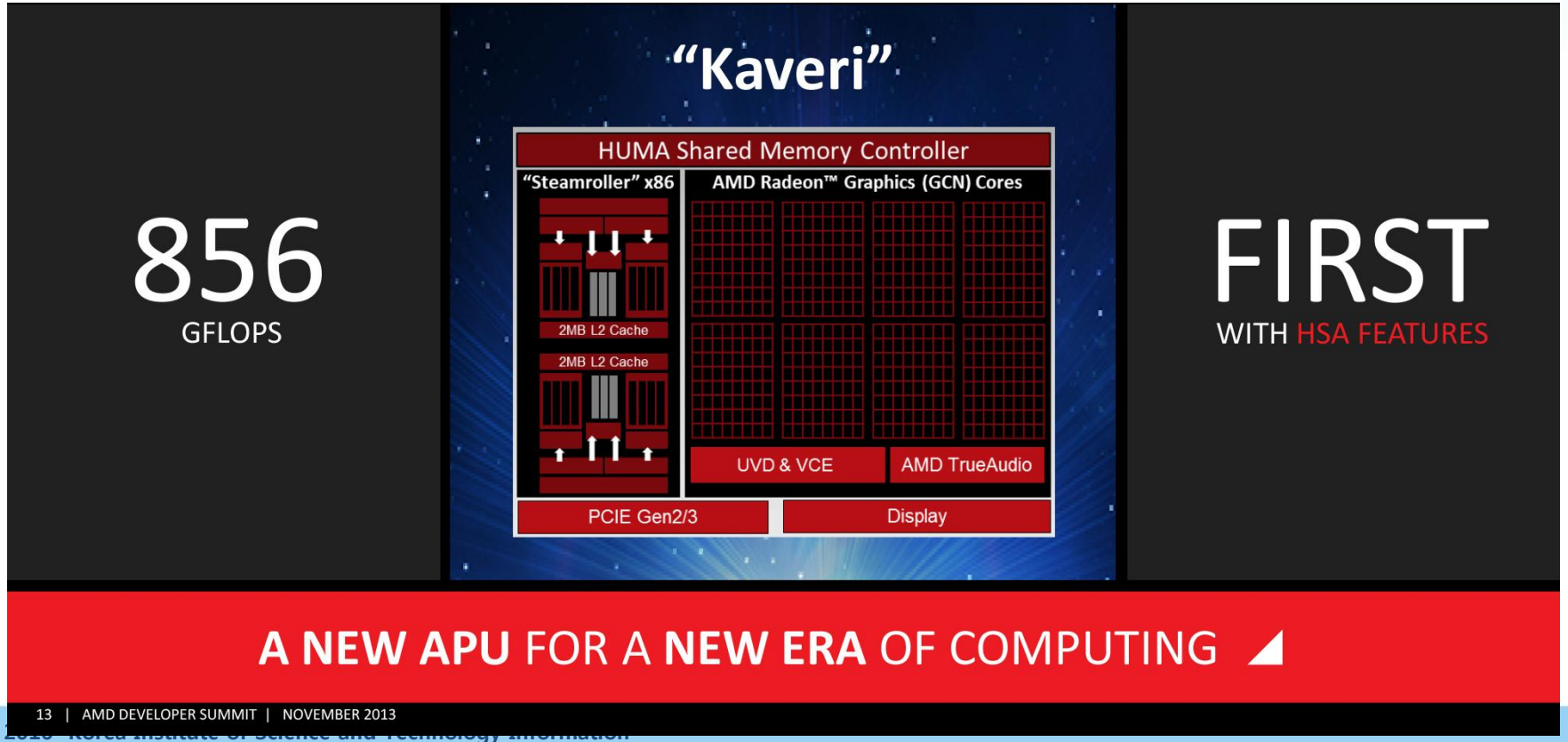
6 Years of Processor Graphics

2010	2011	2012	2013	2014	2015
Iron Lake Intel® HD Graphics	**Sandy Bridge** Intel HD 3000-2000	**Ivy Bridge** Intel HD 4000-2500	**Haswell** Intel HD 5200-4200	**Broadwell** Intel HD 6200-5500	**Skylake** Intel HD 530
Intel® Core™ Processor	2nd Generation Intel Core Processor	3rd Generation Intel Core Processor	4th Generation Intel Core Processor	5th Generation Intel Core Processor	6th Generation Intel Core Processor
• 32nm		• 22nm		• 14nm	
• DirectX® 10.0	• DirectX 10.1	• DirectX 11.0	• DirectX 11.1 • DX Extensions	• DirectX 11.2	• DirectX 12.0
• Up to 10 EUs	• Up to 12 EUs	• Up to 16EUs	• Up to 40 EUs • EDRAM • Iris™ Pro, Iris™	• Up to 48 EUs • EDRAM • Iris Pro, Iris	• Up to 72 EUs • EDRAM+ • Iris Pro, Iris
43 GFLOPS†	**130** GFLOPS†	**256** GFLOPS†	**640** GFLOPS†	**768** GFLOPS†	**1152** GFLOPS†



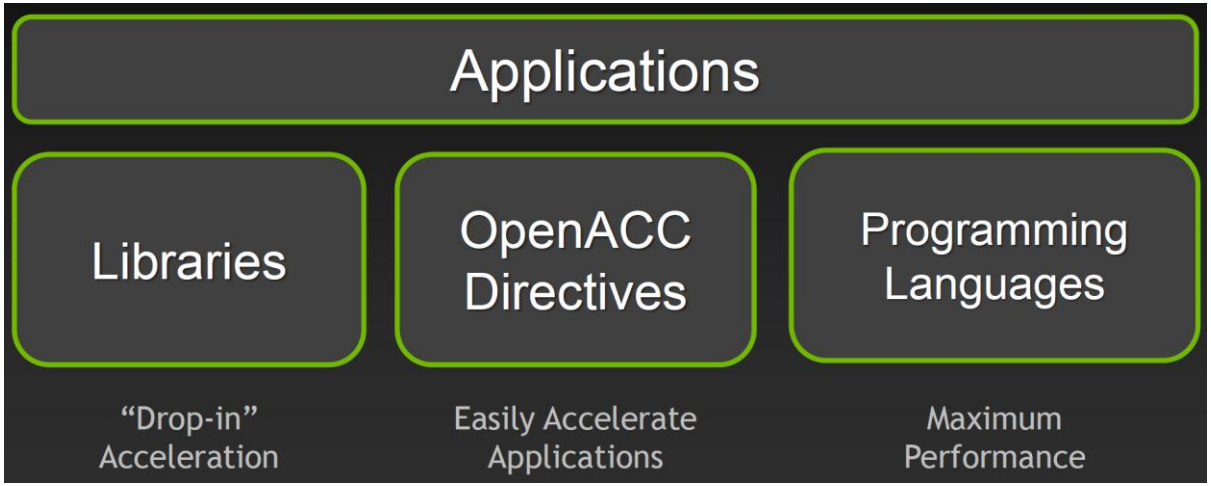
GPGPU (Cont.)

