Stofoverzicht CPP

Lucas Riedstra

16 december 2018

Week 1: Architecturen

Concurrency is een programmeerconcept: het idee van een taak opsplitsen in verschillende delen die *mogelijk* tegelijk kunnen worden uitgevoerd. Parallelisme is het daadwerkelijk tegelijkertijd uitvoeren van taken die concurrent zijn. Voor het berekenen van speedup gebruiken we de formule $s = t_{seq}/t_{par}$.

Parallelisme is nu harder nodig dan vroeger: zowel frequentie schaalt niet meer mee als voltage. Dit schalen heet *Dennard schaling*. Ook het schalen van ILP (*Instruction-level parallelism*, gedetecteerd op hardwareniveau, parallele uitvoering van onafhankelijke instructies) is opgehouden. Ook is de complexiteit van ontwerpen toegenomen, en is er steeds meer ruimte nodig op een chip (voor een verdubbeling van performance is een vervierdubbeling van ruimte nodig). Concluderend is groeit de performance van enkele cores niet meer, en parallelisme is tegenwoordig de enige manier om nog significante stijging in performance te bereiken.

Een groot probleem is dat door het grote aantal transistoren, niet alle transistoren tegelijk gebruikt kunnen worden (natuurkundig probleem), dus een deel van de processor blijft ongebruikt. Dit heet *dark silicon*. Ook is speedup beperkt door *Ahmdahl's law*:

$$t_{new} = t_{par}/s + t_{seq},$$

in dit geval is s het aantal cores. Merk op dat Ahmdahl's law aanneemt dat de probleemgrootte constant blijft. Bij *Gustavson's law* wordt de probleemgrootte meegeschaald:

$$s = f_{seq} + n(1 - f_{seq}),$$

waarbij f_{seq} het sequentiële deel van het programma is.

Superscalar processoren voeren onafhankelijke instructies parallel uit. In de hardware wordt de scheduling gedaan, dit is erg complex. *VLIW*-processoren laten de mogelijkheden voor parallelisme identificeren door de compiler i.p.v. door de hardware. Dit kan problemen geven bij nietdeterministisch gedrag van de instructies

Het gebruiken van meerdere cores om meerdere instructiestreams af te handelen heet *MIMD*. Dit is goed schaalbaar (tot de limieten van Dark Silicon). Als meerdere ALU's worden gebruikt om dezelfde instructie op verschillende data toe te passen noemen we dat *SIMD*. In CPU's is hier nog steeds werk van de programmeur voor nodig. Een probleem is op het moment dat er *conditional logic* in een programma zit: alle threads moeten hetzelfde doen, dus beide takken van een if-statement zullen in principe door een thread worden uitgevoerd.

Stalls kunnen verborgen worden met multithreading: als één thread vastzit, kan een andere thread aan de slag. Dit verhoogt *throughput*, ook al kan het *latency* verlagen. Er zijn twee soorten mogelijkheden: *simultaneous*, waarbij op een willekeurig moment slechts één thread tegelijk wordt uitgevoerd, en *interleaved*, waarbij meerdere threads tegelijk kunnen worden uitgevoerd zolang er ruimte voor is. Logischerwijs is interleaved multithreading lastiger te implementeren.

Voor het beheren van geheugen zijn er twee alternatieven: *shared* of *distributed*. Bij shared geheugen kunnen *race*condities ontstaan. Er zijn meerdere modellen voor shared geheugen: bij een UMA-model kunnen alle cores even snel bij elk stuk geheugen komen (excl. caches). UMA is moeilijk schaalbaar - daarom is er ook NUMA, waarbij de kosten voor een read/write verschillen per processor. Dit geeft meer werk voor de programmeur, want die moet nu ook nadenken over lokaliteit. Een belangrijk punt bij shared memory is *cache coherency*: de cache moet up-to-date blijven. De twee alternatieven zijn *write-invalidate* (stuur alleen een update dat de waarde niet klopt) en *write-update* (stuur ook de nieuwe waarde mee). Het standaard model voor distributed geheugen is een *message-passing*-model.

