Computer Architecture – Handleiding GPU's programmeren met CUDA

This is a short introduction to programming GPUs using CUDA, available in Dutch only. For an English text, we refer the reader to NVIDIA's CUDA Programming Guide: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

In deze handleiding doorlopen we het algemene idee achter het porten van CPU code naar GPU code met behulp van CUDA. CUDA is de GPGPU tookit van NVIDIA en werkt alleen met videokaarten met een NVIDIA GPU. Er is voor CUDA gekozen vanwege de laagdrempelige instap en de goede documentatie. Zie voor de documentatie van de CUDA API:
`https://docs.nvidia.com/cuda/archive/11.8.0/cuda-runtime-api/index.html`

Let op dat dit de documentatie van versie 11.8 is, welke ook staat geïnstalleerd in het environment dat kan worden gebruikt in de LIACS computerzalen.

Om te werken met CUDA heb je `nvcc` nodig, dit is NVIDIA's C(++) compiler voor CUDA. Als je werkt in de LIACS computerzalen, dan wordt dit klaargezet als je het environment van de opdracht activeert. Zie de opdracht voor nadere instructies.

1 Aanpak

We doorlopen nu het algemene idee achter het porten van CPU code naar CUDA. Zoals tijdens college is behandeld, bestaat de GPU uit honderden simpele cores in plaats van enkele complexe cores zoals bij de CPU. De grootste uitdaging is om deze cores zo goed mogelijk "bezig te houden". Dit doen we door een eenvoudige operatie uit ons probleem te isoleren en deze parallel uit te zetten op de vele cores. Zo'n operatie heet in CUDA een *kernel*. Een kernel is een simpele C-functie die vele malen parallel wordt uitgevoerd. Een enkele instantie van zo'n kernel noemen we een *thread*. Threads worden vervolgens verdeeld over *blocks*[1] en die blocks staan weer in een *grid*. Bij de aanroep van de kernel mag de programmeur zelf bepalen hoe groot die blocks zijn en hoe de blocks en het grid worden gerangschikt: één-, twee, drie-dimensionaal. Dit is belangrijk omdat threads binnen één block geheugen delen (het *shared memory* van een streaming multiprocessor) en dus met elkaar kunnen communiceren en synchroniseren.

1.1 Kernels

Stel we hebben we volgende CPU code:

```
int A[100][100]; // Initialiseer op wat voor wijze dan ook
int B[100][100];
for (int x = 0; x < 100; x++)
    for (int y = 0; y < 100; y++)
        B[x][y] = 10 * A[x][y];
```

Dit is een typische geneste for-loop zoals die ook in de filters voorkomen. Het is in dit geval makkelijk te zien dat de loop-body, namelijk de array toekenning, een operatie is die we parallel kunnen uitvoeren (er bestaan geen loop-carried dependences). We beginnen door deze operatie te isoleren in een kernel: een C-functie van het type `__global__ void`:

[1] Binnen blocks worden threads ook nog in *warps* ingedeeld, dit is vooral voor optimalisatie relevant. Zie voor meer informatie de CUDA documentatie en het hoorcollege.

```
// We nemen nu N voor het aantal kolommen, dus N=100
__global__ void
myKernel(int* A, int* B, int N)
{
  // Andere indexering: lineair geheugen in plaats van 2D-array
  B[y+N*x] = 10 * A[y+N*x];
}
```

Nu hebben we eerst nog x en y nodig. We gaan er van uit dat we een grid maken dat 100 bij 100 groot is. Dus voor elk element in het array, is er één thread. Hoe dat precies is ingedeeld, bijvoorbeeld 10 bij 10 blocks met elk 10 bij 10 threads, is nu nog niet belangrijk. We gebruiken in dit geval de globale constanten `threadIdx`, `blockIdx` en `blockDim` die door CUDA worden gedefinieerd. Bijvoorbeeld x verkrijgen we met:

```
// in myKernel()
const int x = threadIdx.x + blockDim.x * blockIdx.x;
```