- AMD APU(Accelerated Processing Unit)
  - APU Kaveri (Nov. 2013) : 855.68 (GFlops)
  - CPU : 3.7 GHz x 4 x 4 x 2 = 118.4 (GFlops)
  - GPU : 0.72 GHz x 512 x 2 = 737.28 (GFlops)



GPGPU Programming

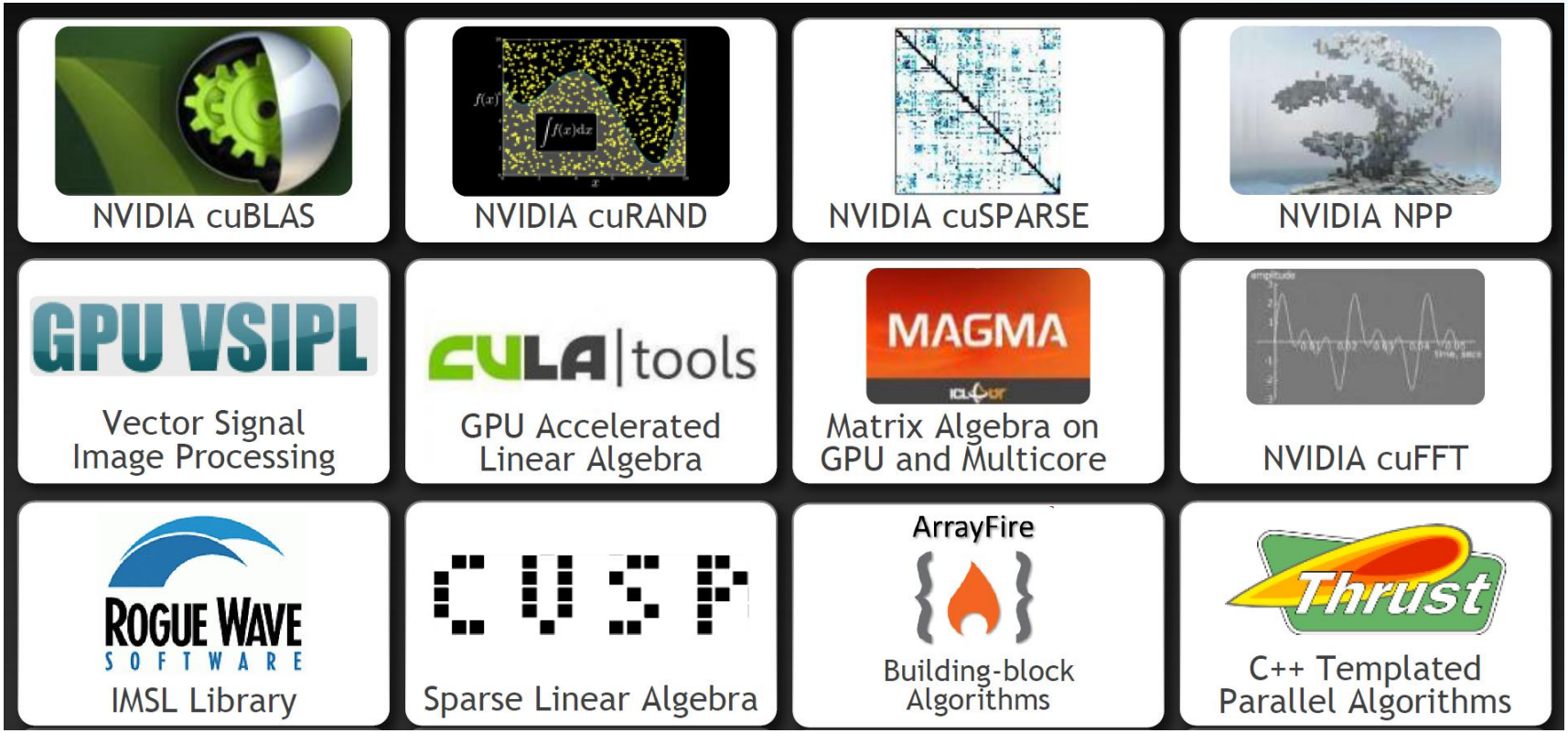
- 3 ways to accelerate applications
  - Libraries, OpenACC, CUDA



Ref. Acclerated Computing 1: GPGPU Programming and Computing, Korea-Japan HPC Winter School 2015

- 3 ways to accelerate applications
  - Libraries, OpenMP, OpenCL

GPU Accelerated libraries



Ref. Acclerated Computing 1: GPGPU Programming and Computing, Korea-Japan HPC Winter School 2015

