

# Travaux Pratiques : Optimisation Mémoire CUDA

---

## Objectifs

À l'issue de ce TD, l'étudiant sera capable de :

- Comprendre et quantifier l'impact des mémoires CUDA sur les performances
  - Implémenter des optimisations utilisant Shared Memory et Registres
  - Utiliser NVIDIA Nsight Compute (NCU) pour profiler et diagnostiquer les goulets
  - Analyser les métriques de bande passante et latence mémoire
  - Appliquer les stratégies de coalescence mémoire
- 

## PARTIE 1 : FONDAMENTAUX DE L'OPTIMISATION MÉMOIRE

### 1.1 Hiérarchie des Coûts Mémoire

Type Mémoire	Latence	Capacité	Scope
<b>Registres</b>	1 cycle	256 KB/SM	Privé thread
<b>Shared Memory</b>	20-30 cycles	96-192 KB/SM	Block
<b>L1 Cache</b>	20-30 cycles	32-192 KB/SM	SM
<b>L2 Cache</b>	150-200 cycles	4-8 MB	GPU entier
<b>Global Memory</b>	200-400+ cycles	2-80 GB	Tous threads

**Coût relatif pour une lecture :**

- Registre : **1x** (référence)
- Shared Memory : **~20-30x**
- L2 Cache : **~150-200x**
- Global Memory : **~200-400x**

### 1.2 Principes d'Optimisation Essentiels

**Réduction de Latence** : Les accès à Global Memory doivent être couverts par du calcul.

**Coalescence Mémoire** : Les threads d'un warp doivent accéder à des adresses contiguës.

**Réutilisation de Données** : Charger une fois en Shared Memory, utiliser N fois.

**Occupation de Registres** : Limiter les registres par thread pour augmenter le parallélisme.

---

## PARTIE 2 : EXERCICES PRATIQUES

Exercice 1 : Addition de Vecteur - Code de Base vs Optimisé

## 1.1 - Code de Base (Naïf)

```
// vector_add_naive.cu
#include <cuda_runtime.h>
#include <iostream>
#include <vector>
#include <stdexcept>

constexpr int VECTOR_SIZE = 1024 * 1024;

inline void checkCuda(cudaError_t err, const char* msg) {
    if (err != cudaSuccess) {
        std::cerr << msg << " : " << cudaGetErrorString(err) << '\n';
        std::exit(EXIT_FAILURE);
    }
}

__global__ void vector_add_naive(float *A, float *B, float *C, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) {
        // Chaque thread accède directement à Global Memory
        C[idx] = A[idx] + B[idx];
    }
}

int main() {
    try {
        const int N      = VECTOR_SIZE;
        const size_t nBytes = N * sizeof(float);

        // Host buffers (C++)
        std::vector<float> hA(N, 1.0f);
        std::vector<float> hB(N, 2.0f);
        std::vector<float> hC(N, 0.0f);

        // Device buffers
        float *dA = nullptr, *dB = nullptr, *dC = nullptr;
        checkCuda(cudaMalloc(&dA, nBytes), "cudaMalloc dA");
        checkCuda(cudaMalloc(&dB, nBytes), "cudaMalloc dB");
        checkCuda(cudaMalloc(&dC, nBytes), "cudaMalloc dC");

        checkCuda(cudaMemcpy(dA, hA.data(), nBytes,
cudaMemcpyHostToDevice), "cpy A");
        checkCuda(cudaMemcpy(dB, hB.data(), nBytes,
cudaMemcpyHostToDevice), "cpy B");

        dim3 block(256);
        dim3 grid((N + block.x - 1) / block.x);

        cudaEvent_t start, stop;
        checkCuda(cudaEventCreate(&start), "event create start");
        checkCuda(cudaEventCreate(&stop), "event create stop");
    }
}
```

```

// Naive
checkCuda(cudaEventRecord(start), "record start naive");
vector_add_naive<<<grid, block>>>(dA, dB, dC, N);
checkCuda(cudaEventRecord(stop), "record stop naive");
checkCuda(cudaEventSynchronize(stop), "sync stop naive");

float msNaive = 0.0f;
checkCuda(cudaEventElapsedTime(&msNaive, start, stop), "elapsed
naive");
std::cout << "Naive Vector Add: " << msNaive << " ms\n";

checkCuda(cudaMemcpy(hC.data(), dC, nBytes,
cudaMemcpyDeviceToHost), "cpy C naive");

// Vérification simple
bool ok = true;
for (int i = 0; i < 10; ++i) {
    if (hC[i] != 3.0f) { ok = false; break; }
}
std::cout << "Check: " << (ok ? "OK" : "FAILED") << '\n';

cudaEventDestroy(start);
cudaEventDestroy(stop);
cudaFree(dA);
cudaFree(dB);
cudaFree(dC);

return ok ? EXIT_SUCCESS : EXIT_FAILURE;
} catch (const std::exception& e) {
    std::cerr << "Exception: " << e.what() << '\n';
    return EXIT_FAILURE;
}
}
}

