GPU

CPU dan GPU

- Awalnya, resource utama untuk komputasi adalah CPU (central processing unit)
- GPU (graphic processing unit) digunakan sebagai akselerator untuk pemrosesan grafik
- Teknologi berkembang, akhirnya GPU menjadi salah satu resource yang dapat dimanfaatkan untuk komputasi juga
- GPUnya disebut sebagai GP GPU (General Purpose GPU) dan dapat dipergunakan sebagai alat hitung yang lebih umum

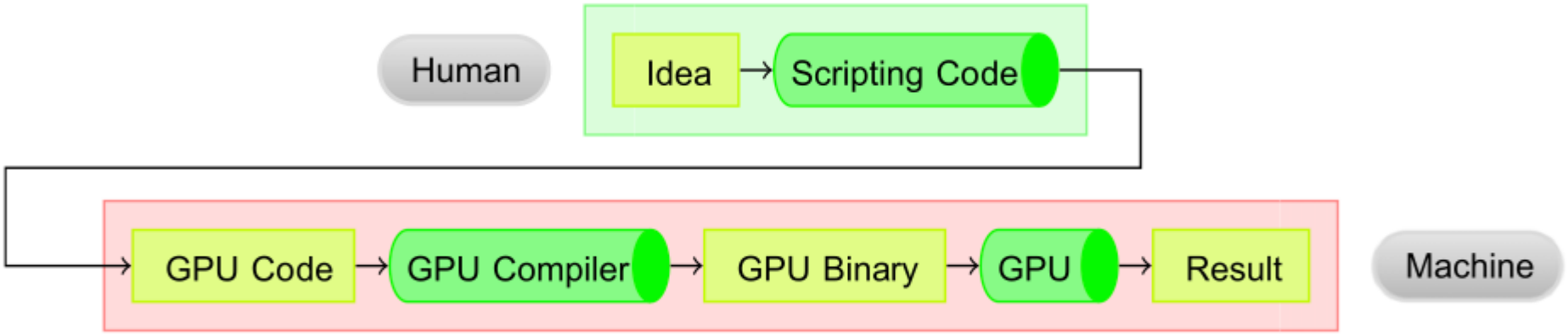
GPU vs CPU

	CPU	GPU
1	Multicore atau manycore (Intel i-9 memiliki 18 core dalam 1 silicon chip, Intel Xeon Phi 7210 memiliki 64 core)	Ribuan core (nVidia 700~4000 core, AMD 1000~2500 core, Intel 110~1100 core)
2	Tiap corenya lebih cepat	Tiap corenya lebih lambat
3	Multi purpose	Specific purpose
4	Higher electricity consumption (on same speed)	Higher hardware cost (on same speed)
5	Bagus untuk pekerjaan yang rumit (kompleks) di domain yang sedikit	Bagus untuk pekerjaan yang sederhana, tapi banyak (di domain data yang luas)

- Note : core CPU dan core GPU berbeda fungsi, jadi tidak bisa dibandingkan langsung dari banyaknya atau kecepatannya

API
(Application Programming Interface)

