Användande av GPU med OpenCL för simulering av flockbeteende

JOHN KARLSSON
OCH LARS LÖFQUIST

Examensarbete i datalogi om 6 högskolepoäng
vid Programmet för datateknik
Kungliga Tekniska Högskolan år 2011
Handledare på CSC var Lars Kjelldahl
Examinator var Mads Dam

Kungliga tekniska högskolan
Skolan för datavetenskap och kommunikation

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Redogörelse för samarbete

Ansträngningen för detta projekt som dokumenteras i milstolparna nedan ägde rum under vårterminen 2011. Vi kommunicerade idéutbyten, beslutsfattande och lärdomar under flertalet möten samt över Internet.

- Första brainstorming, research, tidplanering
- Bedömning och val av GPGPU-språk (OpenCL)
- Bedömning och val av Java-bibliotek för användning av OpenCL (JavaCL)
- Research om Boids-algoritmen och implementation i Java & OpenCL
- Research av GPU-arkitektur och dokumentering av relevanta fakta
- Sammanställande av rapport
- Framtagande av presentation och PowerPoint-material för stöd under exjobbskonferensen
- Presentation under exjobbskonferensen

Arbetets större delar såsom implementation av programmen och sammanställande av rapport kunde delas upp, där den största skillnaden i arbetsuppgifter var mellan att sätta upp ramverk och användning av JavaCL (John) och implementation av Boids-algoritmen med regler, anrop och testfall (Lars).

Vi intygar att projektet representeras av den direkta och samverkande ansträngningen vid alla delar av arbetet från varje individ nedan.

John Karlsson	Lars Löfquist

Sammanfattning

Dagens datorer går mot allt fler processorkärnor både i CPU och GPU. Användning av GPU:n för annat än grafikberäkningar benämns med GPGPU. För att utnyttja denna krävs ett GPGPU-språk där denna rapport visar användandet av OpenCL. Detta kopplas samman med Java genom biblioteket JavaCL. I rapporten tas även upp skillnader med att skriva en Boids-algoritm i Java och att utnyttja OpenCL via JavaCL för samma Boids-algoritm, och vilka prestandavinster detta ger vid olika populationsmängder. Testresultaten visar att på den aktuella testmaskinen var renderingshastigheten 512 Boids cirka 2,5 gånger högre för GPU:n än för CPU:n, och vid 1024 Boids är den cirka 10 gånger högre. Slutsaten är att GPU-implementeringen kan hantera ca 4 gånger fler Boids. Utvärdering görs även av hur pass komplicerat det är för en programmerare, utan tidigare erfarenhet av GPGPU-programmering, att implementera JavaCL lösningen. Detta visade sig inte ha någon stor instegströskel för programmeringen, och det gick lätt att skriva kod, tack vare att OpenCL ligger på en lagom abstraktionsnivå.

Abstract

Today's PCs are moving towards more and more multi core processor, both the CPU and GPU. Using the GPU for non graphics calculations is referred to as GPGPU. To realize this requires a GPGPU language. In this report the OpenCL language is used, which is connected to Java through the library JavaCL. This report investigate differences by writing a Boids algorithm in Java and using OpenCL via JavaCL, and the performance gains it offers in various population quantities. Test results show that the current test machine was rendering speed 512 Boids about 2.5 times higher for the GPU than the CPU, and in 1024 Boids is about 10 times higher. The conclusion is that the GPU implementation can handle about four times the number of Boids. We are also evaluating how complicated it is for a programmer, with no previous experience in GPGPU programming, to implement a JavaCL solution. This proved not to be to hard, and it was easy to start produce code, thanks be to OpenCL is at a moderate level of abstraction.

Innehåll

1 Inledning

Sedan 1960 talet har vi kunnat se att prestandan för kiseltekniken har utvecklas i takt med det som Gordon Moore förespådde 1965, att antalet transistorer som får plats på ett chip har fördubblingstiden två år, vilket senare blev känt som Moores lag. Nya tekniker utvecklas ständigt när gamla tekniker är på väg att nå sin gräns. Vi har sett att enkelkärniga processorer har nått en smärtgräns där de förbrukar för mycket energi för att kunna användas i persondatorer, och då speciellt bärbara enheter som är beroende av sin energi ifrån ett batteri.

För att kunna bygga maskiner med mer prestanda utvecklas därför processorer med flera kärnor som kan köras parallellt och utföra flera beräkningar samtidigt. Idag finns det i huvudsak två stycken processorer i en vanlig persondator, en centralprocessor, CPU (Central Processing Unit) och en grafikprocessor, GPU (Graphics Processing Unit). GPU:er har utvecklas till att ha fler parallella kärnor, eftersom de beräkningar som skall göra på denna enhet väldigt ofta är av sådan natur att de är enkla att parallellisera.

Det finns dock många fler problem som lämpar sig för att parallelliseraras och utföras på GPU. Beräkning på GPU:er som inte är direkt grafikrelaterat kallas för GPGPU (General-Purpose computing on Graphics Processing Units).