```

### Compilation et exécution baseline :

```

nvcc -O3 -std=c++17 -lineinfo vector_add.cu -o vector_add
./vector_add_naive

```

### 1.2 - Code Optimisé avec Coalescence et Stride

```

// vector_add_optimized.cu
#include <cuda_runtime.h>
#include <iostream>
#include <vector>
#include <stdexcept>

const int VECTOR_SIZE = 1024 * 1024;

```

```

inline void checkCuda(cudaError_t err, const char* msg) {
    if (err != cudaSuccess) {
        std::cerr << msg << " : " << cudaGetErrorString(err) << '\n';
        std::exit(EXIT_FAILURE);
    }
}

__global__ void vector_add_optimized(float *A, float *B, float *C, int N) {
    // Accès coalescé : threads consécutifs lisent adresses consécutives
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // Traiter plusieurs éléments par thread (meilleur ratio calcul/accès)
    int stride = gridDim.x * blockDim.x;

    for (int i = idx; i < N; i += stride) {
        C[i] = A[i] + B[i];
    }
}

int main() {
    try {
        const int N      = VECTOR_SIZE;
        const size_t nBytes = N * sizeof(float);

        // Host buffers (C++)
        std::vector<float> hA(N, 1.0f);
        std::vector<float> hB(N, 2.0f);
        std::vector<float> hC(N, 0.0f);

        // Device buffers
        float *dA = nullptr, *dB = nullptr, *dC = nullptr;
        checkCuda(cudaMalloc(&dA, nBytes), "cudaMalloc dA");
        checkCuda(cudaMalloc(&dB, nBytes), "cudaMalloc dB");
        checkCuda(cudaMalloc(&dC, nBytes), "cudaMalloc dC");

        checkCuda(cudaMemcpy(dA, hA.data(), nBytes,
cudaMemcpyHostToDevice), "cpy A");
        checkCuda(cudaMemcpy(dB, hB.data(), nBytes,
cudaMemcpyHostToDevice), "cpy B");

        dim3 block(256);
        dim3 grid((N + block.x - 1) / block.x);

        cudaEvent_t start, stop;
        checkCuda(cudaEventCreate(&start), "event create start");
        checkCuda(cudaEventCreate(&stop), "event create stop");

        // Optimized (grid-stride loop)
        checkCuda(cudaMemset(dC, 0, nBytes), "memset C");
        checkCuda(cudaEventRecord(start), "record start opt");
        vector_add_optimized<<<grid, block>>>(dA, dB, dC, N);
        checkCuda(cudaEventRecord(stop), "record stop opt");
        checkCuda(cudaEventSynchronize(stop), "sync stop opt");
    }
}

```

```

        float msOpt = 0.0f;
        checkCuda(cudaEventElapsedTime(&msOpt, start, stop), "elapsed
opt");
        std::cout << "Optimized Vector Add: " << msOpt << " ms\n";

        checkCuda(cudaMemcpy(hC.data(), dC, nBytes,
cudaMemcpyDeviceToHost), "cpy C opt");

        // Vérification simple
        bool ok = true;
        for (int i = 0; i < 10; ++i) {
            if (hC[i] != 3.0f) { ok = false; break; }
        }
        std::cout << "Check: " << (ok ? "OK" : "FAILED") << '\n';

        cudaEventDestroy(start);
        cudaEventDestroy(stop);
        cudaFree(dA);
        cudaFree(dB);
        cudaFree(dC);

        return ok ? EXIT_SUCCESS : EXIT_FAILURE;
    } catch (const std::exception& e) {
        std::cerr << "Exception: " << e.what() << '\n';
        return EXIT_FAILURE;
    }
}

