Pertemuan 12 — GPU dan CUDA Dasar

Ribuan *core* kecil: arsitektur GPU, model SIMT, dan *kernel* pertama Anda

Hendri Karisma

Semester Ganjil 2026/2027

Program Studi Teknik Informatika, STMIK Tazkia

Alur pertemuan hari ini

1. Dua filosofi: *latency-oriented* vs *throughput-oriented*
2. Arsitektur GPU: *streaming multiprocessor*, *warp*, SIMT
3. *Warp divergence*
4. Model CUDA: *host*, *device*, *kernel*
5. *Grid*, *block*, *thread*, dan indeksnya
6. Hierarki memori GPU
7. Kode: *vector add* dan *matmul* naif
8. Mengukur waktu dengan `cudaEvent`
9. *Transfer cost* PCIe: kapan GPU tidak layak
10. Praktikum di Google Colab

Sasaran pertemuan ini

Anda dapat menjelaskan mengapa GPU cepat untuk pekerjaan paralel masif, menulis dan menjalankan *kernel* CUDA sederhana yang benar, lalu menghitung *speedup* GPU terhadap CPU secara jujur, **termasuk** *cost* memindahkan data.

Dua filosofi prosesor

Dari P10 dan P11 ke GPU

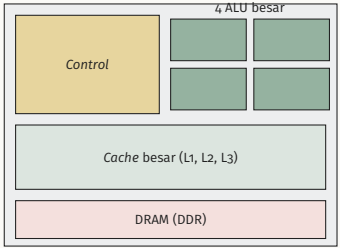
- P10: CPU berusaha membuat **satu** *thread* secepat mungkin dengan *cache* besar, *branch prediction*, dan *out-of-order execution*.
- P11: untuk kerja lebih besar, kita tambah **mesin**, dan membayar *communication cost*.
- Hari ini jalan ketiga: dalam **satu** chip, taruh ribuan unit hitung sederhana yang mengerjakan instruksi yang sama pada data yang berbeda.

Ringkasan

Gagasan ini lahir dari grafika komputer: jutaan piksel, masing-masing dihitung dengan rumus yang sama. Ternyata perkalian matriks, simulasi fisika, dan *deep learning* punya bentuk yang persis sama. Itulah *general-purpose GPU computing*.

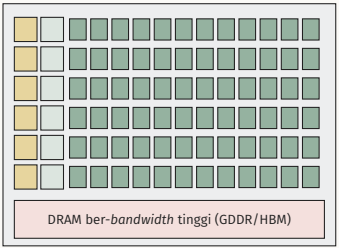
Latency-oriented vs throughput-oriented

CPU



Banyak transistor dipakai untuk **menghindari menunggu**: *cache*, *branch prediction*, *out-of-order execution*. Unggul untuk kode bercabang dan berurutan.

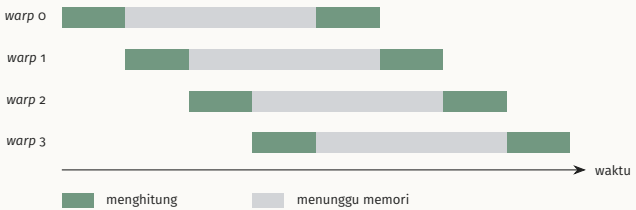
GPU



Hampir semua transistor dipakai untuk **unit hitung**. *Control* dan *cache* kecil, dibagi oleh banyak unit. GPU tidak menghindari menunggu; ia **menyembunyikannya**.

Sumber: diadaptasi dari gambaran klasik dalam buku PMPP (Kirk, Hwu, El Hajj); bukan denah chip sesungguhnya.

Menyembunyikan *latency* dengan berganti *warp*



- Baca dari memori global butuh ratusan siklus.
- Ketika satu *warp* menunggu, *scheduler* di *hardware* langsung menjalankan *warp* lain yang siap. Pergantiannya hampir tanpa *overhead*, karena setiap *warp* sudah punya *register* sendiri.
- Syaratnya: **harus ada cukup banyak** *thread*. GPU dengan sedikit *thread* sama saja dengan CPU yang lambat.