Voor het implementeren voor een architectuur waar cores moeten communiceren zijn er drie punten van belang: welke topologie wordt gebruikt, hoe wordt *routing* gedaan, en hoe wordt *switching* gedaan. Bij een directe topologie heeft elke core een buur, bij een indirecte niet (perse). Voor routing (route bepalen) kan gekozen worden voor *source-based* routing (bij vertrek wordt de route bepaald), of *local routing* (lokaal wordt steeds de volgende stap bepaald). Duidelijk is local routing moeilijker te implementeren, maar heeft wel voordelen. Een standaard routing-voorbeeld is X-Y-routing: ga eerst alle stappen naar links/rechts, en dan alle stappen omhoog/omlaag.

Boodschappen in netwerken worden meestal verdeeld in *packets*. Verschillende soorten switching zijn *store and forward*-switching (wacht per node tot de hele packet gearriveerd is), *wormhole switching* (waarbij een packet nog wordt opgedeeld in kleinere eenheden, *flits*), en *virtual cut-through*-switching (een combinatie van de vorige twee).

Week 2: Multithreading

Parallel programmeren, terminologie

Het programmeren van shared memory-modellen wordt gedaan met *multithreading*. *Virtual shared memory* is een model waarbij geheugen op een bepaalde manier verdeeld is (NUMA), maar het verdeeld zijn van geheugen wordt verborgen door de hardware of software, zodat het voor de programmeur lijkt alsof er shared memory is. Ook hiervoor gebruiken we multithreading. Zoals eerder genoemd is *distributed memory* een ander model, waarbij gebruik wordt gemaakt van *message passing*. Natuurlijk bestaan er ook veel combinaties van shared memory en distributed memory, zogenaamde *hybrids*. Multi-core-CPU's kunnen ook als hybrids worden gezien, omdat caches in principe distributed geheugen zijn. Clusters zijn distributed, supercomputers hybrids.

Natuurlijk heeft elk model zijn eigen problemen: shared memory is moeilijk schaalbaar en heeft last van dataraces, bij distributed memory is het verdelen van data lastig, bij virtual shared memory is er een hoge overhead voor de virtualisatie. Bij hybrids is al moeilijk om te vinden waar data überhaupt is (lokaal of gedeeld).

Een *taak* is een stuk computationeel werk. Een applicatie kan worden gezien als een *directed acyclic graph* (DAG) van taken [1] , waarbij de edges staan voor afhankelijkheden tussen verschillende taken. We kunnen de 'moeilijkheid' van een applicatie meten aan de hand van hoeveel taken er moeten worden uitgevoerd of aan de *span* van de applicatie: hoe diep de bijbehorende DAG is. De span is nuttig omdat knopen op dezelfde diepte *mogelijk* parallel kunnen worden uitgevoerd. *Computatie* is werk dat door taken moet worden uitgevoerd, *communicatie* is informatie-uitwisseling tussen taken. *Synchronisatie* moet plaatsvinden als het werk van een taak van een andere taak afhangt. Hier zijn verschillende methodes voor (locks, barriers, messages).

Scheduling is het bepalen van de volgorde waarin threads worden uitgevorerd, afhankelijk van het programmeermodel is dit bepaald door de gebruiker, door het OS, of door de hardware (bv. in GPU's). Als er veel taken zijn die weinig werk moeten doen, noemen we het werk *fijn*, als weinig taken veel werk doen noemen we dit *grof*. De fijn- of grofheid van het werk heet de *granulariteit*. *Lokaliteit* is het concept van data dichtbij de taken houden die die data nodig hebben. Bij shared memory is dit een belangrijk idee.

Multithreading, terminologie

Een *proces* is een entiteit met eigen *resources*, die geen shared memory gebruikt (maar communiceert d.m.v. message passing), en duur is om te creëren/vernietigen. Het gebruik van meerdere processen heet *multiprocessing*. Een *thread* is een lichtgewicht reeks instructies, deel van een proces, in een shared memory-omgeving, die goedkoop is om te creëren/vernietigen. Het gebruik van meerdere threads per proces heet *multithreading*. Multiprocessing en multithreading zijn manieren om parallelisme in applicaties te verwerken, andere manieren zijn gebruik van SIMD (nauw verwant aan multithreading) en ILP.