```

### 1.3 - Profiling avec NCU

**Compiler avec information de debug :**

```

nvcc -O3 -lineinfo -o vector_add_naive vector_add_naive.cu
nvcc -O3 -lineinfo -o vector_add_optimized vector_add_optimized.cu

```

**Profiler le code naïf :**

```

ncu --set full ./vector_add_naive > report_naive.ncu-report

```

**Profiler le code optimisé :**

```

ncu --set full ./vector_add_optimized > report_optimized.ncu-report

```

**Analyser les rapports :**

```
ncu -i report_naive.ncu-report  
ncu -i report_optimized.ncu-report
```

## 1.4 - Métriques clés à observer

- **Memory Throughput %** : % de bande passante utilisée
- **Global Load Throughput** : Octets/sec en lecture depuis Global Memory
- **L1/TEX Cache Throughput %** : Utilisation du cache L1
- **Achieved Occupancy %** : % de warps actifs

### Questions à résoudre :

1. Quelle est la différence de performance entre naïf et optimisé ?
2. Comment le Memory Throughput % change-t-il ?
3. Quel est l'impact de la coalescence sur L1 Cache Throughput ?

---

## Exercice 2 : Matrix Multiplication - Shared Memory Optimization

### 2.1 - Naive Matrix Multiply (Global Memory seulement)

```
// matmul_naive.cu
#include <cuda_runtime.h>
#include <iostream>
#include <vector>
#include <cmath>

inline void checkCuda(cudaError_t err, const char* msg) {
    if (err != cudaSuccess) {
        std::cerr << msg << " : " << cudaGetErrorString(err) << '\n';
        std::exit(EXIT_FAILURE);
    }
}

__global__ void matmul_naive(float *A, float *B, float *C, int n) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < n && col < n) {
        float sum = 0.0f;
        for (int k = 0; k < n; k++) {
            // Accès non-coalescé en k
            sum += A[row * n + k] * B[k * n + col];
        }
        C[row * n + col] = sum;
    }
}

int main() {
```

```

const int N      = 1024;
const size_t sz = static_cast<size_t>(N) * N;
const size_t nBytes = sz * sizeof(float);

std::vector<float> hA(sz, 1.0f);
std::vector<float> hB(sz, 2.0f);
std::vector<float> hC(sz, 0.0f);

float *dA = nullptr, *dB = nullptr, *dC = nullptr;
checkCuda(cudaMalloc(&dA, nBytes), "cudaMalloc A");
checkCuda(cudaMalloc(&dB, nBytes), "cudaMalloc B");
checkCuda(cudaMalloc(&dC, nBytes), "cudaMalloc C");

checkCuda(cudaMemcpy(dA, hA.data(), nBytes, cudaMemcpyHostToDevice),
"cpy A");
checkCuda(cudaMemcpy(dB, hB.data(), nBytes, cudaMemcpyHostToDevice),
"cpy B");

dim3 block(TILE_SIZE, TILE_SIZE);
dim3 grid((N + block.x - 1) / block.x,
(N + block.y - 1) / block.y);

cudaEvent_t start, stop;
checkCuda(cudaEventCreate(&start), "event start");
checkCuda(cudaEventCreate(&stop), "event stop");

// Naive
checkCuda(cudaEventRecord(start), "record start naive");
matmul_naive<<<grid, block>>>(dA, dB, dC, N);
checkCuda(cudaEventRecord(stop), "record stop naive");
checkCuda(cudaEventSynchronize(stop), "sync naive");

float msNaive = 0.0f;
checkCuda(cudaEventElapsedTime(&msNaive, start, stop), "elapsed
naive");

double flops = 2.0 * N * static_cast<double>(N) * N;
double gflopsNaive = (flops / 1e9) / (msNaive / 1e3);
std::cout << "Naive MatMul: " << msNaive << " ms, "
<< gflopsNaive << " GFLOPs\n";

// Optimized
checkCuda(cudaEventRecord(start), "record start opt");
matmul_optimized<<<grid, block>>>(dA, dB, dC, N);
checkCuda(cudaEventRecord(stop), "record stop opt");
checkCuda(cudaEventSynchronize(stop), "sync opt");

float msOpt = 0.0f;
checkCuda(cudaEventElapsedTime(&msOpt, start, stop), "elapsed opt");

double gflopsOpt = (flops / 1e9) / (msOpt / 1e3);
std::cout << "Optimized MatMul: " << msOpt << " ms, "
<< gflopsOpt << " GFLOPs\n";