Ringkasan

Karena itu program GPU sengaja meluncurkan *thread* jauh lebih banyak dari jumlah unit hitung: puluhan ribu sampai jutaan.

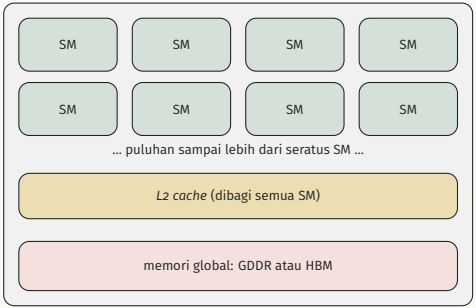
CPU vs GPU dalam angka kasar

	CPU *server*	**GPU pusat data**
Jumlah *core*/unit	Puluhan sampai sekitar 100-an *core*	Ribuan unit hitung sederhana
Thread aktif	Satu atau dua per *core*	Puluhan ribu sekaligus
Memory bandwidth	Ratusan GB/s (DDR)	Ratusan GB/s sampai beberapa TB/s (GDDR, HBM)
Cara menghadapi *latency*	*Cache* besar, *branch prediction*	Berganti *warp*
Unggul untuk	Logika bercabang, sistem operasi, *database*, layanan web	Operasi seragam pada data besar: matriks, citra, simulasi, AI

Heterogeneous computing (P1): CPU mengatur alur program dan bagian yang berurutan; GPU mengerjakan bagian yang paralel masif. Keduanya bekerja bersama.

Arsitektur GPU: SM, *warp*, SIMT

Isi sebuah GPU



Satu ***Streaming Multiprocessor*** (SM) berisi:

- banyak unit hitung FP32/INT ("CUDA *core*"), dan pada GPU modern juga *Tensor Core* untuk perkalian matriks (P13);
- *register file* yang sangat besar, dibagi ke semua *thread* yang aktif;
- *shared memory*/L1 yang dapat diatur pemrogram;
- beberapa *scheduler warp*.

Menambah SM adalah cara GPU *scale-up*: generasi baru, SM lebih banyak.

Warp dan SIMT

- ***Warp*** — kelompok **32 *thread*** yang dijadwalkan bersama pada satu SM.
- **SIMT** (*Single Instruction, Multiple Threads*) — ke-32 *thread* itu menjalankan instruksi yang sama pada waktu yang sama, masing-masing dengan *register* dan datanya sendiri.
- Bedanya dengan SIMD (P10): pemrogram menulis kode untuk **satu** *thread* biasa; *hardware* yang mengelompokkannya.



Ringkasan

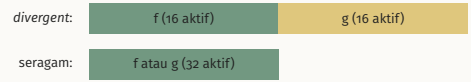
Ukuran *block* sebaiknya kelipatan 32. *Block* berisi 100 *thread* tetap memakai 4 *warp*; *warp* terakhir hanya terisi 4 *thread* dan 28 jalurnya menganggur.

Warp divergence: ketika *thread* berbeda jalan

```
// Buruk: bercabang DI DALAM satu warp
if (threadIdx.x % 2 == 0)
    x[i] = f(x[i]);   // separuh warp jalan
else
    x[i] = g(x[i]);   // separuh lagi menunggu

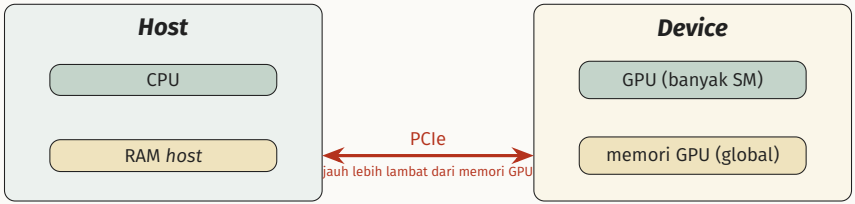
// Lebih baik: bercabang per warp
if ((threadIdx.x / 32) % 2 == 0)
    x[i] = f(x[i]);   // seluruh warp ke f
else
    x[i] = g(x[i]);   // seluruh warp ke g
```