Bij multithreading delen de threads de resources van het proces waar ze onder leven, maar kunnen ze daarbinnen wel onafhankelijk taken uitvoeren. Een groot probleem bij multithreading zijn zogenaamde *race conditions*, waarbij meerdere threads tegelijk naar data willen schrijven. Hier zijn twee standaard oplossingen voor: *replicatie*, waarbij je het stuk data lokaal maakt door het per thread te kopiëren, of gebruik van *locks*, waarbij maar één thread tegelijk toegang heeft tot een stuk data. Nadelen van replicatie zijn dat er meer geheugen nodig is en dat er een reductiefase nodig is om de data per thread "samen te nemen". Het gebruiken van locks kan twee vervelende effecten hebben: *deadlock* (alle threads zijn aan het wachten) of *livelock* (de threads gaan heen en weer tussen bepaalde fases maar maken geen vooruitgang). Ook zorgt het gebruiken van locks ervoor dat je code sequentiëler wordt, dit noemen we *serialization*. Een stuk code waar geheugen wordt gelezen/geschreven dat last kan hebben van dataraces heet een *kritiek gebied*.

Voor het schrijven van een multithreaded programma spreken we van *decomposition*, waarbij wordt geprobeerd zo min mogelijk afhankelijkheid tussen de data te creëren en van *clustering*, waarbij de granulariteit wordt gekozen. Er zijn verschillende manieren om parallelisme uit te buiten. Voorbeelden zijn:

- Taken die *embarrasingly parallel*, duidelijk parallel zijn: taken waar totaal geen afhankelijkheid aanwezig is;

[1] Een DAG verschilt van een *tree* omdat een knoop niet perse maar één ouder heeft. Verder kan het als tree beschouwd worden, met bijbehorende termen als diepte.

- Dataparallelisme, waar data verdeeld wordt tussen verschillende cores;
- Taskparallelisme, waarbij taken tegelijk worden uitgevoerd;
- Het *fork-join*-model, waarbij er een *master thread* is, en de taken eerst gesplitst worden in een aantal stukken (fork), en na uitvoering de resultaten weer verzameld worden door de master thread (join).
- Het *master-worker*-model, waarbij er een master is die de scheduling doet en een poel van workers die werk doen. Bij een *push*-model geeft de master werk aan de workers, bij een *push*-model vragen de workers om werk bij de master.
- Het *divide and conquer*-model, waarbij taken op recursieve wijze gesplitst worden, en onderste laag van de taken door workers wordt afgehandeld.
- Het *bulk synchronous*-model, met vier fases die steeds herhaald worden: computatie (lokaal), communicatie (globaal), beslissing (globaal), communicatie (globaal).

OpenMP

OpenMP streeft het ultieme doel van automatische parallelisatie na. De gebruiker hoeft slechts aan te geven waar geparalleliseerd kan worden (met zgn. *pragmas*), en OpenMP doet het voor je. OpenMP volgt het fork-join-model. Het daadwerkelijk genereren van efficiënte, multithreaded code wordt aan de compiler overgelaten. Belangrijke instructies:

- `#pragma omp parallel` begint een stuk code dat parallel zal worden uitgevoerd. De thread die deze pragma tegenkomt zal nieuwe threads maken, zelf meester worden van dat team, en de code in het parallele stuk dupliceren voor alle threads (fork). Aan het einde van het parallele stuk plaatst OpenMP impliciet een barrière, waar de meester verdergaat met sequentiële executie.
- `#pragma omp for` splitst de for-loop die erna moet komen over de threads. Een vereiste is dat het aantal iteraties van tevoren bekend is. Aan het einde van de loop plaats OpenMP een impliciete barrière.