Detta görs redan idag i datacenter där GPU:er används som beräkningskraft för cirka en tiondel av kostnaden och en tjugondel av elförbrukningen för motsvarande prestanda hos CPU:er. [**1**]

Att använda GPU:n för beräkningar har dock länge varit otillgängligt för programutvecklare eftersom möjligheterna och begränsningarna varit helt beroende av hårdvarans tillverkare och modell. För att abstrahera bort sådana beroenden finns ett antal plattformsoberoende språk som stödjer flera GPU-artitekturer, så kallade GPGPU-språk.

1.1 Frågeställning

Syftet med denna rapport är att titta på skillnader med att skriva en Boids-algoritm i Java och att utnyttja OpenCL via JavaCL för samma Boids-algoritm.

Boids är namnet på en datamodell för att simulera flockbeteendet hos djur. För att simulera stora flockar med väldigt många individer, krävs mycket beräkningskraft. Genom att implementera denna datamodell både för att köras på CPU och på GPU med hjälp av GPGPU, vill vi utreda och jämföra några aspekter. Lämpar sig Boids bra för att parallelliseras?

Hur mycket mer komplicerat är det att implementera denna algoritm för att köras på GPU:n jämfört med att köras på CPU:n?

På vilket sätt är prestandaskillnaden beroende av populationsmängden och hur stor är den?

För implementationen används GPGPU-språket OpenCL som är plattformsoberoende både i hårdvaruaspekten och när det gäller operativsystem. Dessutom utvärderas även interoperabiliteten mellan OpenCL och ett Java-bibliotek kallat JavaCL som används för anrop till OpenCL. Vidare undersöks även tillvägagångssättet för att dela minne mellan OpenCL och OpenGL från JavaCL, vilket kan vara nödvändigt för att undvika prestandaförluster i form utav intensiva dataöverföringar.

1.2 Avgränsningar

Arbetet är avgränsat från att optimera algoritmen på något vis utöver att den delas upp för parallellexekvering i GPGPU-fallet. En ytterligare avgränsning är att undersöka prestandavinsterna på fler än en dator, eftersom en större jämförelse ligger utanför rapportens syfte.

2 Bakgrund

Det finns många fördelar med många kärnor i en och samma processor. Flera små kärnor ger mer prestanda per kiselutrymme för parallella algoritmer. Många små kärnor kan även ge mer finjusterbar energikonsumtion då kärnor som inte används kan vara inaktiva. [**2**] Att använda parallella kretsar gör att strömförbrukningen kan sänkas jämfört med icke parallella kretsar med motsvarande prestanda. [**3**]

Intel som tillverkar och utvecklar processorer har de senaste åren svängt sin fokus från att enbart försöka nå hög prestanda till att utveckla mer energieffektiva enheter. Då stationära datorer var vanligast fanns inte samma behov av strömsnåla processorer. Intels Pentiumplattform var tänkt att kunna användas upp till 10 GHz klockfrekvens. Detta har dock visats sig vara omöjligt på grund av att en sådan processor skulle utveckla enorm värme vilket inte motsvarar de nya kraven. Idag är 8 av 10 datorer som säljs bärbara, och därmed beroende av den begränsade energimängd ett batteri kan ge. Därför har Intel istället gått mot en utveckling där man bland annat utnyttjar flera parallella kärnor för att ge ökad kraft. [**4**]

Just nu syns en utveckling av parallella processorer på två nivåer. CPU:er för arbetsdatorer har mycket komplexa kärnor och har i storleksordningen 2, 4 eller 8 kärnor och klockfrekvenser mellan 1 och 4 GHz. Samtidigt finns sitter det grafikprocessorer (GPU:er) i dessa datorer som vanligtvis har betydligt fler kärnor, vilka dock är betydligt enklare i sin uppbyggnad än CPU:ns. Grafikprocessorerna är optimerade för att hantera grafik. Samtidigt blir GPU:er en allt mer integrerad del i systemutveckling och den hårdvara som används för mer än bara grafikhantering. Detta stärks inte minst av det faktum att AMD, Intel och NVIDIA alla utvecklar kombinationer av CPU- och GPU:er som är integrerade i en och samma enhet. [**5**] [**6**] [**7**]

Traditionella programmeringsspråk bygger till väldigt stor del på en linjär exekvering, där var del av koden utförs efter varandra. Detta stämmer dock inte överens med hur verkliga tillämpningar som många gånger är parallella i sin natur. Då en bild visas är betraktaren intresserad utav att

se hela bilden direkt, det vill säga att alla pixlar som visar bilden tänds samtidigt och inte att den ritas upp pixel för pixel.