Analoog zijn ook y en een eventuele z af te leiden. Vervolgens moeten we onze kernel aanroepen en eerst bepalen we daartoe hoe het grid precies ingedeeld zal worden:

```
const int N = 100; // Aantal rijen en kolommen in ons array
const dim3 blocksize( 10, 10 ); // Aantal threads per block
// Aantal blocks. let op: deze deling kan een rest hebben!
const dim3 numblocks( N / blocksize.x, N / blocksize.y );
```

Het totaal aantal blocks is dus afhankelijk van de blocksize en het aantal elementen. Verschillende verhoudingen zijn mogelijk die elk hun voor- en nadelen hebben. Het fine-tunen van de blocksize kan een performancewinst geven. Let wel op dat er een maximum aantal threads per block is en dat dit verschilt per architectuur versie. Tot slot roepen we nu onze kernel aan, in zijn simpelste vorm:

```
myKernel<<<numblocks,blocksize>>>(dev_A, dev_B, N);
```

Tussen de driedubbele vishaken vinden we achtereenvolgend het aantal blocks en de blocksize terug. Een optionele derde parameter geeft mee hoeveel bytes shared memory moeten worden gereserveerd per block. In de API documentatie vind je meer hierover. Nu hebben we echter nog een A en B nodig in het “device” geheugen...

1.2 Device memory management

Om de arrays A en B voor onze kernel te alloceren, hebben we toegang nodig tot het geheugen van de GPU (“device memory”, het geheugen van de CPU heet “host memory”). We gebruiken de meest simpele vorm van device memory management waarbij we zelf de geheugentransfers naar en van de GPU moeten programmeren. Dit bestaat uit twee operaties: handmatig alloceren/vrijgeven en kopiëren van en naar device memory. Alloceren doen we met de functie `cudaMalloc(void** devPtr, size_t size)` (vergelijkbaar met de C-functie `malloc()`). Zoals je ziet heeft deze functie twee parameters in plaats van één. Het eerste argument is een pointer naar een zogenaamde *device pointer*. Een device pointer bevat een geheugenadres dat zich in het videogeheugen bevindt en is dus niet af te lopen vanuit de CPU. Toch kunnen we deze pointer opslaan en hem later meegegeven aan een kernel zodat deze het corresponderende videogeheugen kan benaderen. Eerst zullen we dus de arrays A en B alloceren op de GPU:

```
int *dev_A, *dev_B;
if( cudaMalloc( &dev_A, N*N*sizeof(int) ) != cudaSuccess )
    // Handel fout af
// Analoog voor B
```

Merk op dat we in CPU code een device pointer vaak beginnen met `dev_` of `d_` om verwarring te voorkomen. Het aflopen van een device pointer op de CPU resulteert namelijk een onzinnige/illegale memory operatie! We kunnen echter wel lezen en schrijven naar het videogeheugen met behulp van de functie `cudaMemcpy()`. Deze functie kent de volgende parameters:

```
cudaError_t cudaMemcpy( void* dst, const void* src, size_t count,
                        enum cudaMemcpyKind )
```

De eerste drie argumenten lijken op de gewone `memcpy()`, terwijl het vierde argument aan CUDA vertelt in welke "richting" de bewerking plaatsvindt. Er zijn vier mogelijkheden, waarvan we in elk geval `cudaMemcpyHostToDevice` en `cudaMemcpyDeviceToHost` nodig hebben. In het eerste geval is `dst` dus een device pointer en `src` een gewone pointer, bij het tweede vice-versa. Het kopiëren van A naar de GPU zou er dus als volgt uit kunnen zien:

```
int A[N][N]; // Initialiseer op de host
int *dev_A; // Device pointer
if( cudaMemcpy( dev_A, &A[0][0], N*N*sizeof(int), cudaMemcpyHostToDevice )
    != cudaSuccess )
    // Handel fout af
```

Na het uitvoeren van de kernel moet er dus nog een soortgelijke operatie plaatsvinden waarbij het resultaat in array B weer wordt gekopieerd naar het host geheugen. Dergelijke operaties kunnen veel tijd kosten als de dataset groot is en het is dan ook zaak om zorgvuldig te plannen wanneer er het beste gekopieerd kan worden. In het geval van onze image pipeline kunnen we bijvoorbeeld volstaan met de bronafbeelding eenmaal te kopiëren en na afloop het resultaat op te halen. Verder is het ook mogelijk een `cudaMemcpy()` opdracht asynchroon te draaien, zodat de CPU in de tussentijd iets anders kan doen. Zie hiervoor de gelinkte API documentatie.