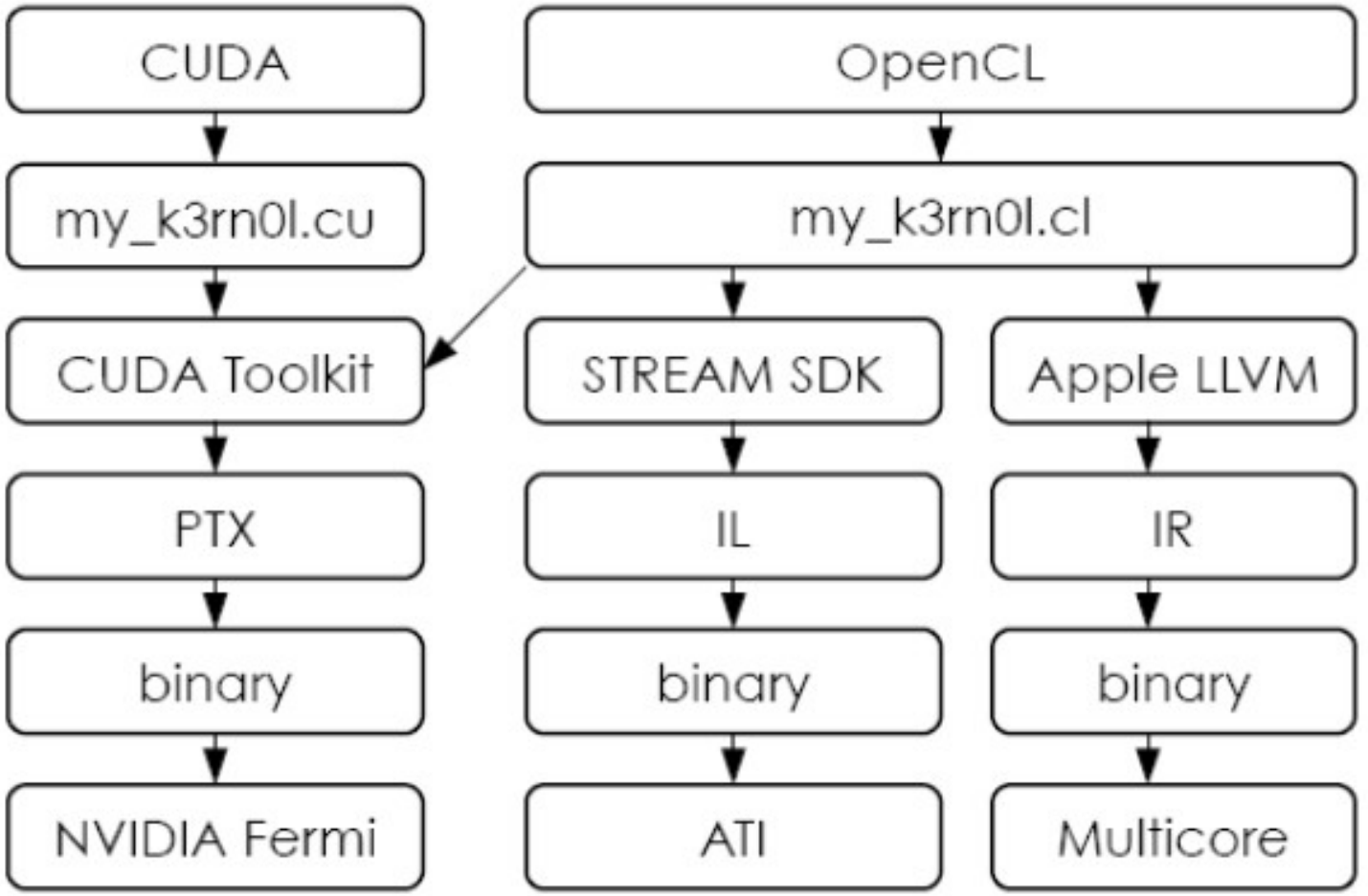
- Untuk bisa memanfaatkan resource GPU sebagai alat komputasi, diperlukan sekumpulan pustaka program (library), salah satunya :
  - CUDA
  - OpenCL



