

Pertemuan 13 – Optimasi GPU dan *Heterogeneous Computing*

Dari *kernel* yang benar ke *kernel* yang cepat, lalu ke ragam akselerator

Hendri Karisma

Semester Ganjil 2026/2027

Program Studi Teknik Informatika, STMIK Tazkia

Alur pertemuan hari ini

1. Mengapa *kernel* naif lambat: *memory-bound*
2. *Memory coalescing*
3. *Tiling* dengan *shared memory*
4. *Occupancy* dan *warp divergence*
5. Pola paralel: *reduction*, *scan*, *histogram*, *stencil*
6. Pustaka: cuBLAS dan cuDNN
7. *Roofline* untuk GPU
8. *Tensor Core* dan presisi rendah
9. TPU, *systolic array*, dan FPGA
10. HBM, PCIe, NVLink, CXL, dan performa per watt

Sasaran pertemuan ini

Anda dapat menerapkan *coalescing* dan *tiling* pada *kernel* perkalian matriks, membaca posisinya pada model *roofline*, serta menjelaskan kapan GPU, TPU, atau FPGA menjadi pilihan yang tepat.

Mengapa *kernel* naif lambat

Mengingat pertemuan 12: perkalian matriks naif

Setiap *thread* menghitung satu elemen C_{ij} :

$$C_{ij} = \sum_{k=0}^{N-1} A_{ik} B_{kj}$$

Di setiap langkah k , *thread* itu:

- ▶ membaca A_{ik} dan B_{kj} dari *global memory*:
 $2 \times 4 = 8$ byte
- ▶ mengerjakan satu perkalian dan satu penjumlahan: 2 FLOP

$$\text{arithmetic intensity} = \frac{2 \text{ FLOP}}{8 \text{ byte}} = 0,25 \text{ FLOP/byte}$$

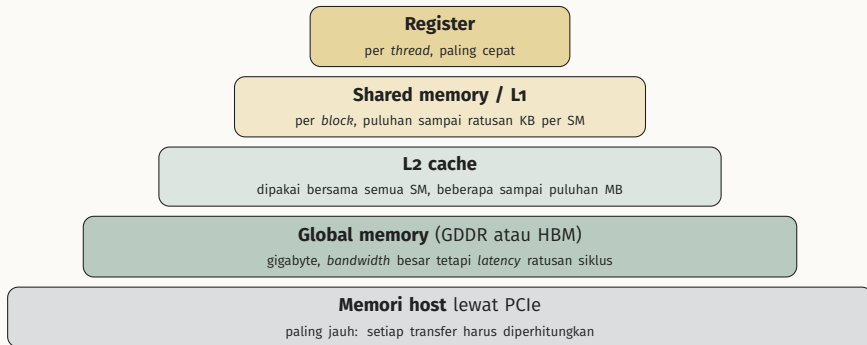
Akibatnya

Pada GPU dengan *bandwidth* 300 GB/s, batas atasnya sekitar $0,25 \times 300 = 75$ GFLOPS, padahal puncak komputasinya ribuan GFLOPS.

Core GPU menganggur menunggu data.
Kernel ini **memory-bound**.

Semua optimasi hari ini punya satu tujuan:
setiap byte yang diambil dari memori harus dipakai sebanyak mungkin.

Hierarki memori GPU: makin dekat, makin cepat, makin kecil



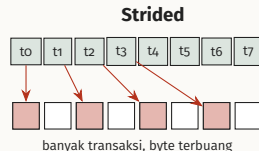
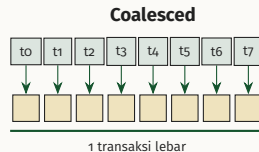
Shared memory adalah “tempat kerja bersama” satu *block* yang diatur sendiri oleh pemrogram. Di sinilah kunci *tiling*.

Memory coalescing

Coalescing: satu warp, satu transaksi

32 *thread* dalam satu *warp* mengeksekusi instruksi baca yang sama bersamaan. Bila alamat yang mereka baca **berurutan**, perangkat keras menggabungkannya menjadi sedikit transaksi memori yang lebar.