Liknande applikationer med parallell natur finns väldigt mycket av. Ett annat exempel är sökning över en stor datamängd. Låt oss jämföra detta med hur en människa skulle söka efter en röd boll i ett bollhav med gröna bollar. Hjärnan kan direkt tolka hela synintrycket och identifiera en röd boll om den ligger någonstans inom synfältet, utan att för den del gå igenom var boll i taget och avgöra färgen på denna. Hjärnan kan arbeta parallellt med den data som ögats tappar och stavar ger. På samma sätt bör en dator hantera en sökning med över en stor datamängd, där datamängden delas upp och processas parallellt. På grund av bristen på denna parallella natur i programmeringsspråk krävs en utveckling för att enkelt och effektivt kunna beskriva för datorer hur de ska kunna arbeta med parallella problem. [**2**]

Även om ett problem har en parallell natur, uppstår svårigheter för programmerare. Många samtidighetsproblem kan uppstå, och miljön gör att det kan uppstå andra buggar än vid sekventiell programmering. Ett vanligt problem är att processer som körs parallellt i slutändan måste vänta på varandra för att till exempel summera ett resultat. [**8**] OpenCL är ett försök att ge ett verktyg åt programmerare att lättare kunna hantera en del av denna problematik.

Att enbart utveckla datorer för att arbeta parallellt är dock inte en lösning på alla prestandaproblem. Det finns en del problem som till sin natur inte låter sig lösas annat än sekventiellt Det går helt enkelt inte att räkna ut nästkommande steg, utan att svaret från det förra steget är känt. Därmed kommer det att finnas en del kod som inte kan accelereras genom att köras parallellt. Amdahl formulerade på 60-talet Amdahls lag. Denna beskriver hur mycket en partiellt parallelliserad process kan köras. Exempelvis så om en process bara kan köras 50% parallellt, kommer den totala minskningen av exekveringstiden aldrig bli mindre än hälften. Kan en process köras parallellt till 95% kommer samma siffra bli 1/20 av tiden, men aldrig mindre än så. [**9**]

3 Teori

3.1 OpenCL

OpenCL är ett öppet programspråk baserat på C99[1] skapat för att möjliggöra parallella beräkningar över heterogena plattformar[2] såsom CPU:er och GPU:er. [**10**] Arkitekturen har en värd→enhetsmodell som innebär att värden, oftast ett vanligt program, har kontroll över en eller flera enheter som exekverar OpenCL-kod.

Miljön, arbetsgrupper och enheter styrs av API:et. Det har även funktioner för att läsa och skriva i delat minne med OpenGL i form av OpenCL-minnesobjekt vilket kan vara en grundförutsättning när stora datamängder behöver överföras. [**10**]

[1] C99, en modern dialekt av programspråket C.

[2] Heterogen plattform, en miljö som består av processorer av olika instruktionsset-arkitekturer (ISA:er).

OpenCL utvecklades ursprungligen av Apple men omformulerades sedan som ett förslag till samarbete mellan flera hårdvaru- och mjukvaruföretag. Detta förslag sändes till Khronos Group som står bakom OpenGL och är ett medlemsfinansierat industrikonsortium, som då startade en arbetsgrupp med bland andra AMD, Apple, IBM, Intel, Nokia, NVIDIA och Umeå universitet som första medlemmar. [**11**] Sex månader senare var den första specifikationen av OpenCL klar.

Khronos är en ideell förening som bygger på filosofin att dess medlemmar säljer mer produkter om de baseras på goda standarder genom att implementera API:et. [**12**] Standarden tas fram i samråd mellan medlemsföretagen och stödet ute på marknaden är således stort.

3.2 JavaCL

JavaCL är ett API-lager ovanpå OpenCL skrivet i Java och består främst av ett objektorienterat API som bibehåller OpenCL-funktionaliteten men som hanterar den på en högre nivå, samt ett lågnivå-API som matchar OpenCL-API:ets funktioner med ett 1:1-förhållande. JavaCL-projektet består även av utility-paket samt Scala[3]-plugins.

Ett centralt verktyg i projektet används för att generera wrappers för OpenCL-kärnkod vilket tillför typsäkerhet mellan objekt i Java och OpenCL. Verktyget ser även till att klasserna dynamiskt läser och kompilerar den associerade OpenCL-koden vid runtime. Tabell 1 visar ett exempel på hur ett anrop sedan görs från Java mot den genererade klassen `Helloworld`, genom ett typsäkert anrop på rad 20.

```
1.  CLContext context = createBestContext();
2.  Helloworld helloWorld = new Helloworld(context);
3.
4.  CLQueue queue = context.createDefaultQueue();
5.
6.  /*
7.   * Initialisera indata för anrop till kärnan.
8.   */
9.  const int dataSize = 1000;
10. FloatBuffer inData;
11. a = NIOUtils.directFloats(dataSize, context.getKernelsDefaultByteOrder());
12. for (int i = 0; i < dataSize; i++)
13.         inData.put(i, i);
14.
15. // Skapa parametrar
16. CLFloatBuffer memIn = context.createFloatBuffer(CLMem.Usage.Input, inData, true);
17. CLFloatBuffer memOut = context.createFloatBuffer(CLMem.Usage.Output, dataSize);
18.
19. // Genomför anrop till kärnan
20. CLEvent kernelCompletion = helloWorld.main(queue, memIn, memOut, new int[] { dataSize }, null);
21.
22. // Resultatet kan nu utläsas från memOut.
23. FloatBuffer output = memOut.read(queue);
```

Tabell 1: Exempelkod för ett enkelt OpenCL-anrop från Java med JavaCL.