  Achter de pragma kan ook iets komen te staan als `private (x, y)`, wat betekent dat de variabelen `x` en `y` per thread gekopieerd zullen worden. De loop counter wordt automatisch private gemaakt.
- `#pragma omp cricital` zegt dat het stuk erna een kritiek gebied is, en op dat moment zal dat stuk code maar door maximaal één thread tegelijk uitgevoerd worden.
- `reduction (op: var)` maakt `var` private en past aan het eind de operator `op` toe op de `var` van elke thread. Een voorbeeld is het sommeren van counters bij alle threads.
- `if (...)` zal de pragma ervoor alleen uitvoeren als aan de conditie voldaan is. Dit kan nuttig zijn als je bij een bepaalde grootte weet dat parallelisatie nuttig wordt.
- OpenMP ondersteunt verschillende *scheduling techniques*, aan te geven met `schedule (<type>, chunk)`. Hierbij is `type` de techniek die gebruikt moet worden, en `chunk` de grootte van een chunk. Verschillende technieken zijn *static*, waarbij elke thread een (bijna) even

groot block krijgt. Als zelf de chunk size wordt aangegeven, worden chunks van die grootte verdeeld totdat de loop klaar is. Dit heet *block-cyclic*. Een ander type is *dynamic*, waarbij threads een chunk krijgen als ze klaar zijn met hun vorige werk (geeft meer overhead, meer synchronisatie). Bij *guided* scheduling daalt de grootte per iteratie.

Het aantal threads kan in het programma zelf besloten worden. Het beste aantal threads hangt af van de applicatie en van de machine.

Pthreads

Pthreads is een interface voor threads, gedefinieerd als een aantal types en functies in C. De verschillende soorten routines zijn management, mutexes, conditional variables, en synchronisatie. Het maken van threads wordt gedaan met `pthread_create`, het wachten op een thread met `pthread_join`. Lokale variabelen van functies horen bij *private data*, globale variabelen horen bij *shared data*. Het regelen van kritieke gebieden wordt gedaan met *mutexes*, waarmee een stuk code kan worden afgesloten en weer "ontgrendeld" zodat variabelen door één thread tegelijk kunnen worden bekeken. De functies hierbij zijn `pthread_mutex_lock` en `pthread_mutex_unlock`. Deze operaties zijn *atomic* - voor de andere threads lijkt het 'meteen' te gebeuren.

Bij wachten op bepaalde gebeurtenissen worden *condition variables* gebruikt, die de mogelijkheid geven om te wachten met functies als `pthread_cond_wait`. Condition variables geven ook een manier om een signaal te sturen naar een bepaalde thread of naar alle threads met gebruik van `signal` en `broadcast`.

Week 3: MPI en Performance

MPI

MPI berust op een paar simpele ideeën:

- Elke taak voert *dezelfde* functie uit;
- Elke taak mag zien hoeveel taken er zijn en wie hij zelf is;
- Alle taken worden gecreëerd bij het opstarten en vernietigd bij het afsluiten van het programma.

Belangrijke functies zijn:

- `MPI_Init`, de eerste functie die moet worden aangeroepen door een proces, die de MPI-omgeving opstart;
- `MPI_Finalize`, de laatste functie die moet worden aangeroepen door een proces, die de MPI-omgeving termineert;
- `MPI_Abort`, dat de uitvoering van een programma onmiddelijk termineert;
- `MPI_Comm_size` en `MPI_Comm_rank`, die het aantal taken en de ID van deze specifieke taak geven;

- `MPI_Send` en `MPI_Recv`, de standaard blocking send- en receive-functies. Maken gebruik van een *tag* om berichten te kunnen herkennen. De volgorde van berichten wordt alleen bij meerdere berichten van één specifieke taak naar een specifieke andere taak over hetzelfde communicatiekanaal.