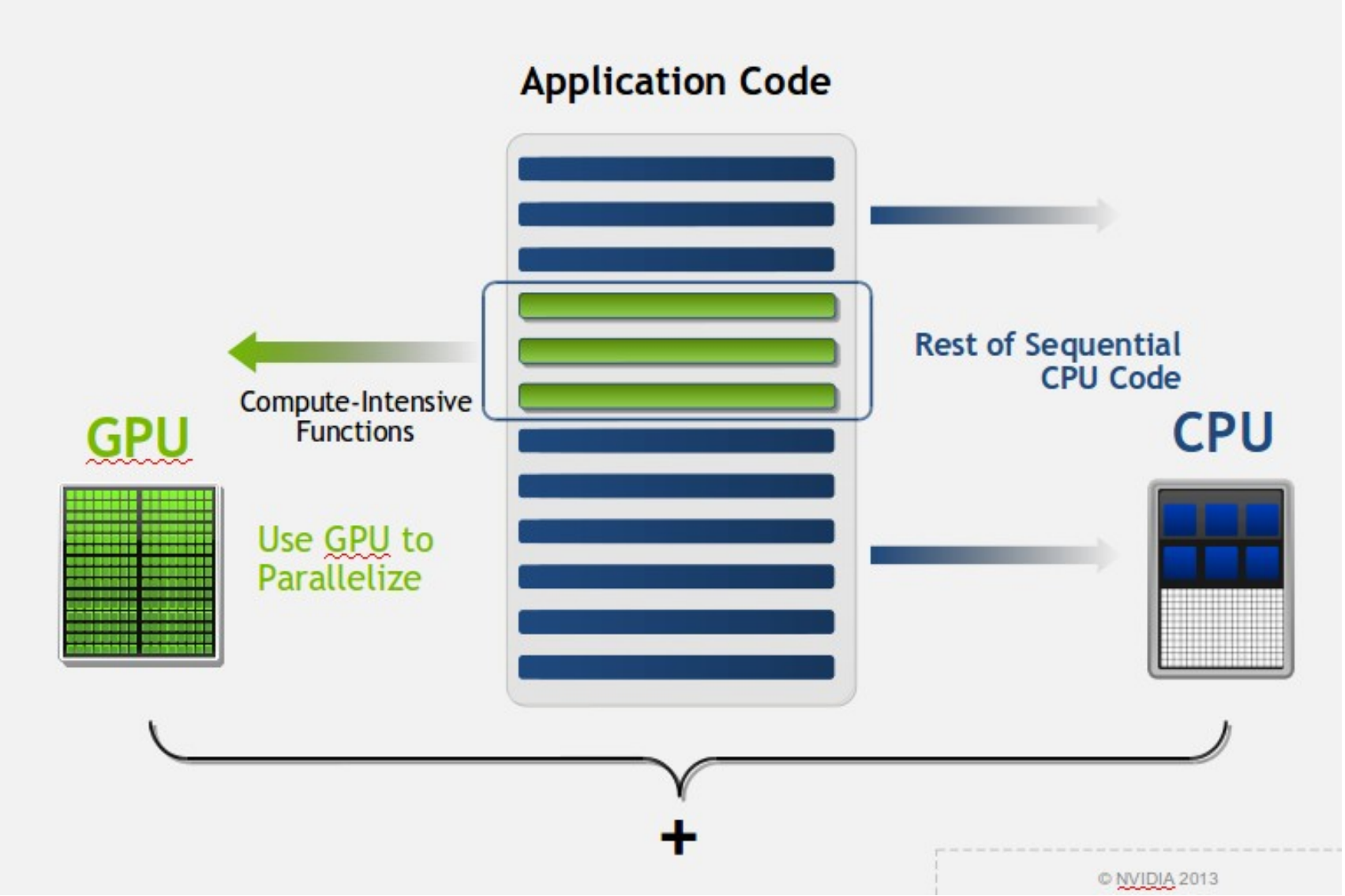
Cuda vs OpenCL

- NVIDIA dan AMD/ATI adalah produsen GPU terbesar saat ini
- OpenCL adalah spesifikasi paralel programming yang terbuka dan bebas royalty
- CUDA adalah proprietary NVIDIA
- NVIDIA punya implementasi OpenCL juga