- **Coalesced**: *thread* t membaca $a[\text{base} + t]$
- **Strided**: *thread* t membaca $a[t * N]$ — tiap *thread* memicu transaksi sendiri, sebagian besar byte yang diambil terbuang



Coalescing pada perkalian matriks

```
// Indeks yang benar: threadIdx.x -> kolom
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
// B[k*N + col]: thread bertetangga membaca
// alamat bertetangga -> coalesced

// Indeks tertukar: threadIdx.x -> baris
int row = blockIdx.x * blockDim.x + threadIdx.x;
int col = blockIdx.y * blockDim.y + threadIdx.y;
// A[row*N + k]: thread bertetangga melompat
// sejauh N elemen -> strided
```

Matriks C disimpan *row-major*: elemen satu baris berdampingan di memori.

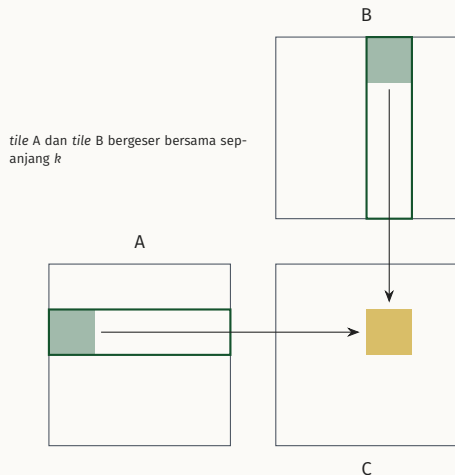
- ▶ Petakan `threadIdx.x` (yang berurutan dalam satu *warp*) ke indeks yang berubah paling cepat di memori: **kolom**.
- ▶ Satu tukar indeks seperti ini bisa mengubah waktu *kernel* beberapa kali lipat, tanpa mengubah jumlah FLOP sama sekali.

Intinya

Pada CPU kita memikirkan *cache line*. Pada GPU kita memikirkan hal yang sama, tetapi untuk 32 *thread* sekaligus.

Tiling dengan shared memory

Gagasan *tiling*: ambil sekali, pakai berkali-kali



Satu *block* $T \times T$ *thread* menghitung satu *tile* C.

1. Semua *thread* bersama-sama memuat satu *tile* A dan satu *tile* B ke *shared memory*.
2. `__syncthreads()`: tunggu sampai *tile* lengkap.
3. Tiap *thread* memakai T nilai dari *tile* itu untuk hasilnya.
4. `__syncthreads()` lagi, lalu geser ke *tile* berikutnya.

Setiap nilai dari *global memory* kini dipakai T kali, bukan sekali.

Kernel perkalian matriks dengan tiling

```
#define TILE 16
__global__ void matmulTiled(const float *A, const float *B,
                           float *C, int N) {
    __shared__ float As[TILE][TILE];
    __shared__ float Bs[TILE][TILE];
    int row = blockIdx.y * TILE + threadIdx.y;
    int col = blockIdx.x * TILE + threadIdx.x;
    float acc = 0.0f;
    for (int t = 0; t < (N + TILE - 1) / TILE; ++t) {
        int aCol = t * TILE + threadIdx.x;
        int bRow = t * TILE + threadIdx.y;
        As[threadIdx.y][threadIdx.x] =
            (row < N && aCol < N) ? A[row * N + aCol] : 0.0f;
        Bs[threadIdx.y][threadIdx.x] =
            (bRow < N && col < N) ? B[bRow * N + col] : 0.0f;
        __syncthreads();           // tile lengkap
        for (int k = 0; k < TILE; ++k)
            acc += As[threadIdx.y][k] * Bs[k][threadIdx.x];
        __syncthreads();           // tile lama selesai dipakai
    }
    if (row < N && col < N) C[row * N + col] = acc;
}
```

Peluncuran:

```
dim3 blk(TILE, TILE);
dim3 grd((N+TILE-1)/TILE,
         (N+TILE-1)/TILE);
matmulTiled<<<grd, blk>>>(
    dA, dB, dC, N);
```

- ▶ Pemuatan *tile* juga *coalesced*:
threadIdx.x bergerak sepanjang baris memori.
- ▶ Kedua __syncthreads() wajib.
Hilangkan satu saja, hasilnya kadang benar kadang salah: *race condition* klasik.
- ▶ Nilai 0 di tepi membuat N yang bukan kelipatan TILE tetap benar.