Een belangrijk idee bij MPI is het overlappen van communicatie en computatie: communicatie is duur, maar als dat op de achtergrond gebeurt, kunnen we de overhead verbergen. Een van de manieren om dit te doen is door gebruik te maken van *asynchrone communicatie*: door *nonblocking* sends of receives te gebruiken, kan een bepaalde thread doorgaan met werken in plaats van wachten tot een boodschap verstuurd/ontvangen is. Hiervoor zijn ook functies als `MPI_Wait`, die wachten tot een nonblocking operatie gelukt is, en `MPI_test`, die de status van een nonblocking operatie teruggeeft.

Performance

Bij het meten van performance zijn er veel factoren die meespelen.

Hardware We bekijken eerst performance in hardware. Hiervoor zijn verschillende metrieken:

- De clockfrequentie in GHz;
- Het aantal *floating-point* operaties per seconde (GFLOPs);
- De bandbreedte van het geheugen in GB/s, dit kan verschillen voor read- en write-operaties;
- De energieconsumptie in Watt.

Natuurlijk kunnen deze metrieken gecombineerd worden tot nieuwe metrieken als FLOPs/byte. De theoretische piekperformance van hardware wordt gegeven door de formule

$$\text{peak} = \text{chips} \times \text{cores} \times \text{vector width} \times \text{FLOPs/cycle} \times \text{clock frequency}.$$

De throughput wordt gegeven door

$$\text{throughput} = \text{memory clock} \times \text{bits per cycle} \times \text{bus width}.$$

De efficiëntie van energiegebruik kan worden gemeten in FLOPS/Watt.

De piek-performance wordt nooit daadwerkelijk gehaald - het gaat uit van dat alle resources volledig worden gebruik en al het mogelijke parallelisme wordt uitgebuit.

Er zijn twee soorten benchmarking: *microbenchmarking*, waarbij je één specifieke functie wil meten en vergelijken met de theoretische piek, en *benchmarking*, waarbij je een heel platform evalueert en platforms vergelijk. Voor het testen van geheugen moet rekening gehouden worden met de effecten van de cache. Als de effecten van de cache niet moeten worden meegenomen, kunnen bijvoorbeeld random pointers worden gebruikt om zeker te weten dat de cache nutteloos is.

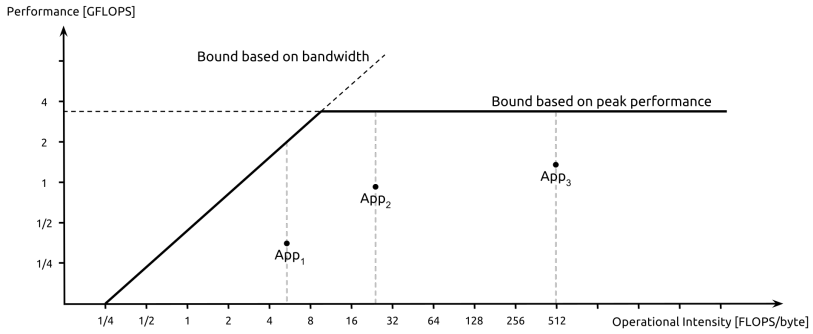
Applicatie Applicaties hebben verschillende fases: sequentiële fases, parallele fases, en communicatiefases. De *wall-clock time* is de tijd die het kost voor de volledige applicatie om te draaien, de *stage execution time* geeft de tijd van één bepaalde fase. Buiten executietijd kan ook gemeten worden in clock cycles. Voor het meten van speedup wordt in principe de *beste* sequentiële tijd gebruikt, maar de beste sequentiële versie bepalen kan lastig zijn. *Superlineaire* speedup is een speedup die hoger is dan het aantal cores, dit is mogelijk door de cache.

We noemen een applicatie *schaalbaar* als het vergroten van de probleemgrootte en het aantal cores de efficiëntie niet verlaagt. Het meten van schaalbaarheid kan op verschillende manieren: bij *strong scaling* wordt de probleemgrootte vastgehouden (in het optimale geval daalt de executietijd), bij *weak scaling* wordt de probleemgrootte per processor vastgehouden (in het optimale geval blijft de executietijd constant). We kunnen een systeem verbeteren door *verticaal* te schalen, door losse nodes sterker te maken (bv. meer cores per CPU), of door *horizontaal* te schalen, door meer nodes toe te voegen. Nog een metriek is *arithmetic intensity*: het aantal floating-point operations dat wordt uitgevoerd per byte aan geheugen die wordt gelezen of geschreven. Deze metriek hoort bij een applicatie. Voor het modelleren van performance kunnen we beginnen met een simpele formule $t = t_{compute} + t_{communicate}$. Het probleem zit in wanneer computatie en communicatie worden overlapt.