[3] Scala är ett multiparadigmprogramspråk som kör i Java-plattformen (JVM).

JavaCL utvecklades redan innan den första OpenCL-implementationen hade kommit ut. Idag finns ett alternativt ramverk kallat JOCL som ett större konsortium står bakom och som bygger på JNI[4] medan JavaCL använder JNA[5]. [**13**] [**14**]

3.3 GPU

3.3.1 Minnesmodell

GPU-minnesmodellen är baserad på tre nivåer. Den första nivån är tillgänglig för samtliga trådar, och är även den enda nivå som värdenheten (CPU) har åtkomst till. Läs- och skrivoperationer på denna nivå betraktas som mycket kostsamma. Nästa nivå är ett delat minne mellan trådblock, och den tredje nivån är ett lokalt minne för en enskild tråd. Ett vanligt programmeringsmönster är att man strukturerar data så att det kan delas upp i respektive trådblocks delade minne för snabbare åtkomst, vilket dock inte är en del av denna rapport. Det så kallade globala minnet i den första nivån är cachat vilket ger en hög effektivitet för R/O[6] data.

3.3.2 Tråddivergens

Vid parallella beräkningar är det fördelaktigt om många trådar exekverar precis samma kod. I arkitekturen grupperas kärnorna i så kallade "warps" av fix storlek där en och samma kod körs. [**15**] Divergens av trådar betyder att ett enkelt kontrolluttryck som "if thread.id = x then f() else h()" försämrar parallellismen eftersom uttrycket isolerar en unik tråd. Det är dock möjligt att utnyttja warp-storleken i kontrolluttryck och hantera hur stor andel av trådarna som divergerar för att få jämn fördelning över alla warp-utrymmen. Hur detta går till kommer ej att diskuteras ytterligare i denna rapport, men det är av betydelse att kontrolluttryck i koden kan påverka parallellismen.

3.4 Boids

För att kunna visualisera flockbetende i datorgrafik föreslog Craig Reynolds 1987 [**16**] en datamodell där han valde att kalla individerna för Boids. Modellens kärnpunkt är att varje boid tilldelas ett antal regler, som efterliknar de i naturen. Trots att populationen består av individer med enkla regler ger detta upphov till komplexa och samordnade rörelsemönster utan att det finns någon ledare. Dessa kan användas för att simulera såväl fågelflockar, insektssvärmar, fiskstim och landlevande djurhjordar.

De grundläggande regler som varje boid får är att:

---

[4] JNI - Java Native Interface är ett gränssnitt för att anropa plattformsspecifik kod från Java och kan användas för att hantera situationer när en applikation inte kan skrivas helt och hållet i Java.

[5] JNA - Java Native Access möjliggör anrop till delade programbibliotek från Java utan att behöva skriva mellanliggande kod. Det ger ett mindre plattformsberoende men vanligtvis lägre prestanda än JNI.

[6] R/O, Read Only

- Undvika kollisioner med andra individer
- Hålla sig nära andra individer
- Anpassa sin egen hastighet till att följa de andra närliggande individerna.
- Anpassa sin egen riktning till att följa de andra närliggande individerna.

Detta görs genom att ge varje boid ett synfält, där den kan läsa av de individer som befinner sig inom detta.

Utifrån dessa tas ett beslut om riktning och hastighet. Ändringen av riktning och hastighet ges dessutom grundläggande fysikaliska egenskaper, som att hastighet och acceleration inte kan överstiga vissa fastställda värden.

4 Metod

För att kunna besvara frågeställningen implementerades en simpel Boids-algoritm [**17**] [**18**] först i Java för att kunna köras i en vanlig virtuell javamaskin på CPU:n. Eftersom Boids-algoritmen består av en loop över populationsmängden, som med fördel kan köras parallellt, gick samma pseudokod att implementera i OpenCL. Loopen över populationsmängden ersattes för att köras parallellt.

För att kunna visualisera implementationerna på skärmen användes en del av demonstrationsapplikationen Particles Demo ifrån JavaCL-biblioteket. [**19**] Detta använder i sin tur JOGL (Java OpenGL) som är ett wrapperbibliotek för att använda OpenGL i Java. Biblioteket användes för att sätta upp en vy med svart bakgrund som ritar partiklar från ett vertexbufferobjekt[7].

När båda implementationerna var klara gjordes en kvantitativ prestandajämförelse med olika höga populationsmängder med java på CPU respektive OpenCL-implementationen på GPU.