Drop-in Acceleration

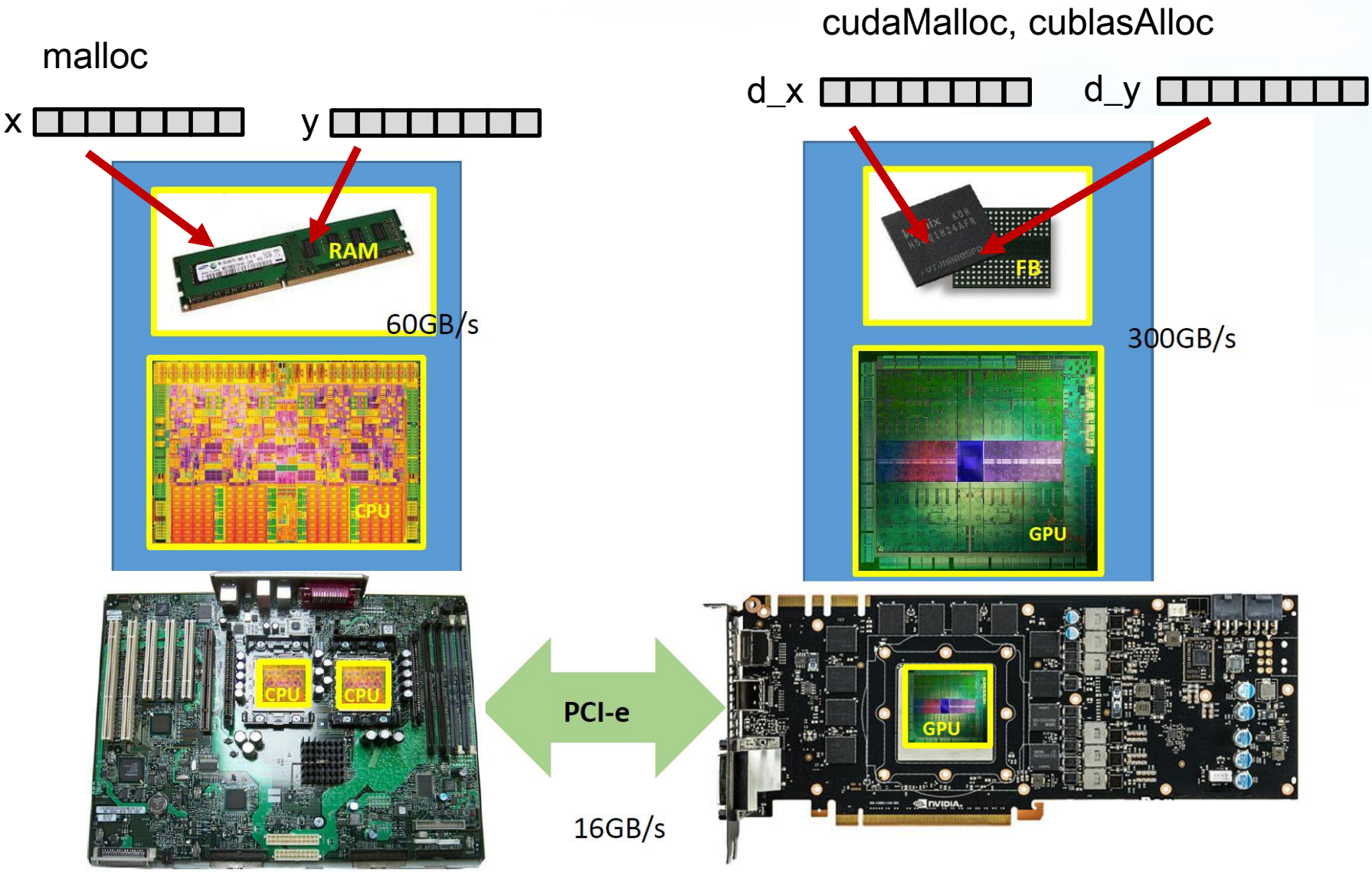
```
int N = 1 << 20;
cublasInit();
cublasAlloc(N, sizeof(float), (void**)&d_x);
cublasAlloc(N, sizeof(float), (void*)&d_y);

cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1);
cublasSetVector(N, sizeof(y[0]), y, 1, d_y, 1);

// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1);

cublasFree(d_x);
cublasFree(d_y);
cublasShutdown();
```