Cuda vs OpenCL



GPU Sebagai Akselerator



Beberapa Aplikasi Yang Sudah GPU Accelerated

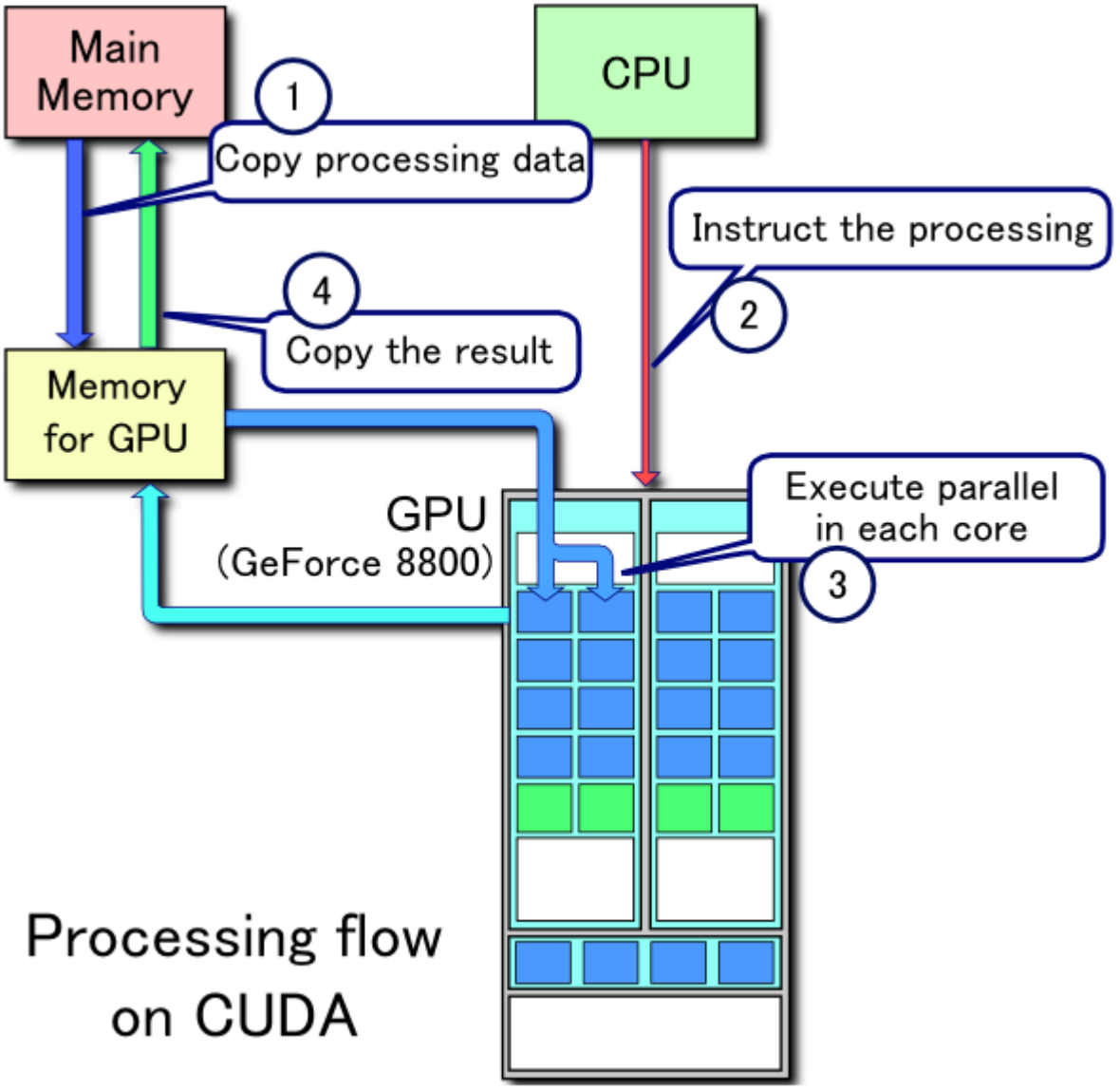
- **Adobe After Effect** (3D Ray Tracing)
- **Adobe Photoshop** (Effects di Mercury Graphics Engine)
- **Adobe Premiere** (Mercury Playback Engine for real-time video editing & accelerated rendering)
- **Adobe Speedgrade** (Real time grading and finishing)
- **Autodesk Maya** (Physics Simulation, Enhance size)
- **Blackmagic DaVinci Resolve** (real time colour correction, denoising)
- **Sony Vegas Pro** (Video effects dan encoding)
- dll

Host, Device, dan Kernel

- Programmer tidak bisa langsung menjalankan kode di GPU
- Kode utama dijalankan pada CPU
  - Host = CPU dan memorinya (*host memory*)
  - Device = GPU dan memorinya (*device memory*)
- Untuk menjalankan kode di GPU, digunakan fungsi yang disebut sebagai *kernel*

CUDA

Alur Proses Komputasi dengan CUDA



- Copy input data from host memory to device memory
- Load GPU code and execute
- Copy result from device memory to host memory