Berapa yang kita hemat?

Per langkah *tile*, satu *thread* memuat 2 float (8 byte) dari *global memory* lalu mengerjakan T perkalian-tambah ($2T$ FLOP):

$$intensity = \frac{2T}{8} = \frac{T}{4} \text{ FLOP/byte}$$

Versi	FLOP/byte
Naif	0,25
Tiled, $T = 16$	4
Tiled, $T = 32$	8

Mengapa tidak T sebesar mungkin?

- ▶ Satu *block* dibatasi 1024 *thread*, jadi $T \leq 32$ untuk *tile* persegi satu elemen per *thread*.
- ▶ *Shared memory* per SM terbatas; *tile* besar mengurangi jumlah *block* yang bisa aktif bersamaan.
- ▶ Pustaka seperti cuBLAS memakai trik lanjutan: tiap *thread* menghitung beberapa elemen dengan register (*register tiling*).

Occupancy dan divergence

Occupancy: cukup banyak warp untuk menyembunyikan latency

Ketika satu warp menunggu memori, scheduler SM langsung menjalankan warp lain. GPU tidak menghindari *latency*; ia **menyembunyikannya** dengan banyak pekerjaan siap jalan.

$$\text{occupancy} = \frac{\text{warp aktif per SM}}{\text{warp maksimum per SM}}$$

Tiga sumber daya yang membatasinya:

- ▶ **Register** per thread
- ▶ **Shared memory** per block
- ▶ **Ukuran block** dan batas block per SM

Cara memeriksa

- ▶ `nvcc -Xptxas -v` menampilkan register dan *shared memory* yang dipakai kernel
- ▶ `cudaOccupancyMaxPotentialBlockSize` menyarankan ukuran *block*
- ▶ Nsight Compute (ncu) melaporkan *achieved occupancy*

Jangan dikejar membabi buta

Occupancy 100% bukan tujuan. Kernel yang memakai banyak register untuk menyimpan data bisa lebih cepat walau *occupancy*-nya lebih rendah. Ukur, jangan menebak.

Warp divergence

Satu *warp* menjalankan satu instruksi untuk 32 *thread*. Bila *thread* dalam satu *warp* mengambil cabang *if* yang berbeda, kedua cabang dijalankan **bergantian**, dan *thread* yang tidak ikut menganggur.

```
// Divergen: genap dan ganjil dalam satu warp
if (threadIdx.x % 2 == 0) kerjaA();
else                          kerjaB();

// Tidak divergen: cabang ditentukan per warp
if ((threadIdx.x / 32) % 2 == 0) kerjaA();
else                             kerjaB();
```

Satu warp, waktu →

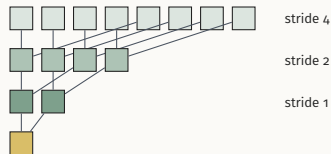


- Biaya terbesar muncul pada cabang yang panjang di dalam *loop*.
- Strategi: susun data atau indeks agar *thread* dalam satu *warp* cenderung mengambil cabang yang sama.

Pola paralel yang sering muncul

Reduction: menjumlahkan sejuta angka

```
// blockDim.x = 256 (pangkat dua)
__global__ void reduceSum(const float *in,
                          float *out, int n) {
    __shared__ float s[256];
    int tid = threadIdx.x;
    int i = blockIdx.x * blockDim.x + tid;
    s[tid] = (i < n) ? in[i] : 0.0f;
    __syncthreads();
    for (int stride = blockDim.x / 2; stride > 0;
         stride >>= 1) {
        if (tid < stride) s[tid] += s[tid + stride];
        __syncthreads();
    }
    if (tid == 0) atomicAdd(out, s[0]);
}
```



- ▶ n angka selesai dalam $\log_2 n$ langkah, bukan n langkah.
- ▶ Thread aktif selalu berurutan dari 0, sehingga warp yang sudah tidak terpakai berhenti seluruhnya (sedikit *divergence*).
- ▶ Hasil tiap *block* digabung dengan `atomicAdd`.

Scan (prefix sum) dan stencil

Scan

Masukan [3, 1, 7, 0, 4] $\rightarrow$ *inclusive scan* [3, 4, 11, 11, 15].