Transfer data to GPU

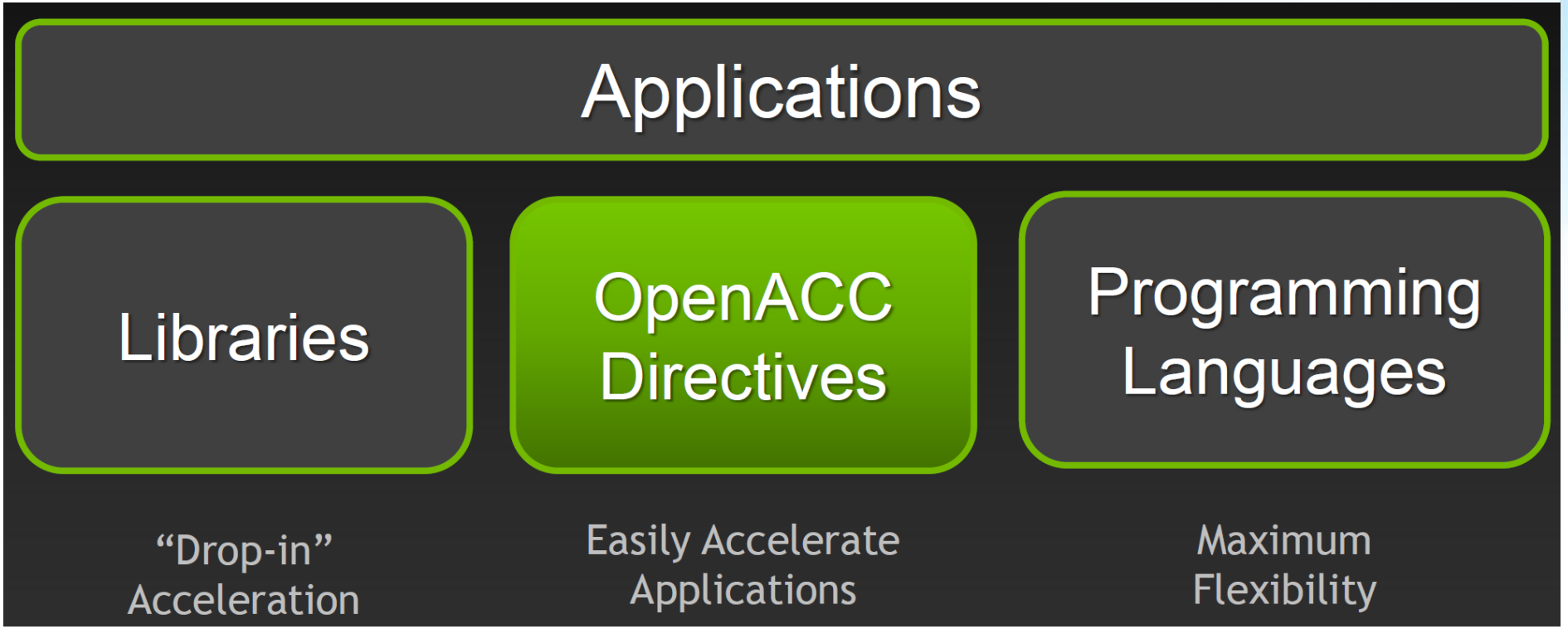
Read data back GPU

Ref. Acclerated Computing 1: GPGPU Programming and Computing, Korea-Japan HPC Winter School 2015

GPGPU (Cont.)


malloc
x
y
RAM
60GB/s
CPU
CPU
CPU
cudaMalloc, cublasAlloc
d_x
d_y
FB
300GB/s
GPU
GPU
PCI-e
16GB/s

3 ways to accelerate applications



Ref. Acclerated Computing 1: GPGPU Programming and Computing, Korea-Japan HPC Winter School 2015

OpenACC (Cont.)

Directive Syntax

- Fortran

  `!$acc directive [clause [,] clause] …]`

  Often paired with a matching end directive surrounding a structured code block

  `!$acc end directive`

- C

  `#pragma acc directive [clause [,] clause] …]`

  Often followed by a structured code block

Ref. Acclerated Computing 1: GPGPU Programming and Computing, Korea-Japan HPC Winter School 2015

OpenACC (Cont.)

Familiar to OpenMP Programmers

SAXPY in C

```
void saxpy(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];
}

...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

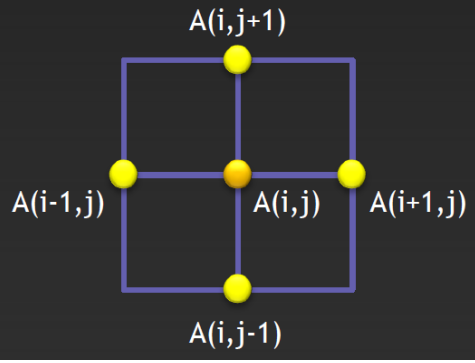
SAXPY in Fortran

```
subroutine saxpy(n, a, x, y)
  real :: x(:), y(:), a
  integer :: n, i
$!acc kernels
  do i=1,n
    y(i) = a*x(i)+y(i)
  enddo
$!acc end kernels
end subroutine saxpy

...
$ Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...
```

Ref. Acclerated Computing 1: GPGPU Programming and Computing, Korea-Japan HPC Winter School 2015

Example : Jacobi Iteration

- Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points
  - Common, useful algorithm
  - Example : Solve Laplace equation in 2D: $\nabla^2 f(x,y) = 0$



$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

Ref. Acclerated Computing 1: GPGPU Programming and Computing, Korea-Japan HPC Winter School 2015

Jacobi Iteration C Code

```
while ( error > tol && iter < iter_max ) {
  error=0.0;

  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {


      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);


      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }

  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

Ref. Acclerated Computing 1: GPGPU Programming and Computing, Korea-Japan HPC Winter School 2015

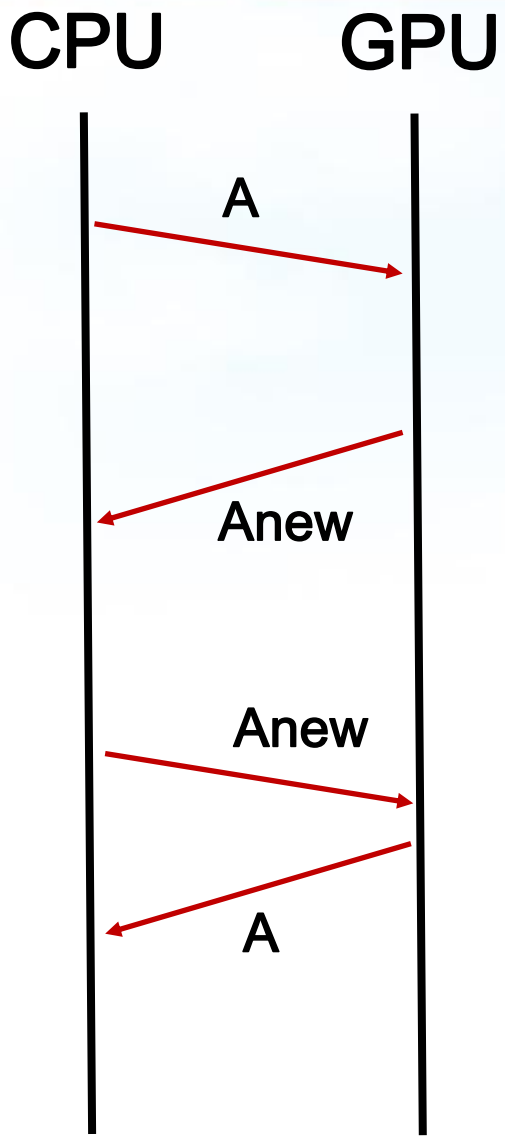
OpenMP C Code

```
while ( error > tol && iter < iter_max ) {
  error=0.0;

#pragma omp parallel for shared(m, n, Anew, A)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }

#pragma omp parallel for shared(m, n, Anew, A)
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

Ref. Acclerated Computing 1: GPGPU Programming and Computing, Korea-Japan HPC Winter School 2015

OpenACC C Code

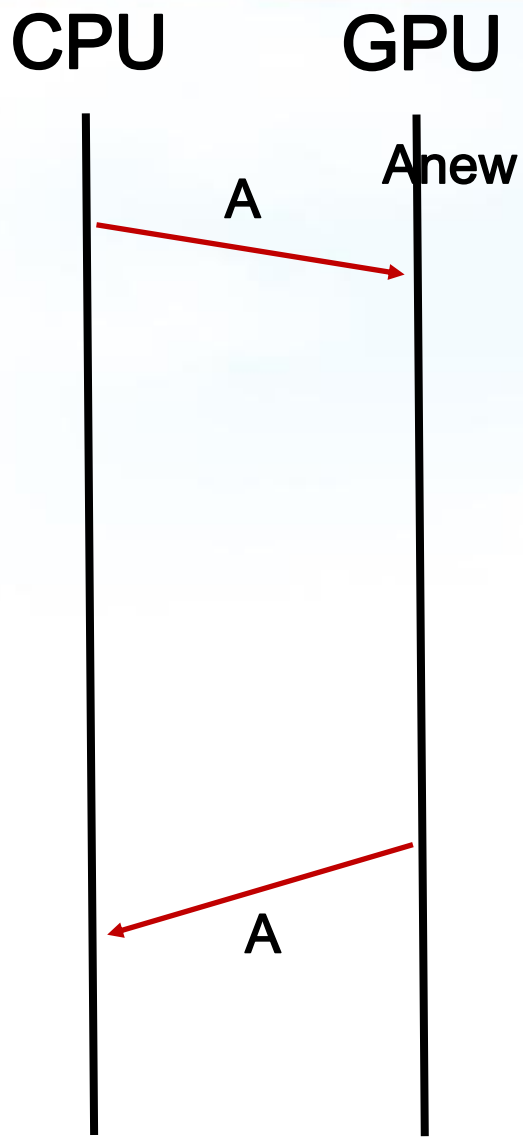
```
while ( error > tol && iter < iter_max ) {
  error=0.0;