Hello World

```
int main (void) {
  printf("Hello world!\n");
  return 0;
}
```

- Ket : program di atas tanpa device code, artinya program c biasa yang dijalankan di host
- NVIDIA compiler bisa menjalankan program tanpa device code
  - file disimpan sebagai .cu
  - compile dengan nvcc

Hello World dengan Device Code

```
__global__ void mykernel(void) {
}

int main (void) {
  mykernel<<<1,1>>>();
  printf("Hello world!\n");
  return 0;
}
```

__global__

- Keyword **`__global__`** mengindikasikan bahwa program tersebut :
  - Dipanggil oleh *host code*
  - Dijalankan di *device*
- *Device function* diproses oleh *NVIDIA compiler*
- *Host function* diproses oleh *standard host compiler*

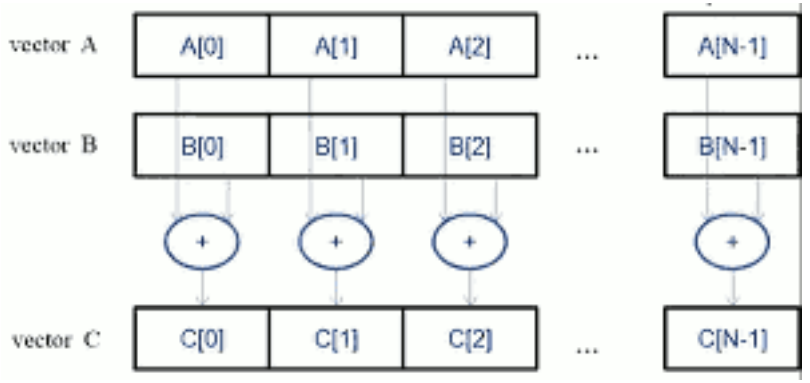
mykernel<<<1,1>>>()

- Disebut sebagai *kernel launch*
- Angka di dalam triple bracket (<<< >>>) menunjukkan konfigurasi yang digunakan untuk menjalankan kernel tersebut secara paralel (*block, thread*)

- Apa yang dilakukan oleh program tersebut setelah dijalankan?

Contoh Lain : Menjumlahkan 2 Vektor

- Input 2 buah vektor A dan B
- Dioperasikan element-per-elemen dengan operasi penjumlahan
- Hasil disimpan dalam vektor C



Menjumlahkan 2 Vektor (1 elemen) - Kernel

```
__global__ void add(int *a, int *b,
int *c) {
   *c = *a + *b;
}
```

- Fungsi (*kernel*) **`add`** dijalankan oleh *device*, jadi **`a,b,c`** harus mengarah ke *device memory*
- Perlu alokasi memori di *device* (GPU)

Menjumlahkan 2 Vektor (1 elemen) – Main Program

```
int main(void) {
  int a,b,c;
  int *da, *db, *dc;
  int size = sizeof(int);
  cudaMalloc((void **) &da, size); cudaMalloc((void **)
&db, size); cudaMalloc((void **) &dc, size);
  a = 2; b = 7;
  cudaMemcpy(da, &a, size, cudaMemcpyHostToDevice);
  cudaMemcpy(db, &b, size, cudaMemcpyHostToDevice);
• add<<<1,1>>>(da,db,dc);
  cudaMemcpy(&c, dc, size, cudaMemcpyDeviceToHost);
  cudaFree(da); cudaFree(db); cudaFree(dc);
  return 0;
}
```

Menjumlahkan 2 Vektor (N elemen)

```
add<<<N,1>>>();
```

- Kita jalankan kernelnya secara paralel N kali
- Tiap pemanggilan **`add()`** disebut sebagai blok (*block*)
- Himpunan dari blok disebut sebagai grid
- Pada contoh berikut, kita menjalankan fungsi **`add()`** dengan **`N`** buah blok per grid
- Setiap pemanggilan dapat merefer ke indeksnya masing masing menggunakan **`blockIdx.x`** sebagai indeks ke arraynya.