- Bila *thread* dalam satu *warp* mengambil cabang berbeda, cabang-cabang itu dijalankan **bergantian**; *thread* yang tidak ikut ditopeng (*masked*).
- Pada contoh pertama, waktu yang dibutuhkan kira-kira waktu `f` ditambah waktu `g`.
- Percabangan yang seragam di dalam *warp* tidak merugikan.
- Catatan: versi kedua memberi `f` ke *warp* genap, bukan *thread* genap, jadi hasilnya berbeda. Agar setara, susun ulang data dulu sehingga elemen yang butuh `f` berkumpul dalam *warp* yang sama.
- Ini sebabnya GPU kurang cocok untuk kode yang sangat bercabang: *parser*, logika bisnis, penelusuran pohon.



Model pemrograman CUDA

Host dan *device*



- Program CUDA adalah program C/C++ biasa yang berjalan di *host*, ditambah fungsi khusus (*kernel*) yang berjalan di *device*.
- Kedua memori terpisah. Data harus **disalin** ke GPU sebelum dihitung dan disalin kembali sesudahnya. (*Unified memory* menyembunyikan salinan ini, tetapi *cost*-nya tetap ada.)
- Ini sistem terdistribusi kecil di dalam satu komputer: dua memori, satu jalur komunikasi, dan *cost* alfa-beta yang sama seperti P11.

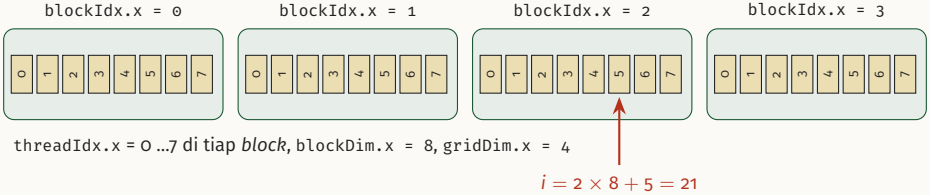
Kernel dan cara meluncurkannya

```
// __global__: dipanggil host, jalan di device
__global__ void tambah(const float *a,
                       const float *b,
                       float *c, int n) {
    int i = blockIdx.x * blockDim.x
          + threadIdx.x;
    if (i < n)           // penjaga batas
        c[i] = a[i] + b[i];
}

// di host:
tambah<<<jumlahBlock, ukuranBlock>>>(
    da, db, dc, n);
```

- `__global__` — *kernel*: dipanggil dari *host*, dijalankan di *device* oleh banyak *thread* sekaligus.
- `__device__` — fungsi pembantu yang hanya dipanggil dari kode GPU.
- `__host__` — fungsi CPU biasa (bawaan bila tidak ditulis).
- `«<grid, block»>` — *launch configuration*: berapa *block*, berapa *thread* per *block*.
- Peluncuran *kernel* bersifat ***asynchronous***: CPU langsung lanjut tanpa menunggu GPU selesai.
- Satu baris kode *kernel* = pekerjaan **satu** *thread*. Loop `for` di versi CPU menghilang.

Grid, block, thread



$$i = \texttt{blockIdx.x} \times \texttt{blockDim.x} + \texttt{threadIdx.x}$$

Contoh: $n = 1000$, *block* 256 *thread*.

$$\text{grid} = \lceil 1000/256 \rceil = 4 \;\Rightarrow\; 1024 \textit{ thread}$$

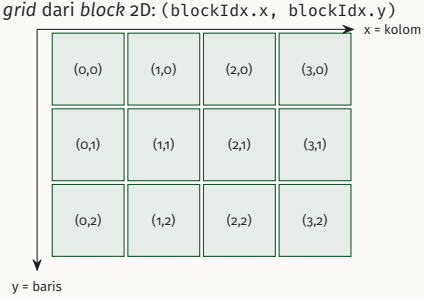
$\lceil \cdot \rceil$: pembulatan ke atas. 24 *thread* terakhir tidak punya data, karena itu wajib ada `if (i < n)`.