#pragma acc kernels
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }

#pragma acc kernels
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```



Ref. Acclerated Computing 1: GPGPU Programming and Computing, Korea-Japan HPC Winter School 2015

OpenACC C Code with Data Management

```
#pragma acc data copy(A), create(Anew)
while ( error > tol && iter < iter_max ) {
  error=0.0;

#pragma acc kernels
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }

#pragma acc kernels
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```



Ref. Acclerated Computing 1: GPGPU Programming and Computing, Korea-Japan HPC Winter School 2015

OpenMP 4.0

- Approved in 2013
- Accelerator device extension

- Directive Syntax
  - #pragma omp target
  - #pragma omp target map(…)

```
long int reduce_openmp4(const int * host_ptr, const int n)
{
    long int sum = 0;
#pragma omp target map(to: host_ptr[0:n]) map(from: sum)
{
#pragma omp parallel for reduction(+:sum)
    for(int i=0; i<n; i++)
        sum += host_ptr[i];
}
    return sum;
}
```

- From GNU gcc 4.9.1, OpenMP 4.0 is fully supported.
- However
  - It is possible for CPU and Intel Xeon Phi
  - It will be available AMD/ATI graphic card from 2016 (expected)
  - No information related to NVIDIA GPU

Compile and Run

- C:

```
pgcc -acc -ta=nvidia -Minfo=accel -o saxpy_acc saxpy.c
```

- Fortran:

```
pgf90 -acc -ta=nvidia -Minfo=accel -o saxpy_acc saxpy.f90
```

gcc 6 in Mac OS X (supports OpenMP 4.0)

```
gcc  -fopenacc  -fopenmp  -o sum.x  sum.c
```

CUDA Programming

```
int N = 1 << 20;
cublasInit();
cublasAlloc(N, sizeof(float), (void**)&d_x);
cublasAlloc(N, sizeof(float), (void*)&d_y);

cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1);
cublasSetVector(N, sizeof(y[0]), y, 1, d_y, 1);

// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1);

cublasFree(d_x);
cublasFree(d_y);
cublasShutdown();
```

cudaMalloc

cudaMemcpy

saxpy <= implement

cudaMemcpy

cudaFree

CUDA Programming

```
__global__ void vecAdd(const float *A, const float *B, float *C, int numElements)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if( i < numElements )
        C[i] = A[i] + B[i];
}

int main(void)
{
    int numElements = 5000;
    size_t size = numElements * sizeof(float);

    float *h_A = (float *)malloc(size);
    float *h_B = (float *)malloc(size);
    float *h_C = (float *)malloc(size);

    for (int i = 0; i < numElements; ++i) {
        h_A[i] = rand()/(float)RAND_MAX;
        h_B[i] = rand()/(float)RAND_MAX;
    }

    float *d_A = NULL, *d_B = NULL, *d_C = NULL;
    cudaMalloc((void **)&d_A, size);
    cudaMalloc((void **)&d_B, size);
    cudaMalloc((void **)&d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Launch the Vector Add CUDA Kernel
    int threadsPerBlock = 100;
    int blocksPerGrid = (numElements + threadsPerBlock -1 ) / threadsPerBlock;
    vecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

CUDA Programming

```
__global__ void vecAdd(const float *A, const float *B, float *C, int numElements)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if( i < numElements )
        C[i] = A[i] + B[i];
}

// Launch the Vector Add CUDA Kernel
int threadsPerBlock = 100;
int blocksPerGrid = (numElements + threadsPerBlock -1 ) / threadsPerBlock;
vecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);
```

- CUDA Kernels
  - kenel_function<<<num_blocks, num_threads>>>(param1, param2, …)