1.3 Fout afhandeling

Bijna alle `cuda*` functies geven een foutcode in de vorm van `cudaError_t`. Net als bij bijvoorbeeld UNIX system calls is het altijd wenselijk om deze foutcodes te inspecteren en eventueel af te vangen. In de skeleton codes is reeds de macro `CUDA_ASSERT` gedefiniëerd. Door CUDA calls in deze macro te verpakken wordt het programma automatisch afgebroken als er een fout optreedt:

```
CUDA_ASSERT( cudaMemcpy(dev_A, A, N*N*sizeof(int), cudaMemcpyHostToDevice) );
```

2 Optimalisatie

De stream-processor architectuur van de GPU verschilt dermate van de 'reguliere' processor dat de meeste code die 1:1 geport is vanaf de CPU, niet het maximale uit de hardware zal halen. Door aanpassingen aan de CUDA code te maken kunnen we de performance verbeteren.

2.1 Branching

De stream processors werken optimaal als alle threads in een warp precies hetzelfde doen. Bij bijvoorbeeld if-condities is hier niet altijd sprake van: sommige threads moeten wel de if-clause uitvoeren en andere niet. Wat er in feite gebeurt is dat alle threads de if-clause uitvoeren, maar

alleen die threads waarvoor de if-clause “true” was zullen de resultaten van de berekening wegschrijven (vergelijk predicated instructies en masking uit vectorarchitecturen). Dus bij het gebruik van veel if-condities bestaat het gevaar dat sommige threads regelmatig nutteloos werk verrichten wat de efficiëntie niet ten goede komt.

Voor eenvoudige if-statements probeert de CUDA compiler predicated instructies te genereren, zoals hierboven omschreven. Voor ingewikkeldere gevallen is dit niet mogelijk. Deze treden bijvoorbeeld op bij loops waar gebruik wordt gemaakt van een conditionele “break”. Sommige threads springen al uit de loop, andere niet. In zo’n geval moeten de threads met een kort executie pad wachten op de threads met een langer executie pad om zo te synchroniseren. Dit noemt men *divergence* en dit is iets dat je wanneer mogelijk wilt vermijden.

2.2 Shared memory

Naast global memory beschikt elk (thread)block ook over een stukje *shared memory*. Dit is een klein (meestal 16-48 KB), maar supersnel geheugen dat door alle threads binnen hetzelfde block toegankelijk is. Dit shared memory is toegankelijk als één groot array vanuit de kernel en zowel statisch (grootte vooraf bekend) als dynamisch te alloceren. Aangezien de dynamische allocatie in de meeste gevallen het handigste is, volgt daarvan een klein voorbeeld:

```
// Kernel
__global__
void dynamicReverse(int *d, int n)
{
    extern __shared__ int s[]; // deze array komt in shared memory
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}

// Aanroep: let op de derde parameter
dynamicReverse<<<1,n,n*sizeof(int)>>>(d_d, n);
```

Deze kernel draait de inhoud van array `d` van `n` elementen om door eerst de array in `s` te laden (in shared memory) en daarna in omgekeerde volgorde weer naar `d` te schrijven. In de implementatie zijn een aantal zaken belangrijk:

1. Het dynamische shared memory wordt met de `extern __shared__` storage class gedeclareerd en kan slechts één type hebben.
2. In de aanroep naar de kernel wordt tussen de vishaken een derde parameter meegegeven: de grootte die voor het shared memory moet worden gereserveerd in bytes.
3. Door de functie `__syncthreads()` aan te roepen wordt gewacht totdat alle threads in dat block de instructies tot aan de synchronisatie barrière hebben uitgevoerd. De voorbeeldcode gebruikt dit mechanisme om later een *read-after-write* te voorkomen.