Tampak serial (tiap hasil bergantung pada sebelumnya), tetapi bisa diparalelkan:

- ▶ **Kogge-Stone / Hillis-Steele:** $\log_2 n$ langkah, tiap langkah menjumlahkan tetangga sejauh 1, 2, 4, . . . ; total kerja $O(n \log n)$
- ▶ **Brent-Kung / Blelloch:** dua fase (naik dan turun pohon), total kerja $O(n)$

Dipakai untuk *stream compaction*, *radix sort*, alokasi keluaran paralel.

Stencil

Nilai baru tiap titik dihitung dari titik itu dan tetangganya: *blur* citra, simulasi panas, persamaan diferensial.

$$y_i = w_{-1}x_{i-1} + w_0x_i + w_1x_{i+1}$$

- ▶ Tiap x_i dibaca oleh beberapa *thread* tetangga
- ▶ Solusinya sama dengan matriks: muat satu *tile* plus *halo* (tepi tambahan) ke *shared memory*, lalu hitung dari sana

Intinya

Hampir semua program GPU tersusun dari segelintir pola: *map*, *reduction*, *scan*, *histogram*, *stencil*, perkalian matriks. Kenali polanya, optimasinya ikut terbawa.

Histogram: ketika banyak *thread* menulis ke tempat yang sama

```
__global__ void histo(const unsigned char *data,
                     int n, unsigned int *bins) {
    __shared__ unsigned int lokal[256];
    for (int b = threadIdx.x; b < 256; b += blockDim.x)
        lokal[b] = 0;
    __syncthreads();
    // grid-stride loop: berapa pun ukuran data
    for (int i = blockIdx.x * blockDim.x + threadIdx.x;
         i < n; i += blockDim.x * gridDim.x)
        atomicAdd(&lokal[data[i]], 1u);
    __syncthreads();
    for (int b = threadIdx.x; b < 256; b += blockDim.x)
        atomicAdd(&bins[b], lokal[b]);
}
```

- ▶ `bins[x]++` biasa dari banyak *thread* = *race condition*: hitungan hilang.
- ▶ `atomicAdd` menjamin benar, tetapi *thread* yang menulis ke *bin* sama harus antre.
- ▶ **Privatization**: tiap *block* punya *histogram* sendiri di *shared memory*, baru digabung di akhir. Antrean pindah ke memori yang jauh lebih cepat.

Intinya

Ini masalah yang sama dengan *lock* di pertemuan 6: berbagi keadaan itu mahal. Solusinya juga sama: kurangi berbagi.

**Pustaka: jangan menulis ulang
yang sudah dioptimasi**

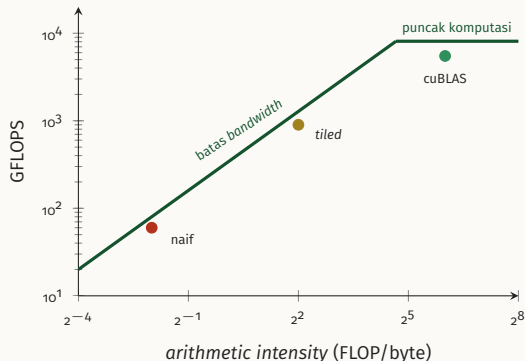
```
#include <cublas_v2.h>
// C = A * B, matriks N x N row-major di device
cublasHandle_t h;
cublasCreate(&h);
const float alpha = 1.0f, beta = 0.0f;
// cuBLAS column-major: hitung C^T = B^T A^T
cublasSgemm(h, CUBLAS_OP_N, CUBLAS_OP_N,
            N, N, N, &alpha,
            dB, N, dA, N, &beta, dC, N);
cublasDestroy(h);
// kompilasi: nvcc -O3 mm.cu -lcublas
```

- ▶ **cuBLAS** — aljabar linier (GEMM, GEMV, dan lainnya), diatur khusus untuk tiap generasi GPU.
- ▶ **cuDNN** — operasi *deep learning*: konvolusi, *attention*, normalisasi. PyTorch memanggilnya di belakang layar.
- ▶ cuBLAS memakai urutan *column-major* (warisan Fortran). Trik menukar A dan B di atas memberi hasil *row-major* yang benar tanpa transpos.