De valda populationsmängderna är 2^N där N = {5, 6, 7, …, 12}. Den undre gränsen är 2^5 eftersom mindre populationsmängder vore ett för trivialt problem, och den övre gränsen på 2^{12} eftersom det ligger i överkant för vad CPU och GPU kan arbeta med i normala frekvenser.

Under arbetets gång gjordes en kvalitativ utvärdering av biblioteken för att nyttja GPU:n samt av hur JavaCL och OpenCL fungerar och interopererar.

En prestandaförlust kunde förväntas om dataöverföringen tar för lång tid i relation till själva algoritmen. Därför användes de OpenCL-funktioner specifika för att dela minne med OpenGL, vilka anropades genom JavaCL. Målet med detta var att kvalitativt värdera om ansträngningsnivån var rimlig i JavaCL, eftersom kommunikationen sker genom tre ramverk.

[7] Vertexbufferobjekt, VBO, är en OpenGL-extension som tillåter att data laddas upp på grafikenheten snarare än behöva ligga i systemminnet.

5 Resultat

5.1 Användande av OpenCL

OpenCL visade sig ligga på en lämplig abstraktionsnivå för att ge utvecklare frihet att fokusera på algoritmen och parallellism snarare än detaljer rörande hårdvara och trådhantering. I Tabell 2 visas den kärnkod som användes i Boids-implementationen. Varje partikel identifieras med kärnans id-nummer enligt `get_global_id(0);` vilket inte begränsas av antalet fysiska kärnor på enheten utan istället styrs av den storlek som valts på arbetsgruppen.

```
1.  __kernel void updateParticle(__global float2* velocities, __global float4* particles,
    const float2 dimensions)
2.  {
3.          int id = get_global_id(0);
4.
5.          float4 particle = particles[id];
6.
7.          float2 position = particle.yz;
8.          float2 velocity = velocities[id];
9.          position += velocities[id];
10.
11.         particle.yz = position;
12.         particles[id] = particle;
13.
14.         // Boids söker till närliggande masscentrum.
15.         float2 rule1 = rule1_velocityChange(particles);
16.         // Boids undviker att krocka med varandra.
17.         float2 rule2 = rule2_velocityChange(particles);
18.         // Boids matchar riktning med varandra.
19.         float2 rule3 = rule3_velocityChange(velocities, particles);
20.         // Boids håller sig inom programfönstret.
21.         float2 rule4 = rule4_velocityChange(particles, dimensions);
22.         limitSpeed(velocities, id); // Boidshastigheterna håller en hastighetsgräns.
23.
24.         velocities[id] += rule1 + rule2 + rule3 + rule4;
25. }
```

Tabell 2: Kod som exekveras i GPU-kärnorna.

Reglerna som appliceras påverkar hastighetsvektorerna och följer den pseudokod som beskrivits tidigare och inga försök till optimering har gjorts. Dessa är implementerade i vanlig C99 i samma OpenCL-programkod.

5.2 Interoperabilitet mellan JavaCL och OpenCL

Antaget att JavaCL-biblioteket har använts för att generera Java-classwrappers kunde typsäkerhet uppnås. Den metod som illustrerades i Tabell 1 av förfarandet vid anrop från JavaCL till OpenCL användes även i Boids-implementationen, vilket som nämnt vid runtime dynamiskt läser och kompilerar OpenCL-koden som associeras till klassen.

Dokumentationen för att generera classwrappers visade sig vara begränsad och de exempel som fanns kunde ej köras i en fristående applikation eftersom dess Maven-konfiguration[8] var felaktig.

[8] Maven, http://maven.apache.org/, är ett verktyg utvecklat av Apache som används för att hantera paketering och projektberoenden i Java.

Problemet kunde kringgås genom att sökvägarna till OpenCL-källkodsfilerna angavs explicit i ett manuellt anrop till de lämpliga klasserna enligt:

```
JNAerator.main(new JavaCLGenerator(new JNAeratorConfig()), params);
```

där `params` är av typen `String[]` och måste innehålla följande tre poster:

```
{"-library", "c", "openclsource.cl"}
```

Anropet behövde göras för varje enskild källkodsfil i projektet.

I undersökningen ingick ej mätning av prestandaökningen som gavs från att dela minne mellan OpenCL och OpenGL [**10**], men skillnaden framgick tydligt under implementeringsfasen i enighet med de källor som rapporten diskuterat. Metoderna som krävdes var endast initiering av kontext och buffer enligt Tabell 3.

```
1.  CLContext context = JavaCL.createContextFromCurrentGL();
2.  int vbo = ... ;
3.  CLByteBuffer buffer = context.createBufferFromGLBuffer(usage, vbo);
```

Tabell 3: Exempelkod för att peka ut en buffer i OpenGL.