Zij nu N_s het aantal sequentiële operaties van een programma en t_s de tijd van één zo'n operatie, en N_p en t_p hetzelfde voor parallel, en N_c en t_c voor een communicatie-operatie. Zij ook n het aantal cores. Dan kunnen we de tijd als volgt voorspellen:

$$t = N_s \cdot t_s + N_p \cdot t_p/n + N_c \cdot t_c.$$

Natuurlijk is dit model erg simplistisch. Er zijn andere modellen zoals het *roofline-model*, waar met een horizontale lijn peak-performance wordt aangegeven en vervolgens daarbij een lijn wordt geplot voor het schalen van bijvoorbeeld de arithmetische intensiteit. Zie plaatje:



Week 4: Mapreduce en Spark

Lecture nodes Adam TODO

Week 5: Grid computing

Lecture nodes Adam TODO

Week 6: GPU's

De markt voor videokaarten is groot, vooral gedreven door gaming. GPU's worden ook steeds meer gebruikt voor totaal andere computationele taken.

Hardware

Een GPU bestaat uit een aantal onderdelen:

- Een CPU die we de *host* noemen.
- Geheugen bij de CPU, ook wel *host memory*.
- Geheugen van de GPU, genaamd *device memory*, met bijbehorende L2-cache.
- Blokken van cores, genaamd *streaming multiprocessors* (SM), met bijbehorende L1-cache. Elke core heeft ook lokaal geheugen (registers).

GPU's zitten op het moederbord via een PCIE-laan, met een theoretische snelheid van 8GB/s.

Het grote verschil tussen een CPU en een GPU is dat een CPU een aantal complexe cores heeft en een GPU heel veel simpele cores. Een CPU moet namelijk "alles" goed kunnen, terwijl een GPU alleen gefocust is op parallelisme op grote schaal. De focus ligt bij CPU's meer op het verlagen van latency (caches + control logic), en bij GPU's meer op het verhogen van throughput (veel registers, multithreading, gedeelde control logic). Het schedulen gebeurt bij GPU's altijd *in hardware*. Op een GPU hebben de host en het device dus verschillende geheugens, en data tussen de twee wordt verplaatst over de PCIe-bus.

Ander geheugen is *texturegeheugen* (read-only geheugen op het device memory met eigen caches), en *constant geheugen* op het device memory dat handmatig kan worden aangepast. Het constante geheugen heeft een eigen speciale cache geoptimaliseerd zodat de *access time* voor elke core gelijk zou moeten zijn.

Het belangrijke geheugen wordt beschermd door *ECC* - een mechanisme dat datacorruptie kan herkennen en tegengaan.

Software

GPU's buiten dataparallelisme uit, dit noemen we SIMT (Single Instruction, Multiple Thread). Alle threads voeren tegelijkertijd dezelfde instructies uit op verschillende dataelementen, en control flow e.d. wordt geregeld door de hardware scheduler. Het creëren van threads, vernietigen van threads, en context switching is op een GPU heel goedkoop.

NVIDIA GPU's worden geprogrammeerd met *CUDA*. Bij CUDA is er een duidelijke bijectie tussen de geprogrammeerde threads en de hardware (CUDA virtualiseert de hardware). Er is een *kernel*, uitgevoerd door elke thread, de threads zitten in *thread blocks*. Threads in hetzelfde block kunnen onderling communiceren, verschillende blocks niet met elkaar. Alle blocks zitten in een *grid*, dat zowel één-, twee-, als driedimensionaal kan zijn. Een thread correspondeert direct met een core, en een thread block correspondeert met een (gevirtualiseerde) SM. Threads worden uitgevoerd in *warps*, en alle threads in een warp voeren dezelfde code uit. Een thread block bestaat uit meerdere warps die los van elkaar worden gescheduled op dezelfde SM.