Intinya

Menulis *kernel* sendiri adalah cara terbaik untuk **memahami**. Memakai pustaka adalah cara terbaik untuk **produksi**. Insinyur yang baik tahu kapan memakai yang mana.

Roofline untuk GPU: naif, tiled, cuBLAS



Ilustrasi: atap memakai angka kira-kira GPU kelas T4 (FP32 sekitar 8 TFLOPS, sekitar 320 GB/s). Posisi titik hanya contoh; ukur sendiri di praktikum.

Batas atas performa:

$$\min(\text{puncak}, I \times \text{bandwidth})$$

dengan $I = \text{arithmetic intensity}$.

- ▶ Titik di kiri atap miring: *memory-bound*. Naikkan *intensity* (*tiling, fusion*).
- ▶ Titik di bawah atap datar: *compute-bound*. Perbaiki eksekusi (*occupancy, ILP, Tensor Core*).
- ▶ Jarak titik ke atap = ruang yang masih bisa diperbaiki.

Intinya

Roofline menjawab pertanyaan penting sebelum mengoptimasi: optimasi mana yang **mungkin** berguna?

Tensor Core dan presisi rendah

Tensor Core: perangkat keras khusus perkalian matriks kecil

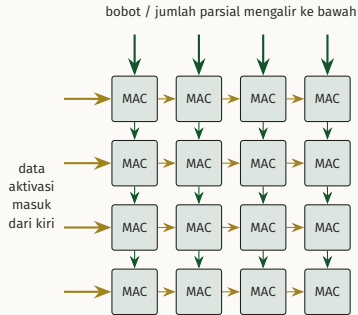
- ▶ Core biasa (CUDA core) mengerjakan satu *fused multiply-add* skalar per siklus.
- ▶ **Tensor Core** mengerjakan $D = A \times B + C$ untuk *tile* matriks kecil sekaligus dalam satu instruksi, umumnya dengan masukan presisi rendah dan akumulasi presisi lebih tinggi.
- ▶ Diakses lewat cuBLAS/cuDNN, PyTorch (*autocast*), atau API WMMA di CUDA.
- ▶ Throughput *Tensor Core* jauh di atas FP32 biasa; inilah alasan utama pelatihan AI modern memakai presisi campuran (*mixed precision*).

Format	Tanda	Eksponen	Mantisa
FP32	1	8	23
FP16	1	5	10
BF16	1	8	7
FP8 E4M3	1	4	3
FP8 E5M2	1	5	2

- ▶ **BF16** memakai eksponen sama dengan FP32: rentang nilainya sama, hanya ketelitiannya turun. Karena itu jarang *overflow* saat pelatihan.
- ▶ **FP16** lebih teliti tetapi rentangnya sempit; butuh *loss scaling*.
- ▶ **FP8** memangkas memori dan *bandwidth* lagi separuh; butuh penskalaan per tensor.

Ragam akselerator: TPU dan FPGA

TPU dan systolic array



- ▶ **Systolic array:** kisi unit *multiply-accumulate* (MAC). Data “dipompa” berirama dari satu unit ke tetangganya, seperti darah dipompa jantung.
- ▶ Setiap nilai yang diambil dari memori dipakai berulang kali saat melewati banyak MAC, tanpa bolak-balik ke memori. Ini *tiling* yang dibakar ke dalam silikon.
- ▶ **TPU** (Google) dibangun di sekitar *systolic array* besar untuk perkalian matriks pada *neural network*.

Harga yang dibayar: sangat efisien untuk perkalian matriks padat, kurang luwes untuk pekerjaan lain.

FPGA: membangun sirkuit untuk satu tugas

- ▶ **FPGA** (*Field-Programmable Gate Array*): chip berisi blok logika, memori kecil, dan unit DSP yang sambungannya dapat diatur ulang.
- ▶ Program tidak dieksekusi instruksi demi instruksi; program **menjadi sirkuit**. Data mengalir melewati *pipeline* perangkat keras, satu hasil per siklus.
- ▶ Ditulis dengan Verilog/VHDL atau *high-level synthesis* dari C/C++.