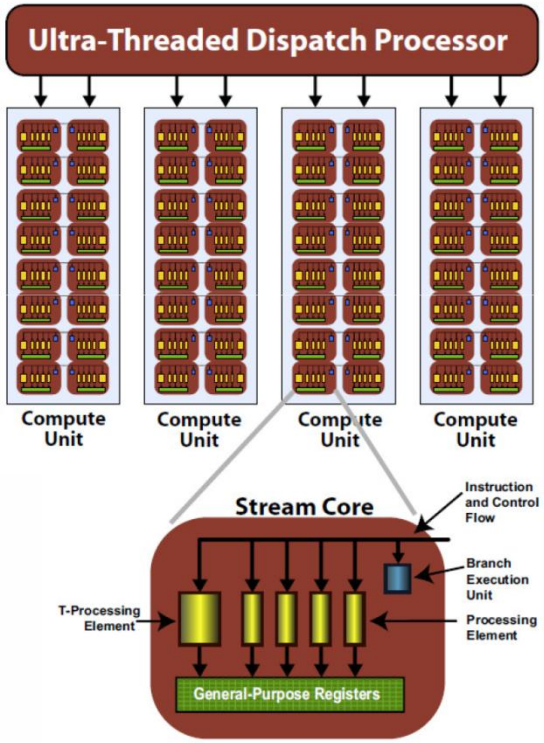
    num_threads = 256, num_blocks = 20

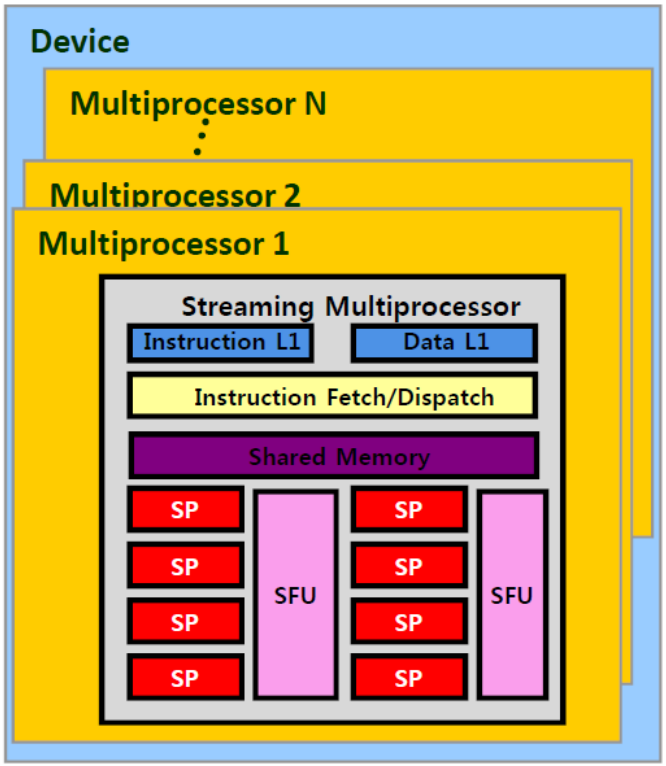
    total # of threads created = 256 x 20 = 5120
- Inside kernel function

  blockDim.x = 256 (num_threads)