Omdat in de hardware verschillende geheugenlagen zijn, hebben we die ook in software: de lokale variabelen in kernels corresponderen met de registers van de cores. Dit geheugen raakt dan ook verloren als de kernel stopt. Er is ook *shared*-geheugen, wat gedeeld is binnen een block. Variabelen die worden meegegeven aan de kernel zitten in het 'globale' geheugen, wat correspondeert met device memory. Daarbij hebben we nog constant gebeugen, waar de host niet bij kan maar wat wordt gedefinieerd door de host. Om data te verplaatsen van host-geheugen naar device-geheugen zijn er functies in CUDA. Belangrijk is dat in de code geen onderscheid wordt gemaakt tussen pointers naar host-geheugen of device-geheugen: het dereferencen van een verkeerde pointer zorgt ervoor dat het programma crasht.

Een CUDA-programma bestaat uit twee delen: de device code (de code die elke thread uitvoert) en de host code (die de CPU uitvoert). De cyclus bestaat dan uit het verplaatsen van data van de CPU naar de GPU, het aanroepen en uitvoeren van de kernels en het terugverplaatsen van data van de GPU naar de CPU. De device code wordt geoptimaliseerd door de NVIDIA compiler, de host code door een C/C++-compiler als g++.

Het aanroepen van een kernel gebeurt met `ker <<<grid, block>>>()`, waarbij `grid` de grid size is (die dus ook 2D of 3D kan zijn) en `block` de block size (met gelijke dimensie). Het alloceren en dealloceren van geheugen op de GPU gebeurt met functies als `cudaMalloc`, `cudaMemset`, en `cudaFree`.

Belangrijke punten:

- De volgorde waarin threads in een block gestopt worden, en de volgorde waarin blocks gescheduled worden is niet gedefinieerd. Verschillende blocks kunnen tegelijkertijd draaien, of niet.
- Er bestaat niet zoiets als synchronisatie tussen verschillende blocks (wel binnen een block). Voor het implementeren van een functie als parallelle reductie zullen dus meerdere kernels nodig zijn. In principe geen ramp, want creatie van kernels is niet duur.
- Ook bij een GPU zijn dataraces mogelijk, hiervoor kunnen *atomics* gebruikt worden, die garanderen dat slechts één thread bij een stuk geheugen kan tijdens een operatie. Atomics geven alsnog geen garantie over volgorde waarin operaties worden uitgevoerd, en zijn duur (duurder dan waarde lezen + operatie + wegschrijven).

We weten dat threads allemaal hetzelfde moeten doen. Bij conditional statements (if-else o.i.d.) worden dus in principe allebei de delen uitgevoerd, tenzij geen van de threads in een block een bepaalt deel uitvoert. Dit betekent dat *divergentie* alleen een probleem is als in een block vaak verschillende keuzes worden genomen. Mocht dit een probleem zijn, dan kan het slim zijn een andere mapping voor je threads te kiezen zodat threads die dezelfde beslissing nemen in hetzelfde block terechtkomen.

Als een thread iets leest uit device geheugen, neemt het meteen een heel stuk geheugen mee terug (naar de cache). Hierdoor is het slim om ervoor te zorgen dat memory accesses die dichtbij elkaar zitten ook vlak achter elkaar worden uitgevoerd. Dit idee heet *memory coalescing*. Het verschl met caching is dat de accesses niet perse direct naast elkaar hoeven te zitten, je hebt al voordeel als de accesses één stuk geheugen zitten. De afstand tussen verschillende accesses noemen we *stride* - hoe hoger de stride, hoe meer bandbreedte verspild wordt.

Een *stream* is een reeks operaties die in volgorde worden uitgevoerd. Voor het synchroniseren van alle streams is er de functie `cudaDeviceSynchronize`.

Week 7: Gastcolleges