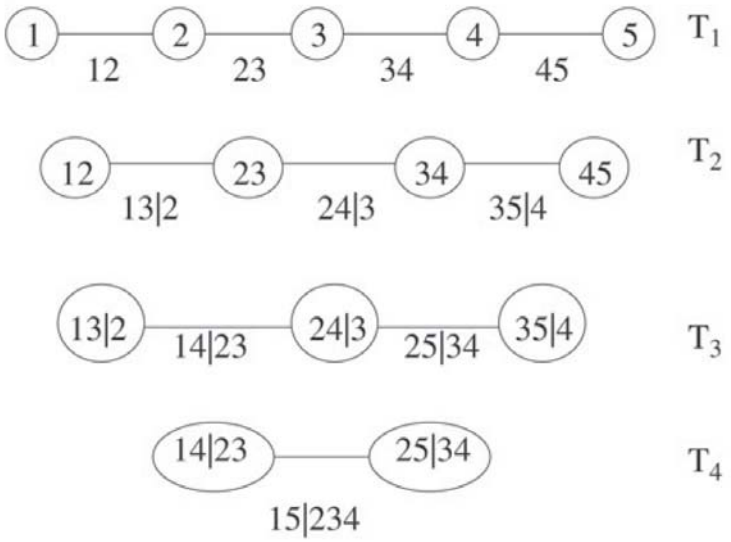
Norsk Regnesentral
NORWEGIAN COMPUTING CENTER

Notat

Parallellisering av simulering fra vines



notatnr SAMBA/18/11

Forfattere Marit Holden
Kjersti Aas

Dato Mai 2011

Norsk Regnesentral

Norsk Regnesentral (NR) er en privat, uavhengig stiftelse som utfører oppdragsforskning for bedrifter og det offentlige i det norske og internasjonale markedet. NR ble etablert i 1952 og har kontorer i Informatikkbygningen ved Universitetet i Oslo. NR er et av Europas største miljøer innen anvendt statistikk. Det jobbes med svært mange forskjellige problemstillinger slik som finansiell risiko, jordobservasjon, estimering av torskebestanden og beskrivelse av geologien i petroleumsreservoarer. NR er også ledende i Norge innen utvalgte deler av informasjons- og kommunikasjonsteknologi. Innen IKT-området har NR innsatsområdene e-inkludering, informasjonssikkerhet og multimedia multikanal. NRs visjon er forskningsresultater som brukes og synes.

Norsk Regnesentral
Norwegian Computing Center
Postboks 114, Blindern
NO-0314 Oslo, Norway

Besøksadresse
Office address
Gaustadalléen 23
NO-0373 Oslo, Norway

Telefon · telephone
(+47) 22 85 25 00
Telefaks · telefax
(+47) 22 69 76 60

Internett · internet
www.nr.no
E-post · e-mail
nr@nr.no

Tittel	**Parallellisering av simulering fra vines**
Forfattere	**Marit Holden**
	Kjersti Aas
Dato	Mai
År	2011
Publikasjonsnummer	SAMBA/18/11

Sammendrag

Modellering av avhengighetsstrukturer er et viktig tema innen finans. Det har etter hvert vist seg at ingen kjent parametrisk fordeling representerer multivariate finansielle data perfekt. Både i den akademiske verdenen og etter hvert også i praksis, har man derfor en stund jobbet med copulas. En copula er en multivariat fordeling med uniforme [0,1] marginal-fordelinger. Ved å kombinere ulike typer copulas med vidt forskjellige marginalfordelinger har man en stor fleksiblitet i den multivariate modelleringen. NR har de siste årene jobbet med enda mer fleksible strukturer, som vi har kalt "par-copula-dekomposisjoner" (også kalt vines). Disse strukturene gir en dekomponering av en multivariat sannsynlighetstetthet i kun bivariate copulas og marginalfordelinger. Selv om par-copula-dekomposisjonene etter hvert har blitt en del benyttet i ulike praktiske anvendelser, er bruken begrenset av at både estimering av parametere i slike strukturer, og simulering av realisasjoner fra dem er beregningsmessig krevende.

I prosjektet beskrevet i denne rapporten var målet å forsøke å få ned beregningstiden på simuleringene ved å parallellisere den korresponderende C-koden. Dette ble først gjort ved bruk av OpenMP, og deretter ved å utnytte beregningskapasiteten til GPU-ene på PC-ens grafikkort. De oppnådde resultatene er veldig lovende. CPU-tidene for OpenMP og GPU-versjonene av C-programmet er på henholdsvis 17 og 7% av CPU-tiden for den opprinnelige versjonen av programmet.