- Satu *block* selalu berjalan di **satu** SM, dan *thread* di dalamnya dapat berbagi *shared memory* serta saling menunggu (`__syncthreads`).
- *Block* yang berbeda tidak boleh bergantung satu sama lain; GPU bebas menjalankannya dalam urutan apa pun.
- Itulah yang membuat program yang sama *scale* otomatis dari GPU kecil ke GPU besar: GPU besar hanya menjalankan lebih banyak *block* sekaligus.

Indeks dua dimensi untuk matriks

```
dim3 blok(16, 16);              // 256 thread
dim3 grid((kolomN + 15) / 16,   // sumbu x
          (barisN + 15) / 16);  // sumbu y
kernel<<<grid, blok>>>(...);

// di dalam kernel:
int kolom = blockIdx.x * blockDim.x + threadIdx.x;
int baris = blockIdx.y * blockDim.y + threadIdx.y;
if (baris < barisN && kolom < kolomN) {
    // elemen (baris, kolom) disimpan
    // row-major: baris * kolomN + kolom
}
```



`dim3` punya tiga sumbu (x, y, z). Pilih bentuk *grid* yang cocok dengan bentuk data: 1D untuk vektor, 2D untuk matriks dan citra, 3D untuk volume.

Hierarki memori GPU

Memori	Terlihat oleh	Umur	Kecepatan	Dipakai untuk
Register	Satu *thread*	Selama *thread*	Tercepat	Variabel lokal biasa
Shared memory (`__shared__`)	Semua *thread* satu *block*	Selama *block*	Sangat cepat, di dalam SM	Berbagi data antar *thread*, *tiling* (P13)
Constant (`__constant__`)	Semua *thread*, hanya baca	Selama program	Cepat bila semua membaca alamat sama	Parameter, koefisien filter
Global	Semua *thread* dan *host* (lewat salinan)	Selama program	Paling lambat (ratusan siklus)	*Array input* dan *output*

Ringkasan

Pola *GPU optimization* hampir selalu sama dengan P10: **jangan bolak-balik ke memori yang lambat**. Ambil data dari memori global sekali, simpan di *shared memory* atau *register*, lalu pakai berkali-kali. Pekan depan kita melakukannya dengan *tiling*.

Kode: *vector add* dan *matmul* naif

Lima langkah setiap program CUDA



- Langkah merah (2 dan 4) melewati PCIe. Untuk soal ringan, keduanya bisa jauh lebih lama daripada langkah 3.
- `cudaMemcpy` bersifat *blocking*: ia menunggu *kernel* sebelumnya selesai. Karena itu langkah 4 sekaligus menjadi *synchronization point*.
- Setiap fungsi *runtime* CUDA mengembalikan kode *error*. Kesalahan yang tidak diperiksa membuat program “berhasil” dengan hasil sampah.

Vector add: *kernel* dan *error checking*

```cpp
#include <cstdio>
#include <cstdlib>

// Hentikan program bila panggilan CUDA gagal, lengkap dengan lokasinya.
#define CEK(x) do { cudaError_t e = (x); if (e != cudaSuccess) {              \
        fprintf(stderr, "CUDA error: %s di %s:%d\n", cudaGetErrorString(e),  \
                __FILE__, __LINE__);                                          \
        exit(1); } } while (0)

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

- Peluncuran *kernel* tidak mengembalikan nilai. Galatnya diambil dengan `cudaGetLastError()` (*configuration* salah) dan `cudaDeviceSynchronize()` (*error* saat berjalan, misalnya akses di luar batas).
- *Error* yang sering muncul di Colab: *"no kernel image is available"*, tanda kode dikompilasi untuk arsitektur GPU yang berbeda. Lihat *slide* praktikum.

Vector add: kode *host*