```

```

    checkCuda(cudaMemcpy(hC.data(), dC, nBytes, cudaMemcpyDeviceToHost),
    "cpy C");

    // Vérification rudimentaire (pour A=1, B=2 → chaque C[i]=2*N)
    float ref = 2.0f * N;
    double maxErr = 0.0;
    for (int i = 0; i < 10; ++i) {
        maxErr = std::max(maxErr, std::abs(hC[i] - ref));
    }
    std::cout << "Max error: " << maxErr << '\n';

    cudaFree(dA);
    cudaFree(dB);
    cudaFree(dC);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    return 0;
}

```

## 2.2 - Optimized Matrix Multiply avec Shared Memory

```

// matmul_optimized.cu
#include <cuda_runtime.h>
#include <iostream>
#include <vector>
#include <cmath>

const int N = 1024;
const int TILE_SIZE = 32; // Taille des tiles en Shared Memory

__global__ void matmul_optimized(float *A, float *B, float *C, int n) {
    // Shared memory pour tiles
    __shared__ float As[TILE_SIZE][TILE_SIZE];
    __shared__ float Bs[TILE_SIZE][TILE_SIZE];

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    float sum = 0.0f;

    // Boucle sur les tiles
    for (int t = 0; t < (n + TILE_SIZE - 1) / TILE_SIZE; t++) {
        // Charger tile de A en Shared Memory
        if (row < n && t * TILE_SIZE + tx < n) {
            As[ty][tx] = A[row * n + t * TILE_SIZE + tx];
        } else {
            As[ty][tx] = 0.0f;
        }
    }
}
```

```

    // Charger tile de B en Shared Memory
    if (col < n && t * TILE_SIZE + ty < n) {
        Bs[ty][tx] = B[(t * TILE_SIZE + ty) * n + col];
    } else {
        Bs[ty][tx] = 0.0f;
    }

    __syncthreads();

    // Calculer avec données en Shared Memory (très rapide !)
    for (int k = 0; k < TILE_SIZE; k++) {
        sum += As[ty][k] * Bs[k][tx];
    }

    __syncthreads();
}

if (row < n && col < n) {
    C[row * n + col] = sum;
}
}

int main() {
    const int N      = 1024;
    const size_t sz  = static_cast<size_t>(N) * N;
    const size_t nBytes = sz * sizeof(float);

    std::vector<float> hA(sz, 1.0f);
    std::vector<float> hB(sz, 2.0f);
    std::vector<float> hC(sz, 0.0f);

    float *dA = nullptr, *dB = nullptr, *dC = nullptr;
    checkCuda(cudaMalloc(&dA, nBytes), "cudaMalloc A");
    checkCuda(cudaMalloc(&dB, nBytes), "cudaMalloc B");
    checkCuda(cudaMalloc(&dC, nBytes), "cudaMalloc C");

    checkCuda(cudaMemcpy(dA, hA.data(), nBytes, cudaMemcpyHostToDevice),
    "cpy A");
    checkCuda(cudaMemcpy(dB, hB.data(), nBytes, cudaMemcpyHostToDevice),
    "cpy B");

    dim3 block(TILE_SIZE, TILE_SIZE);
    dim3 grid((N + block.x - 1) / block.x,
              (N + block.y - 1) / block.y);

    cudaEvent_t start, stop;
    checkCuda(cudaEventCreate(&start), "event start");
    checkCuda(cudaEventCreate(&stop), "event stop");

    // Naive
    checkCuda(cudaEventRecord(start), "record start naive");
    matmul_naive<<<grid, block>>>(dA, dB, dC, N);
    checkCuda(cudaEventRecord(stop), "record stop naive");
}