⚠ Merk op dat in bovenstaande code iedere thread een data-element kopieert naar het shared memory. Het heeft in het algemeen geen zin (en geen effect) om een pointer in shared memory te plaatsen. Want in dat geval wordt de pointer uit shared memory gelezen (snel), maar het data-element waar de pointer naar wijst wordt nog steeds uit global memory gelezen (langzaam).

De bovenstaande code en nog meer uitleg over shared memory zijn op de onderstaande URL te vinden:
`https://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/`

2.3 Performance Counters

Om erachter te komen waar de performanceproblemen zich voordoen, en welke dat mogelijk zijn, kunnen we gebruik maken van performance counters. We kunnen tijdens het draaien van CUDA programma's verschillende hardware performance counters uitlezen met behulp van de `nvprof` profiler. Met `nvprof` kunnen *events* en *metrics* worden gemeten. *Events* zijn tellingen van bepaalde gebeurtenissen, *metrics* zijn vaak (samenvattende) meetwaarden gebaseerd op *events*.

⚠ Helaas is het niet mogelijk om met `nvprof` events en metrics te meten in de LIACS computerzalen. Dit komt doordat de NVIDIA kernel module sinds 2021 met een speciale optie moet worden geladen om dit mogelijk te maken. Omwille van security staat deze optie standaard uit.

Heb je zelf een NVIDIA GPU ter beschikking? Dan kun je deze optie wel aanzetten en gebruik maken van `nvprof`, zie: `https://developer.nvidia.com/ERR_NVGPUCTRPERM`.

Gebruik je CUDA 11 of later in combinatie met een recente NVIDIA GPU? Dan wordt `nvprof` mogelijk niet meer ondersteund en kun je kijken naar 'NSight Compute', `nv-nsight-cu-cli`.

Met de onderstaande aanroepen kunnen lijsten van meetbare *events* dan wel *metrics* worden opgevraagd:

```
nvprof --query-events
nvprof --query-metrics
```

Een lijst *events* gescheiden door komma's kan worden opgegeven achter de command line parameter `-e`, en voor *metrics* met `-m`. Met het volgende commando worden een *event* en *metric* gemeten:

```
nvprof -e divergent_branch -m achieved_occupancy ./problem1 test2048.png
```

Het is mogelijk dat `nvprof` niet zomaar werkt. Probeer dan de command-line optie `-unified-memory-profiling off` toe te voegen.

2.3.1 Divergent branches

In de eerste subsectie bespraken we al het concept 'divergent branching'. We zien dus dat we de mate waarin dit plaatsvindt kunnen meten met het event `divergent_branch`. Daarnaast kun je ook naar de metric `branch_efficiency` kunnen kijken.

2.3.2 Occupancy

We willen graag dat alle resources binnen de GPU worden gebruikt en dat er geen delen niks aan het doen zijn. Zo kunnen we bijvoorbeeld nagaan dat alle streaming multiprocessors altijd wat te doen hebben en dus bezig zijn met het uitvoeren van een warp. Er zijn verschillende metrics waarnaar kan worden gekeken: `sm_efficiency`, `achieved_occupancy` en `sm_efficiency_instance`. Bekijk met `-query-metrics` wat deze *metrics* precies meten.

Er kunnen verschillende redenen zijn waarom een streaming multiprocessor niet aan een warp kan werken: bijvoorbeeld wanneer alle warps blokkeren op een load. Je zou dan bijvoorbeeld de thread block dimensie aan kunnen passen om juist meer threads (meer warps) toe te voegen, om te kijken of de GPU dan bij blokkerende warps kans ziet om aan een andere warp door te werken.

2.4 Verdere mogelijkheden

Er zijn nog veel meer mogelijkheden, zoals bijvoorbeeld het gebruik van de hardware texture-units. Deze kunnen worden benaderd via *texture references* en *surfaces*. We zullen deze in dit document niet verder bespreken. Als je hier meer over wilt leren, verwijzen we je naar de (zeer) uitgebreide CUDA documentatie:
`https://docs.nvidia.com/cuda/archive/11.8.0/cuda-runtime-api/index.html`