Menjumlahkan 2 Vektor (N elemen) - Kernel

```
__global__ void add(int *a, int *b, int *c) {
   *c[blockIdx.x]=*a[blockIdx.x]+*b[blockIdx.x];
}
```

Menjumlahkan 2 Vektor (N elemen) – Main Program

```
#define N 512

int main (void){

    int a,b,c;

    int *da, *db, *dc;

    int size = N*sizeof(int);

    cudaMalloc((void **) &da, size); cudaMalloc((void **) &db,
size); cudaMalloc((void **) &dc, size);

    a = (int *)malloc(size); random_ints(a,N);

    b = (int *)malloc(size); random_ints(b,N);

    c = (int *)malloc(size);

    ...
```

Menjumlahkan 2 Vektor (N elemen) – Main Program

```
    ...
    cudaMemcpy(da, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(db, b, size, cudaMemcpyHostToDevice);
    add<<<N,1>>>(da,db,dc);
    cudaMemcpy(c, dc, size, cudaMemcpyDeviceToHost);
    free(a); free(b); free(c);
    cudaFree(da); cudaFree(db); cudaFree(dc);
    return 0;
}
```

Thread

- Kita bisa menjalankan fungsinya dengan beberapa thread dengan cara mengubah kodenya
- Pada contoh di bawah, kernel dijalankan dengan **N** buah thread per block

```
__global__ void add(int *a, int *b, int *c) {
    *c[threadIdx.x]=*a[threadIdx.x]+*b[threadIdx.x];
}
#define N 512
int main (void){
    ...
    add<<<1,N>>>(da,db,dc);
    ...
}
```

Block dan Thread

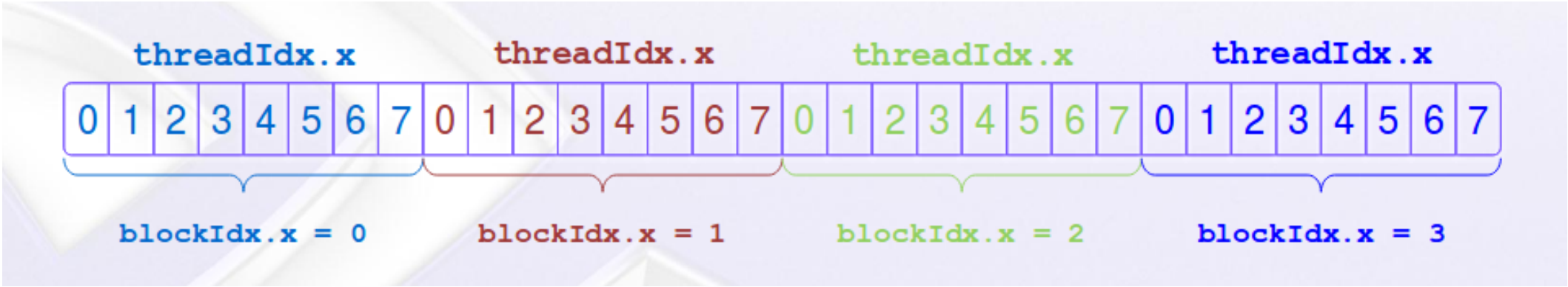
```
mykernel<<<m,n>>>();
```

m = banyak blok yang digunakan

n = banyak thread yang digunakan

dengan **n** thread tiap bloknya, maka indeks yang unik diberikan oleh formula

```
int index = threadIdx.x + blockIdx.x * n
```



blockDim.x

- Ada variabel yang menyimpan ukuran banyaknya thread dalam setiap blok, yaitu **`blockDim.x`**

```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x ;
    *c[index]=*a[index]+*b[index];
}

#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main void {
    ...
    add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(da,db,dc);
    ...
}
```

Jika Ukuran N !| blockDim.x

- Untuk mencegah terjadinya error karena mengakses array yang di luar indeks, perlu dibatasi
- Variabel **n** menyimpan ukuran array yang diproses