```

```

    checkCuda(cudaEventSynchronize(stop), "sync naive");

    float msNaive = 0.0f;
    checkCuda(cudaEventElapsedTime(&msNaive, start, stop), "elapsed
naive");

    double flops = 2.0 * N * static_cast<double>(N) * N;
    double gflopsNaive = (flops / 1e9) / (msNaive / 1e3);
    std::cout << "Naive MatMul: " << msNaive << " ms, "
        << gflopsNaive << " GFLOPs\n";

    // Optimized
    checkCuda(cudaEventRecord(start), "record start opt");
    matmul_optimized<<<grid, block>>>(dA, dB, dC, N);
    checkCuda(cudaEventRecord(stop), "record stop opt");
    checkCuda(cudaEventSynchronize(stop), "sync opt");

    float msOpt = 0.0f;
    checkCuda(cudaEventElapsedTime(&msOpt, start, stop), "elapsed opt");

    double gflopsOpt = (flops / 1e9) / (msOpt / 1e3);
    std::cout << "Optimized MatMul: " << msOpt << " ms, "
        << gflopsOpt << " GFLOPs\n";

    checkCuda(cudaMemcpy(hC.data(), dC, nBytes, cudaMemcpyDeviceToHost),
"cpy C");

    // Vérification rudimentaire (pour A=1, B=2 → chaque C[i]=2*N)
    float ref = 2.0f * N;
    double maxErr = 0.0;
    for (int i = 0; i < 10; ++i) {
        maxErr = std::max(maxErr, std::abs(hC[i] - ref));
    }
    std::cout << "Max error: " << maxErr << '\n';

    cudaFree(dA);
    cudaFree(dB);
    cudaFree(dC);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    return 0;
}

```

## 2.3 - Profiling comparatif

```

# Compiler avec -lineinfo
nvcc -O3 -lineinfo -o matmul_naive matmul_naive.cu
nvcc -O3 -lineinfo -o matmul_optimized matmul_optimized.cu

# Profiler

```

```

ncu --set full ./matmul_naive > report_mm_naive.ncu-report
ncu --set full ./matmul_optimized > report_mm_optimized.ncu-report

# Afficher rapports
ncu -i report_mm_naive.ncu-report
ncu -i report_mm_optimized.ncu-report

```

## 2.4 - Métriques essentielles pour MatMul

- **Global Load Throughput** : Doit diminuer drastiquement
- **Shared Memory Throughput** : Doit être élevé en version optimisée
- **Memory Throughput Utilization** : Comparaison naive vs optimisé
- **Cache Hit Ratio** : Amélioration L1/L2 en version optimisée

### Questions :

1. Quel est le ratio de performance (speedup) entre naïve et optimisé ?
  2. Quel est le reduction factor pour Global Load Throughput ?
  3. Quel est le bottleneck identifié par NCU pour chaque version ?
- 

## Exercice 3 : Bank Conflicts en Shared Memory

### 3.1 - Shared Memory avec Bank Conflicts

```

// bank_conflicts.cu
#include <stdio.h>
#include <cuda_runtime.h>

__global__ void transpose_with_bank_conflicts(float *in, float *out, int N)
{
    __shared__ float s_data[32][32];

    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Lecture coalescée depuis Global Memory
    s_data[ty][tx] = in[y * N + x];
    __syncthreads();

    // Accès transposé : BANK CONFLICTS !
    // Threads dans un warp accèdent à s_data[tx][ty]
    // Cela crée des conflits de banques
    out[x * N + y] = s_data[tx][ty];
}

int main() {
    const int N = 1024;
}

```

```

float *d_in, *d_out;
int size = N * N * sizeof(float);

cudaMalloc(&d_in, size);
cudaMalloc(&d_out, size);

cudaMemset(d_in, 1, size);

dim3 threads(32, 32);
dim3 blocks((N + threads.x - 1) / threads.x,
             (N + threads.y - 1) / threads.y);

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);
transpose_with_bank_conflicts<<<blocks, threads>>>(d_in, d_out, N);
cudaEventRecord(stop);
cudaEventSynchronize(stop);

float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
printf("Transpose WITH bank conflicts: %.3f ms\n", milliseconds);

cudaFree(d_in);
cudaFree(d_out);

return 0;
}

```

### 3.2 - Shared Memory sans Bank Conflicts (avec Padding)

```

// bank_conflicts_fixed.cu
#include <stdio.h>
#include <cuda_runtime.h>

__global__ void transpose_without_bank_conflicts(float *in, float *out, int N) {
    // Padding d'1 élément : 33 au lieu de 32
    __shared__ float s_data[32][33];

    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    s_data[ty][tx] = in[y * N + x];
    __syncthreads();

    // Pas de bank conflicts grâce au padding