```
int main(void) {
    int n = 1 << 24;                          // 16 juta elemen
    size_t bytes = n * sizeof(float);
    float *a = (float *)malloc(bytes), *b = (float *)malloc(bytes),
          *c = (float *)malloc(bytes);
    for (int i = 0; i < n; i++) { a[i] = 1.0f; b[i] = 2.0f; }

    float *da, *db, *dc;                                  // 1
    CEK(cudaMalloc(&da, bytes)); CEK(cudaMalloc(&db, bytes));
    CEK(cudaMalloc(&dc, bytes));
    CEK(cudaMemcpy(da, a, bytes, cudaMemcpyHostToDevice));  // 2
    CEK(cudaMemcpy(db, b, bytes, cudaMemcpyHostToDevice));

    int blok = 256, grid = (n + blok - 1) / blok;          // 3
    tambah<<<grid, blok>>>(da, db, dc, n);
    CEK(cudaGetLastError());
    CEK(cudaDeviceSynchronize());

    CEK(cudaMemcpy(c, dc, bytes, cudaMemcpyDeviceToHost));  // 4
    printf("c[0] = %.1f, c[n-1] = %.1f\n", c[0], c[n - 1]); // 3.0

    cudaFree(da); cudaFree(db); cudaFree(dc);              // 5
    free(a); free(b); free(c);
    return 0;
}
```

- Nomor di komentar mengikuti lima langkah tadi.
- `(n + blok - 1) / blok` adalah pembagian yang dibulatkan ke atas.
- Selalu periksa hasil: bila `c[0]` bukan 3.0, ada yang salah walaupun program tidak *crash*.
- Dengan $n = 2^{24}$, tiga *array* berukuran total 192 MB; 128 MB disalin ke GPU dan 64 MB disalin kembali.

Perkalian matriks naif

```
// C = A x B, semua matriks n x n, row-major
__global__ void matmul(const float *A,
                       const float *B,
                       float *C, int n) {
    int baris = blockIdx.y * blockDim.y + threadIdx.y;
    int kolom = blockIdx.x * blockDim.x + threadIdx.x;
    if (baris < n && kolom < n) {
        float jumlah = 0.0f;
        for (int k = 0; k < n; k++)
            jumlah += A[baris * n + k] * B[k * n + kolom];
        C[baris * n + kolom] = jumlah;
    }
}

// peluncuran
dim3 blok(16, 16);
dim3 grid((n + 15) / 16, (n + 15) / 16);
matmul<<<grid, blok>>>(dA, dB, dC, n);
```

- Satu *thread* menghitung satu elemen C. Untuk $n = 1024$ ada sekitar satu juta *thread*.
- Total kerja $2n^3$ FLOP, sedangkan data hanya $3n^2$ elemen. Inilah sebabnya *matmul* cocok untuk GPU: komputasi tumbuh jauh lebih cepat daripada data yang harus disalin.
- Tetapi versi ini boros: setiap langkah k meminta 2 *float* (8 B) untuk 2 FLOP, *arithmetic intensity* hanya 0,25 FLOP/B. Bila semuanya dari DRAM, *roofline* (P10) menguncinya pada *bandwidth*. Dalam praktik, *cache* L1/L2 dan bacaan A yang dibagi dalam *warp* membuatnya jauh lebih cepat dari taksiran itu.
- P13: *tiling* dengan *shared memory* memakai ulang data secara terkendali dan menaikkan intensitas itu lebih jauh.

Mengukur waktu dan *transfer cost*

Mengukur waktu dengan cudaEvent

```
cudaEvent_t mulai, selesai;
cudaEventCreate(&mulai);
cudaEventCreate(&selesai);

tambah<<<grid, blok>>>(da, db, dc, n);  // pemanasan
cudaEventRecord(mulai);
tambah<<<grid, blok>>>(da, db, dc, n);
cudaEventRecord(selesai);
cudaEventSynchronize(selesai);