  136^{th} threads in 19^{th} block (index starts 0) = 19 x 256 + 136 = 5000

CUDA Programming





NVIDIA GPU

GPU

Multiprocessor

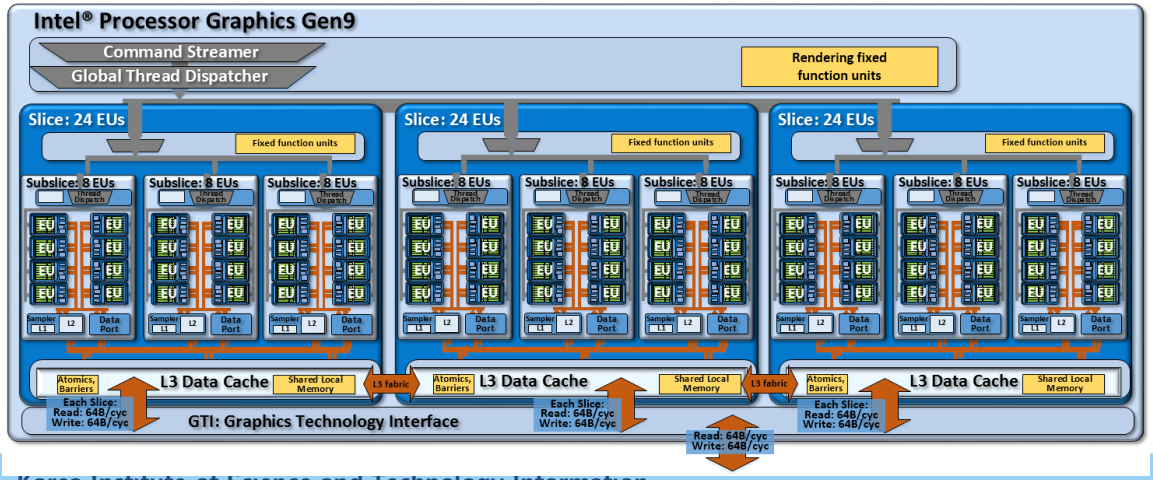
Streaming Processor

AMD GPU

GPU

Compute Unit

Stream Core



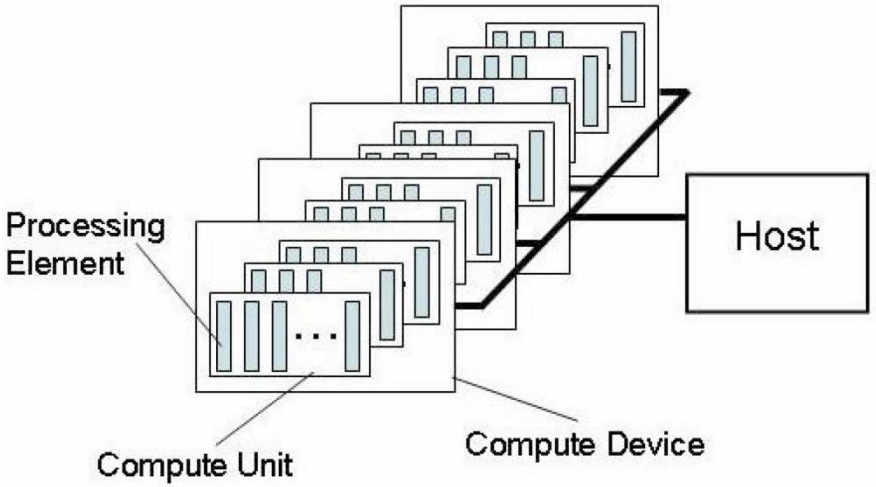
Intel GPU

Slice

subslice

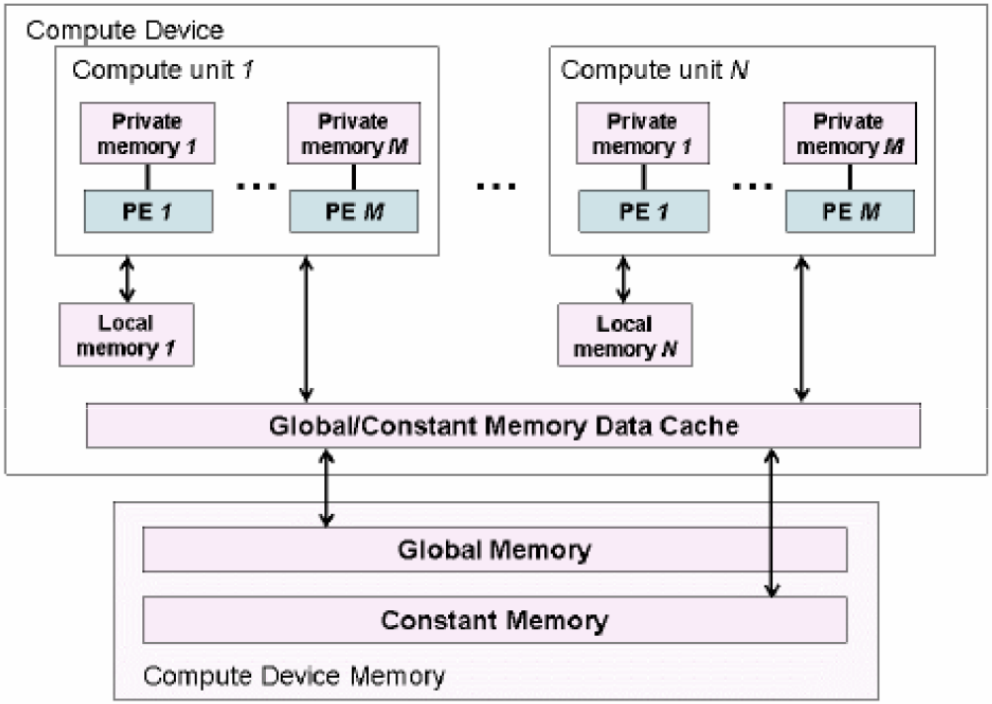
Execute Unit

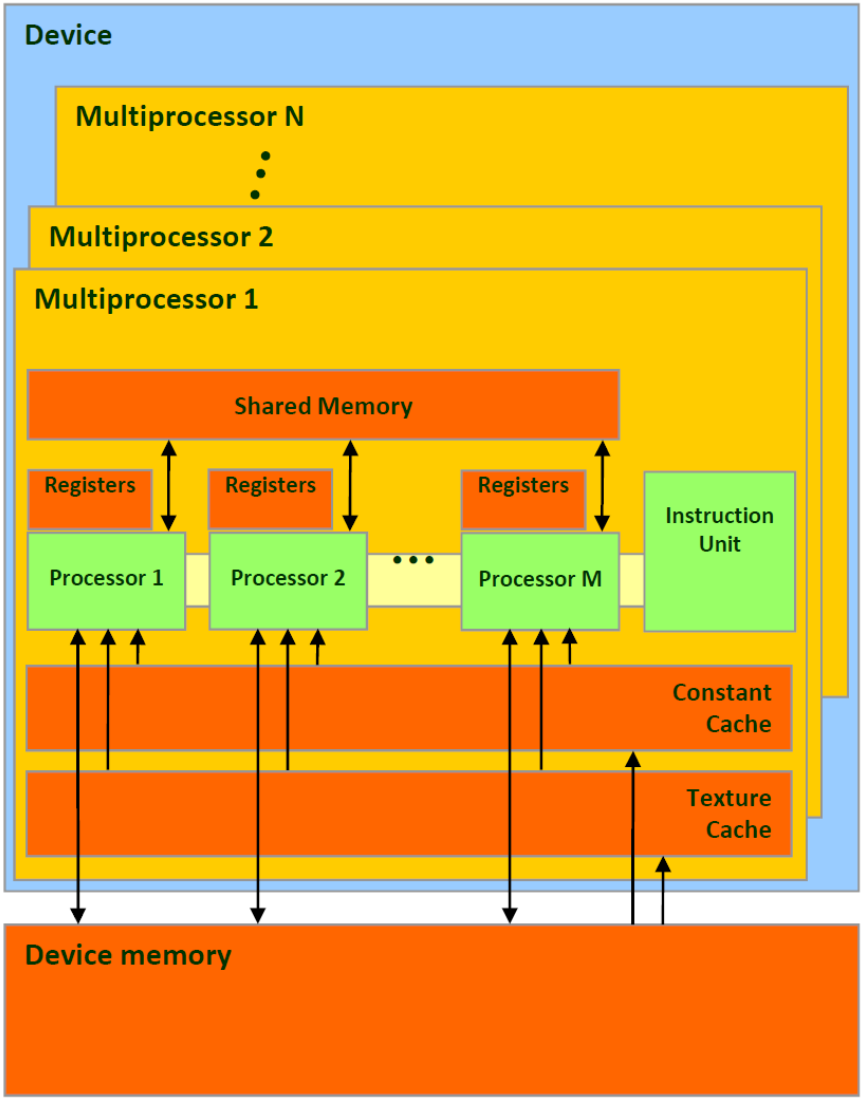
OpenCL Platform Model



OpenCL Terminology	NVIDIA	ATI
Compute Device	Device	Compute Device
Compute Unit	Streaming Multiprocessor	Compute Unit (SIMD Engine)
Processing Element	Streaming Processor (CUDA Processor/Core)	Stream Core (SIMD Unit)
	Scalar Core	Processing Element
	Warp	Wavefront

OpenCL Memory Model





OpenCL Terminology	NVIDIA	ATI
Private	Register Local	Private
Local	Shared	Local
Constant	Constant	Constant
Global	Global	Global

OpenCL Programming

CUDA Programming

```
__global__
void kernel_func(…)
{
}

cudaMalloc
cudaMemcpy
kernel_func<<<…>>>(…)
cudaMemcpy
cudaFree
```