5.3 Testresultat

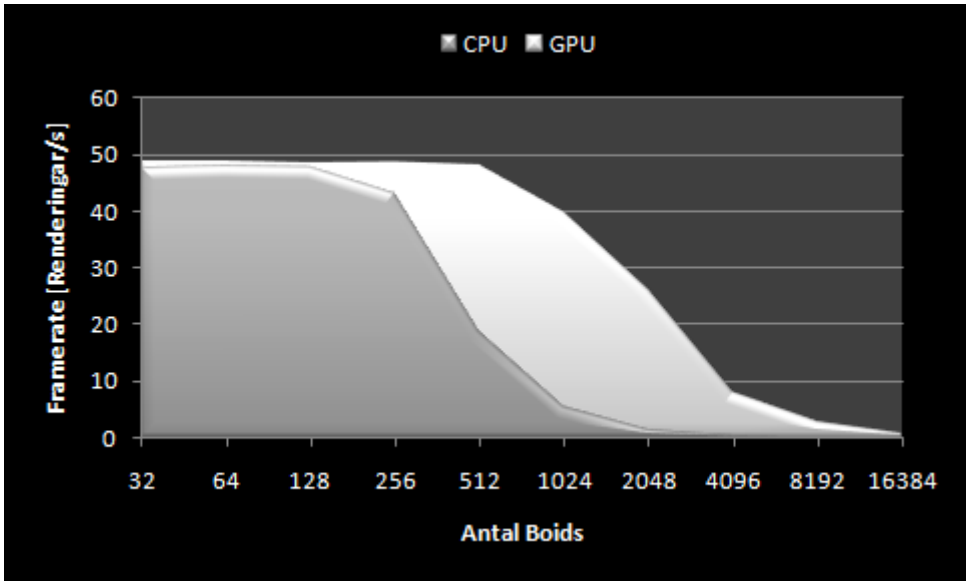
Testresultaten visas i figur 1 där antalet Boids fördubblas i varje steg längs den horisontella axeln. De respektive enheterna som användes i testet var:

- CPU: Intel® Core™ i7-720QM (4@1.6 GHz)
- GPU: NVIDIA Quadro FX 880M (48@550 MHz)

Värdena visar att prestandan sjunker snabbare för CPU:n vars kurva är brantare än den för GPU:n. Detta beror på att CPU:n behöver iterera över samtliga Boids som i sin tur itererar över alla övriga Boids för att ta beslut, vilket förklarar att förhållandet är omvänt proportionellt mot fördubblingshastigheten. GPU:n hanterar detta bättre som endast itererar i en nivå.

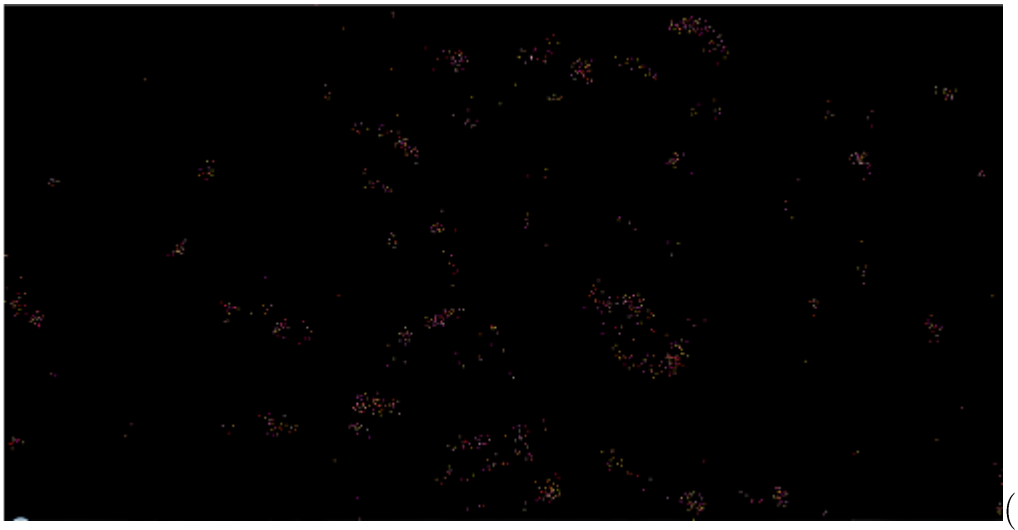
Ett tak sattes vid 50 renderingar per sekund, vilket understigs av CPU:n vid 256 Boids och av GPU:n först vid 1024 Boids.