```

```

        out[x * N + y] = s_data[tx][ty];
    }

int main() {
    const int N = 1024;
    float *d_in, *d_out;
    int size = N * N * sizeof(float);

    cudaMalloc(&d_in, size);
    cudaMalloc(&d_out, size);

    cudaMemset(d_in, 1, size);

    dim3 threads(32, 32);
    dim3 blocks((N + threads.x - 1) / threads.x,
                 (N + threads.y - 1) / threads.y);

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaEventRecord(start);
    transpose_without_bank_conflicts<<<blocks, threads>>>(d_in, d_out, N);
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);

    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);
    printf("Transpose WITHOUT bank conflicts (padded): %.3f ms\n",
milliseconds);

    cudaFree(d_in);
    cudaFree(d_out);

    return 0;
}

```

### 3.3 - Profiling et détection de Bank Conflicts

```

nvcc -O3 -lineinfo -o bank_conflicts bank_conflicts.cu
nvcc -O3 -lineinfo -o bank_conflicts_fixed bank_conflicts_fixed.cu

# Profiler
ncu --set full ./bank_conflicts > report_bc.ncu-report
ncu --set full ./bank_conflicts_fixed > report_bc_fixed.ncu-report

# Analyser
ncu -i report_bc.ncu-report
ncu -i report_bc_fixed.ncu-report

```

## Métriques à analyser :

- **Shared Memory Throughput** : Différence avec/sans padding
- **Warp State Statistics** : Nombre de stalls
- **Shared Memory Bank Conflicts** : NCU détecte automatiquement

## Questions :

1. Quel est l'impact du padding sur les performances ?
2. Quel est le ratio de Bank Conflicts détecté ?
3. À quoi correspondent les stalls observés ?

---

## PARTIE 3 : GUIDE D'UTILISATION DE NVIDIA NSIGHT COMPUTE (NCU)

### 3.1 - Installation et Setup

```
# Sur Debian/Ubuntu
sudo apt-get install nvidia-nsgt-compute

# Ou télécharger depuis : https://developer.nvidia.com/nsight-compute
# Version CLI: ncu
# Version GUI: nsight-compute
```

### 3.2 - Utilisation en CLI

#### Mode simple : Profiler rapide

```
# Profiler avec configuration par défaut
ncu ./mon_programme

# Profiler avec ensemble de métriques complet
ncu --set full ./mon_programme

# Profiler avec sortie fichier
ncu --set full ./mon_programme > report.ncu-report
```

#### Mode avancé : Sélection de métriques

```
# Profiler avec ensemble "compute_api_trace"
ncu --set compute_api_trace ./mon_programme

# Profiler avec ensemble "memory_workload_analysis"
ncu --set memory_workload_analysis ./mon_programme
```

```
# Profiler avec ensemble "sm_warp_occupancy"  
ncu --set sm_warp_occupancy ./mon_programme
```

## Profiler des kernels spécifiques

```
# Profiler le kernel "my_kernel" uniquement  
ncu --kernel-regex my_kernel ./mon_programme  
  
# Profiler le 5ème lancement du kernel "my_kernel"  
ncu --kernel-regex my_kernel --kernel-id 5 ./mon_programme
```

## 3.3 - Interpréter les rapports NCU

### Sections principales du rapport

#### 1. GPU Speed Of Light Throughput

Indique si le GPU est sous-utilisé (< 60% = problème).

```
Metric Value | Problème si < 60%  
Memory Throughput | Accès Global Memory inefficace  
Compute Throughput | Pas assez de parallélisme  
DRAM Throughput | Saturation bande passante
```

#### 2. Launch Statistics

```
Block Size = 256 threads → Bon pour la plupart des GPUs  
Shared Memory = 48 KB → Vérifie occupation vs registres  
Dynamic Shared Memory → Shared Memory allouée dynamiquement
```

#### 3. Memory Workload Analysis

```
Global Load Throughput → Octets/sec lus depuis Global Memory  
Global Store Throughput → Octets/sec écrits vers Global Memory  
L1 Cache Throughput → Efficacité L1 cache  
L2 Cache Throughput → Efficacité L2 cache  
DRAM Throughput → Bande passante DRAM utilisée
```

#### 4. Warp State Statistics

```
Issue Efficiency → % de warps émis par cycle  
Register Bank Conflicts → Conflits accès registres
```

## 5. Source Code Analysis

Montre les hotspots ligne par ligne (si compilé avec -lineinfo).

