

CUDA “avancé”

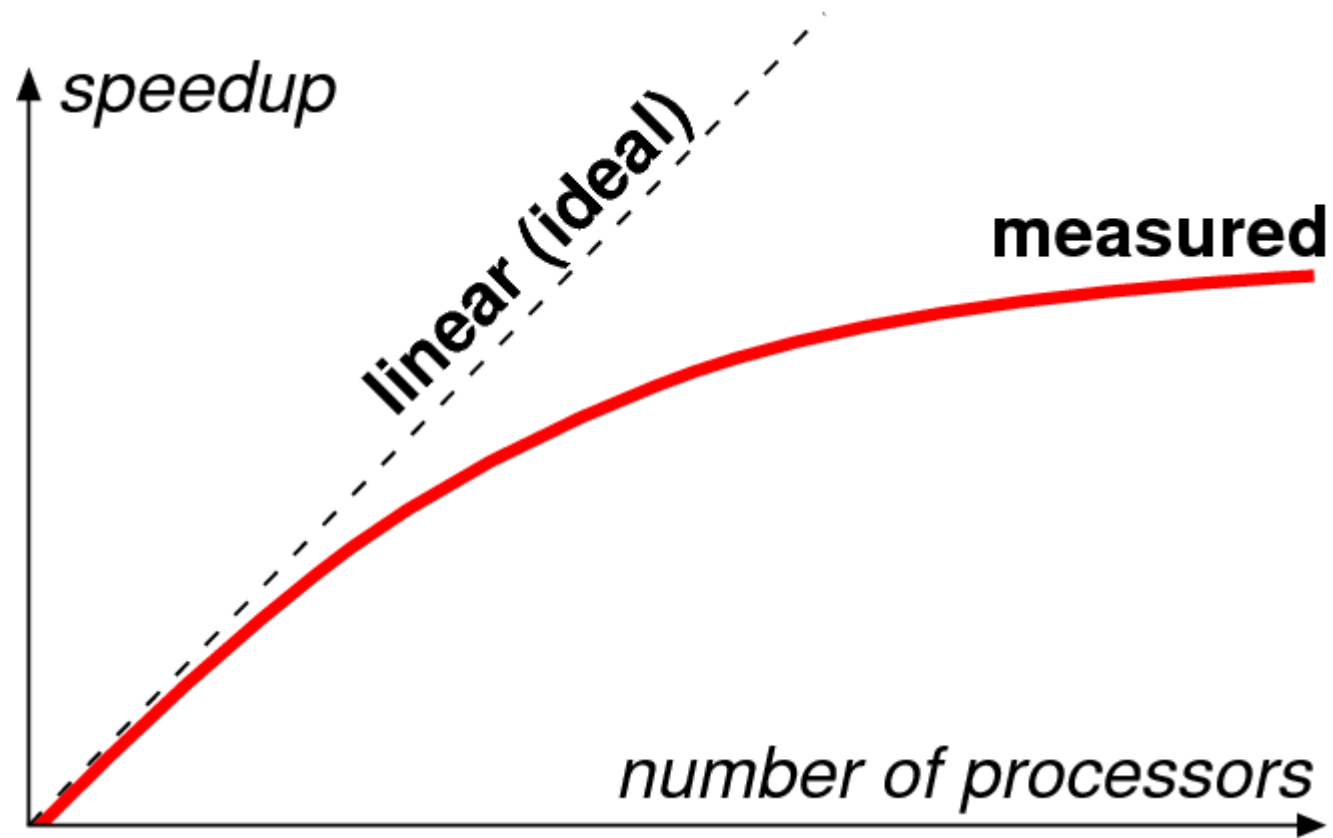
Printemps 2025

Contenu

- Notion de speedup
- Profiler son code GPU
- Error checking

Vect Add

- Nous utiliserons un code “jouet” pour illustrer les différents concepts/techniques
- Il s’agit de l’addition de vecteurs
 - Il s’agit de l’opération $\mathbf{c} = \mathbf{a} + \mathbf{b}$
 - \mathbf{a} , \mathbf{b} et \mathbf{c} sont des vecteurs de taille N



Notion de speedup

- Normalement, on compare la performance d'un code parallèle avec l'implémentation séquentielle la plus performante connue.
- Quel sens sur un GPU ?
 - Pas d'implémentation séquentielle.
 - Pas moyen de faire varier le nombre de coeurs.
- Speedup multi-GPU ?

Performance code GPU

- Toujours intéressant de mesurer le temps d'exécution de son code.
- Comparaison avec CPU (attention aux comparaisons avantageuses pour le GPU)
- Optimisation du code.
- Comparaison de différents modèles de GPU.
- Comparaison avec différentes configurations d'exécution.

Mesurer son code
GPU

Profiler son code

- Profiler son code se fait avec nsight
- Nsight compute : ncui-ui
- Nsight systems : nsys-ui

Demo time

- Allocation avec un GPU

```
salloc --ntasks=1 --partition=shared-gpu  
--time=00:45:00 --gpus=1 --mem=24G
```

- Compilation (utilisation de C++ le `std::cout`)

```
nvcc vec_add.cu --std=c++11 -o vecadd
```

- Profiler avec `nsight compute` ou `systems`

Autre façon de mesurer le temps

```
__global__ void kernel(..., int *runtime) {  
    t_idx = ...  
    clock_t start_time = clock();  
    // kernel stuffs here  
    clock_t stop_time = clock();  
    runtime[t_idx] = (int)(stop_time - start_time);  
}
```

- Problème ? Les warps ne sont pas forcément synchro.

Autre façon de mesurer le temps

```
#include <chrono>
```

```
using namespace std::chrono;
```

```
auto start = high_resolution_clock::now();  
vectorAdd<<<NUM_BLOCKS, NUM_THREADS>>>(d_a, d_b, d_c, N);  
cudaDeviceSynchronize();  
auto end = high_resolution_clock::now();  
auto duration = duration_cast<milliseconds>(end - start);  
std::cout << "Elapsed Time: " << duration.count() << "ms" << std::endl;
```

- Problème ? Petits temps pas forcément fiables.

CUDA error checking

Error checking

- Les fonctions de l'API CUDA retourne des codes d'erreur
- La bonne pratique suggère de les vérifier

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#error-checking>

https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__ERROR.html

Check basique

```
cu_error = cudaMalloc(&data, data_size);  
  
if (cudaSuccess != cu_error) {  
    printf("Failed to allocate memory\n");  
    return 1;  
}
```

Wrapper maison

```
#define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__); }

inline void gpuAssert(
    cudaError_t code,
    const char *file,
    int line
) {
    if (code != cudaSuccess) {
        fprintf(
            stderr, "GPUassert: %s %s %d\n",
            cudaGetErrorString(code), file, line
        );
    }
}

gpuErrchk(cudaMalloc(&int_ary, nb_int*sizeof(int)));
```

Sticky vs non-sticky

- Il y a deux “types” d’erreurs retournées:
 - Sticky: une fois que cette erreur est retournée, tous les autres appels à CUDA retournent cette erreur
 - Non-sticky: disparaît à l’appel CUDA suivant
 - Mais la dernière peut être récupérée avec: `cudaPeekAtLastError()`
 - `CudaGetLastError()`, idem mais reset l’erreur en success
- Après une erreur, on peut réinitialiser le device si nécessaire:
 - `CudaDeviceReset()`: détruit tout le contexte actuel du device