OpenCL Programming

```
__kernel  void kernel_func(…)
{
}

Decide platform
Find and select device

Allocate device memory
Copy data from host to device
Select kernel function
Build (compile) kernel function
Run kernel func
Copy data from device to host
Deallocate device memory
```

OpenCL Programming

```
#include <stdio.h>
#ifdef __APPLE__
#include <OpenCL/opencl.h>
#else
#include <CL/cl.h>
#endif

int main()
{
    // 1. find device
    int status;
    cl_platform_id platform;
    status = clGetPlatformIDs(1, &platform, NULL);
#ifdef DEBUG
    char name[100];
    printf("# of platforms : %d\n", nPlatforms);
    status = clGetPlatformInfo(platform, CL_PLATFORM_VENDOR, 100, name, NULL);
    printf("platform vendor : %s\n", name);
#endif

    cl_device_id devices[3]; // Intel Core i7, HD Graphics 4000, GeForce GT 650M
    status = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 3, devices, NULL);
#ifdef DEBUG
    for(int i=0; i<3; i++) {
        clGetDeviceInfo(devices[i], CL_DEVICE_NAME, 100, name, NULL);
        printf("device name : %s\n", name);
    }
#endif

    return 0;
}
```

```
K0754N1:src dokto76$ ./hello.x
platform vendor : Apple
device name : Intel(R) Core(TM) i7-3720QM CPU @ 2.60GHz
device name : HD Graphics 4000
device name : GeForce GT 650M
```

OpenCL Programming

```
// 0. initialize user data
float f_data[NUM_OF_DATA];
for(int i=0; i<NUM_OF_DATA; i++)
    f_data[i] = i+1.0;

// 1. find device and create context
cl_platform_id platform;
clGetPlatformIDs(1, &platform, NULL);

cl_device_id devices[3]; // Intel Core i7, HD Graphics 4000, GeForce GT 650M
clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 3, devices, NULL);
cl_device_id device = devices[1];
cl_context context = clCreateContext(0, 1, &device, NULL, NULL, NULL);

// 2. create runtime object (command queue)
cl_command_queue queue = clCreateCommandQueue(context, device, 0, NULL);

// 3. read kernel source code and compile it
char *KernelSource = read_kernelsource("kernel.cl");
cl_program program = clCreateProgramWithSource(context, 1,
                                (const char **) &KernelSource, NULL, NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "func", NULL);
```

OpenCL Programming

```
// 4. allocate buffer to device
cl_mem mem_f_data = clCreateBuffer(context, CL_MEM_READ_WRITE,
                                   sizeof(float)*NUM_OF_DATA, NULL, NULL);

// 5. copy data to device
clEnqueueWriteBuffer(queue, mem_f_data, CL_TRUE, 0,
                     sizeof(float)*NUM_OF_DATA, &f_data, 0, NULL, NULL);

// 6. set kernel argument
int num_of_data = NUM_OF_DATA;
clSetKernelArg(kernel, 0, sizeof(cl_mem), &mem_f_data);
clSetKernelArg(kernel, 1, sizeof(int), &num_of_data);

// 7. execute kernel code
size_t global = NUM_OF_DATA;
size_t local = 1;
clEnqueueNDRangeKernel(queue, kernel,1, NULL,&global,&local,0,NULL,NULL);
clFinish(queue);

// 8. read data from GPU to host and print result
clEnqueueReadBuffer(queue, mem_f_data, CL_TRUE, 0,
                     sizeof(float)*NUM_OF_DATA, f_data, 0, NULL, NULL);
```

OpenCL Programming

```
__kernel void func(__global float *f_data, int num_of_data)
{
    int i = get_global_id(0);
    if( i < num_of_data ) {
        f_data[i] *= 2.0;
    }
}
```

```
K0754N1:src dokto76$ xcrun clang -arch x86_64 -isysroot /Applications/Xcode.app/
Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.9.sdk -fram
ework OpenCL -DDEBUG -o source.x source.c
K0754N1:src dokto76$ ./source.x
2.000000
4.000000
6.000000
8.000000
10.000000
12.000000
14.000000
16.000000
18.000000
20.000000
```

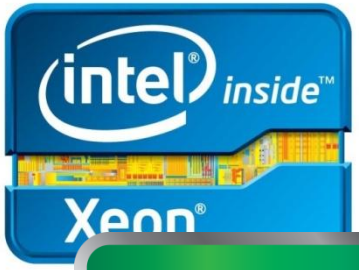
Compile

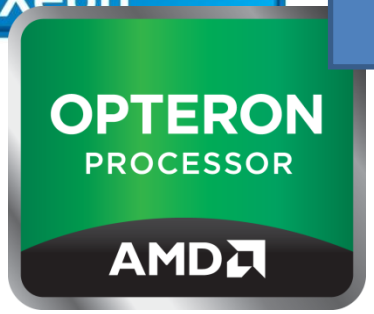
$ gcc -o source.x source.c -lOpenCL

Heterogeneous Computing Resources

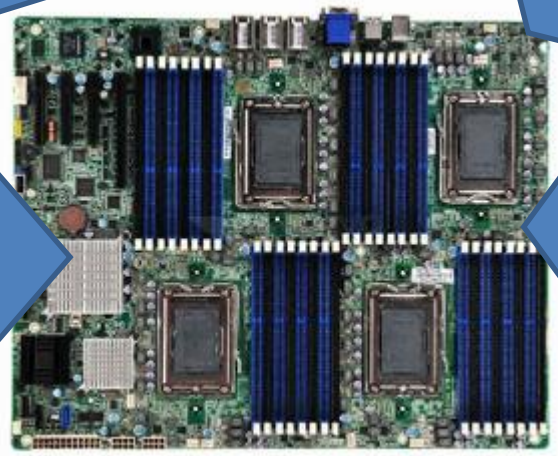
MIC

FPGA





CPU



GPU

Summary

- GPU computing is possible by using
  - Libraries
  - Directive based OpenACC or OpenMP
  - CUDA or OpenCL

- GNU gcc covers OpenMP 4.0 for CPU, MIC and will cover for GPU(ATI) soon

- We can use CPU, MIC, GPU(on chip, card) by using OpenCL

Thank you

Korea Institute of Science and Technology Information

2016