### 3.4 - Workflow de Profiling Efficace

1. Compiler avec -lineinfo  
nvcc -O3 -lineinfo -o programme programme.cu
2. Profiler version baseline  
ncu --set full ./programme > baseline.ncu-report
3. Analyser rapport  
ncu -i baseline.ncu-report  
→ Identifier goulot d'étranglement
4. Optimiser le code
5. Profiler version optimisée  
ncu --set full ./programme\_opt > optimized.ncu-report
6. Comparer rapports  
→ Vérifier amélioration des métriques clés
7. Itérer jusqu'à saturation performances

### 3.5 - Commandes NCU essentielles

```
# Lister toutes les métriques disponibles
ncu --list-metrics

# Lister toutes les sections disponibles
ncu --set guidance ./programme

# Profiler avec guidance (recommandations NVIDIA)
ncu --set guidance ./programme

# Exporter rapport en CSV
ncu --set full --export report.csv ./programme

# Afficher rapport interactif
ncu -i report.ncu-report

# Profiler sur GPU spécifique
ncu --device 0 ./programme
```

### 3.6 - Profiling Avancé : Isolation de problèmes

#### Problème : Global Memory Throughput faible

```
# Profiler avec focus mémoire  
ncu --set memory_workload_analysis ./programme  
  
# Analyser coalescence  
ncu --set memory_properties ./programme  
  
# Vérifier bank conflicts  
ncu --set shared_memory ./programme
```

#### Actions correctives :

- Vérifier accès coalescés (threads consécutifs accèdent adresses consécutives)
- Augmenter réutilisation données (Shared Memory)
- Aligner accès sur taille transaction (128 octets)

#### Problème : Low Compute Throughput

```
# Profiler occupancy  
ncu --set sm_warp_occupancy ./programme  
  
# Analyser registres  
ncu --set registers ./programme
```

#### Actions correctives :

- Réduire registres par thread (compiler avec `--maxrregcount`)
- Augmenter blocks par SM
- Vérifier branches divergentes

---

## PARTIE 4 : EXERCICES D'ANALYSE

### Exercice 4 : Analyse Complète avec NCU

Pour chaque exercice précédent (Vector Add, MatMul, Transpose) :

1. **Performance** : Temps de calcul (naive vs optimisé)

2. **Tableau de métriques** :

- Memory Throughput %
- Global Load Throughput (GB/s)
- L1/L2 Cache Throughput %
- Achieved Occupancy %

### 3. Analyse goulot :

- Quel est le goulot selon NCU ?
- Est-ce mémoire ou calcul ?
- Qu'elle est la marge d'amélioration ?

### 4. Recommandations :

- Selon les rapports NCU, qu'elle optimisation appliquer ?
- 

## PARTIE 5 : PROJET FINAL

Projet : Implémenter et Optimiser une Réduction Parallèle

**Objectif** : Implémenter la réduction parallèle (somme d'un vecteur) en 3 versions :

1. **Version 1 (Baseline)** : Global Memory seulement
2. **Version 2 (Shared Memory)** : Utiliser Shared Memory
3. **Version 3 (Fully Optimized)** : Registres + Shared Memory + Bank Conflict Resolution

**Attendus :**

- Codes sources compilables
  - Rapports NCU pour les 3 versions
  - Analyse comparative (tableau performances)
  - Graphe speedup vs version baseline
  - Recommandations d'optimisation
- 

## PARTIE 6 : RESSOURCES ET RÉFÉRENCES

Documentation NVIDIA

- [CUDA C++ Programming Guide](#)
- [Nsight Compute Profiling Guide](#)
- [CUDA Best Practices Guide](#)

Tutoriels Vidéo

- "Intro to NVIDIA Nsight Compute" (NVIDIA Developer)
- "Optimizing Parallel Reduction in CUDA" (Mark Harris, NVIDIA)

Outils

- NVIDIA Nsight Compute (ncu) : Profiler
- NVIDIA Nsight Systems : Analyse système globale
- nvidia-smi : Monitoring GPU basique

Commandes Utiles

```
# Vérifier disponibilité des compteurs GPU
ncu --check

# Lister GPUs disponibles
nvidia-smi

# Monitoring en temps réel
nvidia-smi dmon

# Compiler avec optimisations et infos debug
nvcc -O3 -lineinfo -arch sm_80 -o programme programme.cu

# Exécuter avec profiling NVIDIA-SMI
nvidia-smi stats -i 0

# Vérifier propriétés GPU
nvidia-smi --query-gpu=name,compute_cap,memory.total --format=csv
```