Cocok untuk

- ▶ Pemrosesan paket jaringan pada *line rate* (SmartNIC)
- ▶ *Latency* sangat rendah dan dapat diprediksi: perdagangan frekuensi tinggi, kendali industri
- ▶ Inferensi model kecil dengan konsumsi daya rendah

Harganya

Waktu kompilasi (sintesis) lama, pemrogramannya sulit, dan frekuensi *clock*-nya lebih rendah daripada CPU atau GPU.

Memilih akselerator: tidak ada yang menang di semua hal

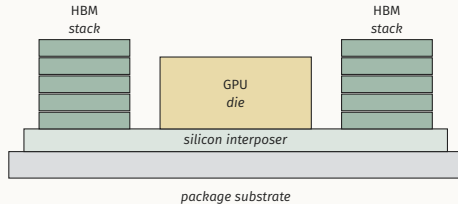
	Kekuatan	Kelemahan	Contoh pemakaian
CPU	Logika bercabang, <i>latency</i> rendah per tugas, serba bisa	<i>Throughput</i> paralel terbatas	Basis data, <i>web server</i> , pengendali GPU
GPU	Ribuan <i>thread</i> , <i>bandwidth</i> tinggi, ekosistem matang (CUDA)	Daya besar, transfer PCIe, <i>divergence</i>	Pelatihan AI, simulasi, citra
TPU / ASIC	Efisiensi energi tertinggi untuk satu jenis operasi	Kurang luwes; tersedia terbatas	Pelatihan dan inferensi <i>neural network</i> skala besar
FPGA	<i>Pipeline</i> khusus, <i>latency</i> dapat diprediksi	Sulit diprogram, sintesis lama	Jaringan, <i>edge</i> , <i>trading</i>

Intinya

Heterogeneous computing berarti setiap bagian pekerjaan dikirim ke prosesor yang paling cocok, dan CPU tetap menjadi “dirigen” yang mengatur semuanya.

Memori dan interkoneksi

HBM: memori yang ditumpuk di sebelah prosesor

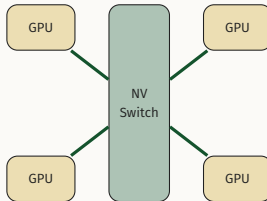
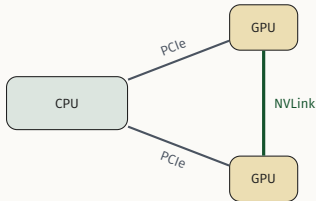


Skema disederhanakan, tidak berskala.

- ▶ **HBM** (*High Bandwidth Memory*): beberapa *die* DRAM ditumpuk vertikal dan diletakkan di samping GPU di atas *interposer*.
- ▶ Ribuan jalur data pendek memberi *bandwidth* beberapa TB/s pada GPU pusat data, jauh di atas DDR biasa.
- ▶ Harganya: mahal, kapasitas per GPU terbatas, dan rumit diproduksi.

Ingat *roofline*: *bandwidth* lebih tinggi menggeser atap miring ke atas. Banyak beban AI (terutama inferensi LLM) dibatasi oleh memori, bukan FLOPS.

PCIe, NVLink, NVSwitch, dan CXL



- ▶ **PCIe:** jalur standar CPU ke GPU. Satu slot x16 generasi 4 sekitar 32 GB/s per arah; tiap generasi baru kira-kira menggandakannya.
- ▶ **NVLink:** sambungan langsung GPU ke GPU milik NVIDIA, *bandwidth* berkali lipat PCIe.
- ▶ **NVSwitch:** switch NVLink sehingga setiap GPU dalam satu server dapat bicara dengan setiap GPU lain pada kecepatan penuh.
- ▶ **CXL (Compute Express Link):** standar terbuka di atas fisik PCIe yang menambahkan koherensi *cache* dan *memory pooling*: memori dapat dibagi atau ditambahkan lintas perangkat.

Intinya

Di dalam satu server pun, “bagaimana komputer berkomunikasi” tetap pertanyaan utamanya. Pertemuan 15 melanjutkannya ke ribuan GPU lintas server.