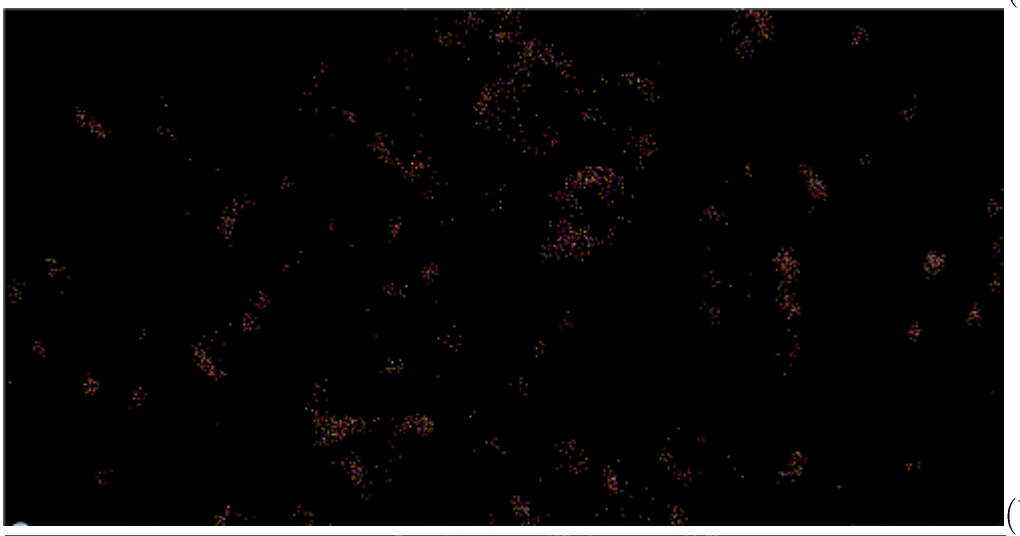
Den vertikala skillnaden mellan två olika värden anger prestandaskillnaden. Vid 512 Boids är renderingshastigheten cirka 2,5 gånger högre för GPU:n än för CPU:n, och vid 1024 Boids är den cirka 10 gånger högre. Den horisontella skillnaden visar antalet ytterligare fördubblingar som GPU klarar gentemot CPU:n för att hålla samma renderingshastighet. Skillnaden är cirka 2 i de flesta fall, vilket innebär att GPU:n i dessa tester normalt klarade av 4 gånger fler Boids än CPU:n.

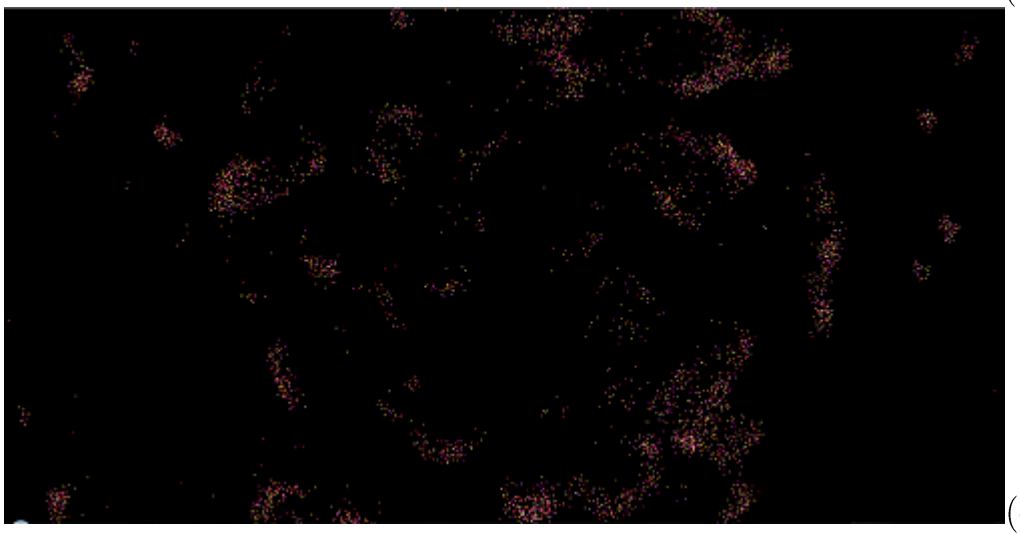


Figur 1: Renderingar per sekund för GPU respektive CPU vid fördubbling av antal Boids.

I Figur 2 visas skärmdumpar från programmet för tre olika mängdantal Boids, där 2048 stycken var det högsta antal som gav en jämn rendering på 27 bilder per sekund som kan utläsas från Figur 1.


(a)


(b)


(c)

Figur 2: Skärmdumpar (1600x900) från programmet i GPU-läge vid 1024 (a), 2048 (b) respektive 4056 (c) antal Boids.

6 Diskussion

Under arbetets gång hörde vi talas om wrappers för OpenCL i många andra språk och det allmänna intresset för OpenCL verkar vara stort. Teorin om GPU-programmering är både välspridd och lättförståelig och det finns många intressanta tekniker till problemlösning. En sådan teknik var till exempel att distribuera datamängden över rätt delar av minnet. Själva programeringsverktygen låg inte heller långt ifrån händerna, utan det upplevdes som att man kunde plocka upp dem och börja arbeta direkt.