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x ;
    if (index < n )
        *c[index]=*a[index]+*b[index];
}
```

Kenapa Dipisahkan Antara Thread dan Block

- Thread memiliki mekanisme untuk secara efisien
  - Berkomunikasi
  - Singkronisasi
- Contoh Kasus : Stensil 1D

Open CL

Yang Berbeda dengan CUDA

- Deklarasi kernel

  `__kernel void mykernel()`
- Tipe variabel

  `__global, __local`
- Alokasi memori

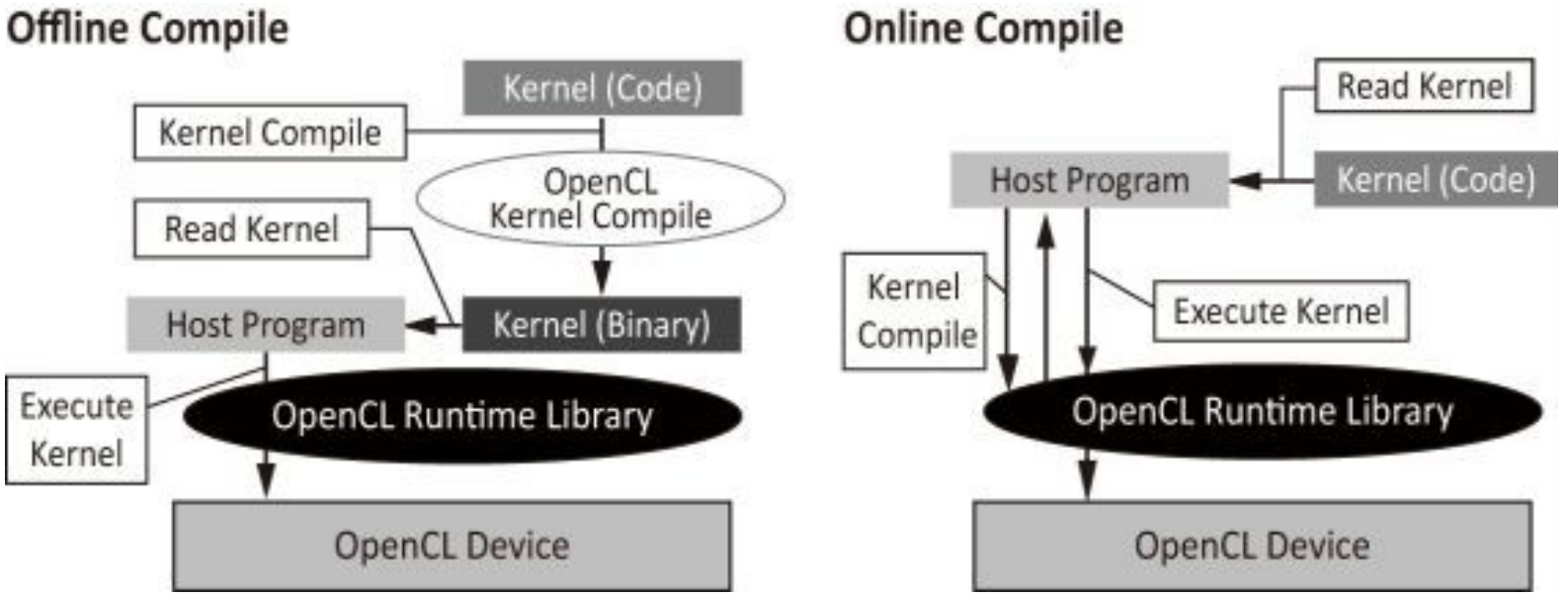
  `cl_mem , clCreateBuffer`
- Singkronisasi

  `__synchthreads()` → `barrier()`
- Istilah block dan thread

  block → **work-group** , **thread** → **work-item**
- Alur

Alur Proses Komputasi dengan OpenCL

- Offline = kernel binary dibaca oleh host code
- Online = kernel source dibaca oleh host code



Menjumlahkan 2 Vektor Kernel

```
__kernel void add(__global double *a,
                  __global double *b,
                  __global double *c,
                  const unsigned int n)
{
  int id = get_global_id(0);
  if (id < n)
        c[id] = a[id] + b[id];
}
```

- CUDA sudah ada lebih dahulu, lebih banyak library yang disediakan
- CUDA lebih mudah dioperasikan (easy to compile)
- OpenCL lebih *versatile*
- OpenCL masih akan terus berkembang dan potensinya lebih besar dari CUDA (karena open)

Referensi