Performa per watt

- ▶ Pusat data dibatasi oleh **daya listrik dan pendinginan**, bukan hanya harga chip. Ukuran yang makin penting: **FLOPS per watt** dan **energi per tugas**.
- ▶ Memindahkan data sering memakan energi lebih besar daripada menghitungnya, apalagi bila datanya dari DRAM yang jauh. Optimasi *locality* adalah optimasi energi.
- ▶ Presisi rendah menghemat energi dua kali: operasinya lebih murah dan byte yang dipindahkan lebih sedikit.
- ▶ Daftar **Green500** meranking superkomputer berdasarkan FLOPS per watt, mendampingi TOP500.

Hitungan sederhana untuk laporan

energi (J) $\approx$ daya rata-rata (W) $\times$ waktu (s)

$$\text{efisiensi} = \frac{\text{FLOP yang dikerjakan}}{\text{energi}}$$

Daya GPU dapat dibaca dengan `nvidia-smi -query-gpu=power.draw -format=csv -l 1` selama *kernel* berjalan.

Intinya

Kernel yang 2 kali lebih cepat pada daya yang sama juga 2 kali lebih hemat energi. Optimasi performa sekaligus optimasi biaya dan jejak karbon.

Praktikum

Praktikum: naif → *tiled* → cuBLAS di Colab

1. Buka Google Colab, pilih *runtime* GPU, jalankan `!nvidia-smi` dan catat jenis GPU-nya.
2. Mulai dari *kernel* naif pertemuan 12. Tambahkan versi *tiled* (`TILE = 8, 16, 32`) dan versi cuBLAS dalam satu program.
3. Ukur waktu tiap versi dengan `cudaEvent` untuk $N = 512, 1024, 2048, 4096$. Jalankan sekali untuk pemanasan, lalu ambil median dari 5 kali.
4. Hitung $\text{GFLOPS} = 2N^3 / (t \times 10^9)$, dan periksa kebenaran: bandingkan hasil tiap versi dengan cuBLAS (selisih maksimum kecil).
5. Hitung *arithmetic intensity* teoretis tiap versi, cari spesifikasi puncak GPU Anda, lalu gambar *roofline* beserta tiga titik hasil pengukuran.
6. Opsional: jalankan `ncu` pada versi *tiled* dan catat *achieved occupancy* serta *memory throughput*.

Praktikum: kerangka pengukuran

```
cudaEvent_t mulai, selesai;
cudaEventCreate(&mulai);
cudaEventCreate(&selesai);
cudaEventRecord(mulai);
matmulTiled<<<grd, blk>>>(dA, dB, dC, N);
cudaEventRecord(selesai);
cudaEventSynchronize(selesai);
float ms = 0.0f;
cudaEventElapsedTime(&ms, mulai, selesai);
double gflops = 2.0 * N * N * N / (ms * 1e6);
printf("N=%d  %.3f ms  %.1f GFLOPS\n",
      N, ms, gflops);
```

```
%%writefile mm.cu
// ... seluruh kode di sini ...

!nvcc -O3 -arch=sm_75 mm.cu \
      -o mm -lcublas
!./mm 2048
```

Sesuaikan -arch dengan GPU yang didapat (misalnya sm_75 untuk T4). Sel %%writefile dan ! adalah sintaks Colab/Jupyter.

Intinya

Ingat: peluncuran *kernel* bersifat *asynchronous*. Tanpa cudaEventSynchronize, yang terukur hanya waktu mengantre, bukan waktu kerja.

Tugas untuk pertemuan berikutnya

1. Selesaikan praktikum dan kumpulkan: kode, tabel waktu dan GFLOPS, grafik *roofline* dengan tiga titik, serta satu paragraf penjelasan mengapa urutan kecepatannya demikian.
2. Baca PMPP Bab 5–6 (*memory architecture, performance considerations*) dan bagian pola paralel (*reduction, histogram*) yang relevan dengan rencana tugas besar Anda.
3. Serahkan **rencana tugas besar UAS** satu halaman: komputasi yang dipilih, versi yang akan dibuat (*serial, multicore*, banyak node, GPU), dan metrik yang akan diukur.
4. Siapkan Docker untuk menjalankan klaster Spark kecil di laptop.

Pertemuan depan

Kita naik dari satu GPU ke ribuan mesin biasa: HDFS, MapReduce, dan Spark — pola paralel untuk data yang terlalu besar bagi satu mesin mana pun.

Terima kasih

Pertanyaan?