Anledningen till att GPGPU-språk inte används i samma utsträckning som andra populära språk, trots dess enkelhet och stora stöd, är möjligen att de bara behövs vid intensiva beräkningar av parallella problemformuleringar. Men det finns inget hinder för att använda tekniken till att snabba upp många enklare problem. Det må vara en utsvävning i ett vanligt hobbyprojekt, men kraften från GPU:n via OpenCL kan idag laddas ner som plugins till de virtuella maskinerna bakom till exempel Scala och Matlab, vilket fungerar som ytterligare beräkningskraft utan att tillföra beroenden eller dialekter till språket. Ett annat exempel på att den teknologiska utvecklingen går mot en tätare samverkan mellan GPU och CPU är som nämnt i rapportens bakgrund att flera hårdvarutillverkare kombinerar de två teknikerna i en och samma enhet. Därför ställer man sig den öppna frågan huruvida utvecklare kommer att behöva lära sig mer om GPU-utveckling, eller om tekniken snart följer med på köpet i de gränssnitt som vi redan använder idag?

Grafikkortet som användes i testerna hade enligt videocardbenchmark.net lite mer än en åttondel av prestandan hos det bästa grafikkortet, medan processorn enligt cpubenchmark.net hade lite mer än en fjärdedel av prestandan hos den bästa processorn. Detta kan tolkas som att GPU i relation till CPU vanligtvis bör ligga ännu högre än de resultat som gavs i rapporten.

Sammanfattningsvis så har vi kommit fram till att Boids-algoitmen lämpar sig mycket väl att parallelliseras och köras på GPU:n. Vi får en ordentlig prestanda ökning, samtidigt som det även ger mer kraft över på CPU:n för andra program och operativsystemet.

Det visade sig även att själva programmeringen inte var så mycket mer komplicerad, även för en programmerare som inte är van att använda GPU:n att implementera algoritmen på denna vilket gör att det inte är någon enormt kunskapssteg att börja använda JavaCl och OpenCl. Dock ställde konfigurationssvårigheter till det något, vilket är något vi hoppas blir bättre allt medan projekten mognar.

7 Referenser

[1] T. D. C. Solutions. (2011, Apr.) Tesla Data Center Solutions. [Online]. http://www.nvidia.com/object/preconfigured-clusters.html [Hämtad 15 mars 2011]

[2] Krste Asanovíc, *The Landscape of Parallel Computing Research: A View from Berkeley*. 2006.

[3] S. S. a. R. W. B. Anantha P. Chandrakasan, *Low-power CMOS digital design*. 1992.

[4] Intervju med Intel-representant av Lars Löfquist 2011-04-11

[5] AMD. (2011, Apr.) The AMD Fusion™ Family of APUs. [Online]. http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx [Hämtad 8 april 2011]

[6] Intel. (2011, Apr.) INTEL® MICROARCHITECTURE CODENAME SANDY BRIDGE. [Online]. http://www.intel.com/technology/architecture-silicon/2ndgen/index.htm [Hämtad 8 april 2011]

[7] NVIDIA. (2011, Jan.) NVIDIA Announces "Project Denver" to Build Custom CPU Cores Based on ARM Architecture, Targeting Personal Computers to Supercomputers. [Online]. http://pressroom.nvidia.com/easyir/customrel.do?easyirid=A0D622CE9F579F09&version=live&releasejsp=release_157&xhtml=true&prid=705184 [Hämtad 9 april 2011]

[8] D. A. a. J. L. H. Patterson, *Computer Organization and Design, Fourth Edition*. Morgan Kaufmann Publishers, 2009.

[9] G. Amdahl, *The validity of the single processor approach to achieving large-scale computing capabilities*. AFIPS Press, 1967.

[10] K. O. W. Group. (2010, Sep.) The OpenCL Specification. [Online]. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf [Hämtad 3 feb 2011]

[11] Khronos. (2008, Jun.) Khronos Launches Heterogeneous Computing Initiative. [Online]. http://www.khronos.org/news/press/releases/khronos_launches_heterogeneous_computing_initiative/ [Hämtad 10 april 2011]

[12] Khronos. (2011) Khronos Membership Overview and FAQ. [Online]. http://www.khronos.org/members/

[13] (2011) Java OpenCL. [Online]. http://jogamp.org/jocl/www/ [Hämtad 10 april 2011]

[14] (2011, Apr.) JavaCL - OpenCL bindings for Java. [Online]. http://code.google.com/p/javacl/ [Hämtad 20 feb 2011]

[15] N. John Nickolls & William J. Dally. (2010, Mar.) HE GPU COMPUTING ERA. [Online]. http://sbel.wisc.edu/Courses/ME964/2011/Literature/onGPUcomputingDally2010.pdf [Hämtad 11 april 2011]

[16] C. W. REYNOLDS, "Flocks, herds and schools: A distributed behavioral model.," in *Conference on Computer Graphics and interactive Techniques.*, 1987.

[17] P. C. (2007) Boids Pseudocode. [Online]. http://www.vergenet.net/~conrad/boids [Hämtad 13 feb 2011]

[18] A. R. SILVA, *Improving Boids Algorithm in GPU using Estimated Self Occlusion, ACM.* 2008.

[19] O. Chafik. (2009) OpenCL bindings for Java - Particles Demo Program. [Online]. http://code.google.com/p/javacl/ [Hämtad 2 april 2011]