float ms = 0.0f;
cudaEventElapsedTime(&ms, mulai, selesai);
printf("kernel: %.3f ms, bandwidth efektif: %.1f GB/s\n",
       ms, 3.0 * bytes / ms / 1e6);
```

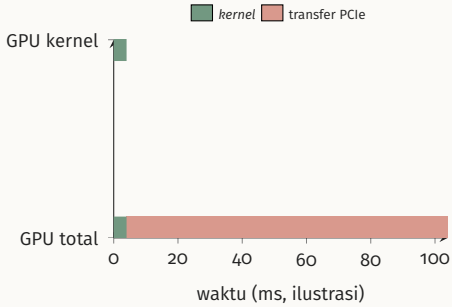
- Karena peluncuran *kernel asynchronous*, mengukur dengan jam CPU di sekitar peluncuran akan mencatat hampir nol. *Event* dicatat di *stream* perintah GPU itu sendiri.
- Peluncuran pertama memuat konteks dan kode; jangan diukur. Lakukan pemanasan, lalu ulangi beberapa kali dan ambil median (P9).
- *Vector add* membaca 2 dan menulis 1 *array*: *bandwidth* efektif $= 3 \cdot \text{bytes}/t$. Bandingkan dengan *bandwidth* memori GPU di spesifikasinya.
- Ukur juga `cudaMemcpy` dengan cara yang sama, secara terpisah.

Cost tersembunyi: transfer lewat PCIe

Hitungan kasar *vector add* dengan $n = 10^8$ *float* pada GPU T4 (GPU yang biasa didapat di Colab):

- Data: 3 *array* $\times$ 400 MB = 1,2 GB.
- *Kernel*: dibatasi *bandwidth* memori GPU ($\approx$ 320 GB/s) $\Rightarrow$ $1{,}2/320 \approx$ **4 ms**.
- Transfer: PCIe 3.0 x16, praktiknya sekitar 10–12 GB/s $\Rightarrow$ $1{,}2/12 \approx$ **100 ms**.
- CPU satu *thread* juga dibatasi *bandwidth* DRAM, sekitar puluhan sampai seratusan ms.

Angka kira-kira; ukur sendiri di praktikum.



Ringkasan

Speedup kernel terlihat 20$\times$ lebih; *speedup* total bisa **kurang dari 1**. Laporkan keduanya.

Kapan GPU tidak layak

GPU cenderung kalah bila

- Data kecil: *launch overhead* dan transfer lebih besar dari kerjanya
- Data dipakai sekali lalu dibuang (*arithmetic intensity* rendah, seperti *vector add*)
- Kode sangat bercabang (*warp divergence*)
- Langkah saling bergantung berurutan, sulit diparalelkan
- Data harus bolak-balik CPU–GPU di setiap langkah

GPU cenderung menang bila

- Komputasi jauh lebih besar dari data (*matmul*: n^3 vs n^2)
- Operasi seragam pada jutaan elemen
- Data **tinggal** di GPU sepanjang banyak langkah (*training* model AI, simulasi berulang)
- Ada *library* yang sudah dioptimalkan (cuBLAS, cuDNN, cuFFT)

Ringkasan

Yang perlu diukur adalah apakah **seluruh alur kerja**, termasuk memindahkan data, lebih cepat dan lebih murah di GPU. Itu pertanyaan yang sama dengan *microservices*: komunikasi jarang gratis.

Praktikum di Google Colab

Praktikum: menjalankan CUDA di Colab

1. Buka *notebook* baru, pilih **Runtime** → **Change runtime type** → **T4 GPU**.
2. Periksa GPU dan *compiler*:

```
!nvidia-smi
!nvcc --version
```

3. Tulis *file* dari sel *notebook*, kompilasi, jalankan:

```
%%writefile vadd.cu
// ... salin kode vector add di sini ...
```

```
!nvcc -O2 -arch=sm_75 vadd.cu -o vadd && ./vadd
```

T4 adalah GPU arsitektur Turing dengan *compute capability* 7.5, karena itu `-arch=sm_75`. Bila Colab memberi GPU lain, sesuaikan dengan *output* `nvidia-smi`. `%%writefile` harus menjadi baris pertama selnya. Bila kuota GPU Colab habis, Kaggle Notebooks menyediakan GPU dengan cara serupa.

Praktikum: yang harus diukur dan dilaporkan

A. *Vector add* ($n = 2^{20}, 2^{22}, 2^{24}, 2^{26}$)

n	CPU (ms)	GPU *kernel* (ms)	GPU total (ms)	GB/s efektif
2^{20}	…	…	…	…
2^{24}	…	…	…	…

B. *Matmul* naif ($n = 256, 512, 1024, 2048$)

n	CPU (ms)	GPU *kernel* (ms)	GPU total (ms)	GFLOPS
512	…	…	…	…
1024	…	…	…	…

- "GPU total" = `cudaMemcpy` masuk + *kernel* + `cudaMemcpy` keluar.
- GFLOPS *matmul* $= 2n^3/t$.
- Versi CPU: *loop* C biasa (boleh dengan OpenMP dari P10) dan hasilnya dipakai untuk memeriksa kebenaran versi GPU (selisih kecil karena urutan penjumlahan *float* wajar).
- Pertanyaan: pada n berapa *speedup* total *vector add* melewati 1? Dan *matmul*?

Untuk belajar lebih jauh

- **Buku PMPP** — David Kirk, Wen-mei Hwu, Izzat El Hajj, *Programming Massively Parallel Processors*, edisi ke-4. Bab 1–5 untuk hari ini; Bab 5–6 dan pola paralel untuk P13.
- **Kuliah Wen-mei Hwu** (University of Illinois Urbana-Champaign) — *Heterogeneous Parallel Programming* pernah tersedia di Coursera; materi sejenis dan kuliah berbasis PMPP kini dapat ditemukan terbuka, termasuk rekaman di YouTube.
- ***CUDA C++ Programming Guide*** — dokumentasi resmi NVIDIA, rujukan untuk setiap fungsi *runtime* yang dipakai hari ini.
- **CUDA Samples** — contoh resmi di GitHub NVIDIA, termasuk `vectorAdd` dan `matrixMul`.

Ringkasan

Konsep hari ini (*grid*, *block*, *warp*, hierarki memori) berlaku juga di luar CUDA: HIP pada GPU AMD, SYCL, dan Triton memakai gagasan yang sama dengan nama sedikit berbeda.

Penutup

Ringkasan

- CPU menghindari *latency* dengan *cache* dan *branch prediction*; GPU menyembunyikannya dengan ribuan *thread* dan pergantian *warp*.
- GPU tersusun dari banyak SM; tiap SM menjalankan *warp* berisi 32 *thread* secara SIMT. Percabangan di dalam *warp* membuat jalur dijalankan bergantian.
- Program CUDA: *host* memanggil *kernel* di *device* dengan `«<grid, block»>`; tiap *thread* menghitung indeksnya sendiri.
- Memori: *register*, *shared*, *constant*, global, dari tercepat sampai terlambat.
- Lima langkah: `cudaMalloc`, salin masuk, *kernel*, salin keluar, `cudaFree`; periksa setiap *error*.
- Ukur dengan `cudaEvent`, dan selalu laporkan waktu *kernel* dan waktu total termasuk PCIe.

Tugas untuk pertemuan berikutnya

1. Selesaikan dua tabel praktikum (*vector add* dan *matmul* naif) di Colab. Sertakan grafik waktu terhadap n untuk CPU, GPU *kernel*, dan GPU total.
2. Ubah ukuran *block matmul* (8×8, 16×16, 32×32) dan catat pengaruhnya. Mengapa 32×32 masih boleh, tetapi 64×64 gagal? (Petunjuk: batas *thread* per *block*.)
3. Tulis satu paragraf: apakah komputasi tugas besar UAS kelompok Anda cocok untuk GPU? Hitung kasar rasio komputasi terhadap data yang harus disalin.
4. Baca PMPP Bab 5 (*memory architecture and data locality*).

Pertemuan depan

Membuat *matmul* naif jauh lebih cepat dengan *tiling* dan *shared memory*, membacanya pada model *roofline*, lalu mengenal akselerator lain: *Tensor Core*, TPU, dan FPGA, serta HBM, NVLink, dan CXL.

Terima kasih

Pertanyaan?