Emneord	Parallellprogrammering, OpenMP, GPU, C-vines, D-vines, par-copula-konstruksjoner, simulering
Målgruppe	
Tilgjengelighet	Åpen
Prosjektnummer	SFI02-Risk 220302
Satsningsfelt	Finans, forsikring og råvaremarkeder
Antall sider	13

Innhold

1 Introduksjon

Modellering av avhengighetsstrukturer er et viktig tema innen finans. Det har etter hvert vist seg at ingen kjent parametrisk fordeling representerer multivariate finansielle data perfekt. Både i den akademiske verdenen og etter hvert også i praksis, har man derfor en stund jobbet med copulas [1]. En copula er en multivariat fordeling med uniforme [0,1] marginal-fordelinger. Ved å kombinere ulike typer copulas med vidt forskjellige marginalfordelinger har man en stor fleksiblitet i den multivariate modelleringen. NR har de siste årene jobbet med enda mer fleksible strukturer, som vi har kalt par-copula-dekomposisjoner [2]. Disse strukturene gir en dekomponering av en multivariat sannsynlighetstetthet i kun bivariate copulas og marginal-fordelinger. En høydimensjonal multivariat fordeling kan dekomponeres på veldig mange ulike måter. I denne rapporten har vi konsentrert oss om to typer av dekomponering, kalt henholdsvis C- og D-vines [3]. Selv om par-copula-dekomposisjoner etter hvert har blitt en del benyttet i ulike praktiske anvendelser, er bruken begrenset av at både estimering av parametere i slike strukturer, og simulering av realisasjoner fra dem, er beregningsmessig veldig krevende.

I prosjektet beskrevet i denne rapporten var målet å forsøke å få ned beregningstiden på simuleringer fra C- og D-vines ved å parallellisere den korresponderende C-koden. Parallelliseringen ble gjort på to ulike måter. Først forsøkte vi å utnytte det at PC-er i dag ofte har flere prosessorer (bærbare PC-er har som regel 2-4 prosessorer, mens stasjonære vanligvis har 4-8). Til dette benyttet vi OpenMP. Se [4] og [5] for en beskrivelse av OpenMP, og [6] for nyttige tips i forbindelse med bruk av OpenMP. Deretter så vi på hvordan vi kunne utnytte beregningskapasiteten til GPU-ene på PC-ens grafikkort ved å benytte programmeringsspråket CUDA C. For en beskrivelse av hvordan man programmerer GPU-ene i CUDA C, se [7] og [8].

Resten av denne rapporten er organisert som følger. I kapittel 2 og 3 beskriver vi hvilke endringer som måtte gjøres i den opprinnelige simuleringskoden i forbindelse med parallelliseringen ved hjelp av henholdsvis OpenMP og GPU, mens kapittel 4 beskriver eksperimenter som ble gjort for å se hvor mye vi kunne klare å redusere CPU-tiden for simuleringene ved hjelp av de parallelliserte programmene.

2 Parallellisering ved hjelp av OpenMP

I dette kapittelet beskriver vi hvilke endringer som måtte gjøres i den opprinnelige simuleringskoden i forbindelse med parallelliseringen ved hjelp av OpenMP.

2.1 Oppretting av et prosjekt i Visual Studio

OpenMP-versjonen av simuleringsprogrammet er utviklet i Visual Studio. Oppretting av et prosjekt i Visual Studio gjøres som vanlig, men ved bruk av OpenMP man må i tillegg gjøre følgende: Velg *Project → Properties → Configuration properties → Command Line* og legg til */openmp* under "*Additional properties*".

2.2 Tilfeldig-tall-generator

Tilfeldig-tall-generatoren som benyttes i dagens versjon av simuleringsprogrammet har en forholdsvis kort periode. Dette kan gi uheldige effekter når man skal parallellisere programmet. I den parallelliserte versjonen av programmet har vi derfor byttet til Mersenne Twister [9], som er en av de nyere tilfeldig-tall-generatorene. Mersenne Twister-algoritmen er implementert i klassen MTRand som har flere funksjoner for å generere tilfeldige tall. Vi bruker bare funksjonene for å generere uavhengige, uniformt fordelte tall mellom 0 og 1. For å generere tilfeldige tall fra andre fordelinger brukes de samme funksjonene som før, bortsett fra at de alle indirekte kaller den nye versjonen av funksjonen for å generere uniformt fordelte tall. Følgende endringer ble gjort i forbindelse med skifte av tilfeldig-tall-generator:

- Har lagt til filen MersenneTwister.h til vinesOpenMP-koden (lastet ned fra fra http://www-personal.umich.edu/~wagnerr/MersenneTwister.html).

- Har byttet ut innhold av lib_ran_unif01 med kall på MTRand-funksjonen rand(). Denne leverer det neste tilfeldige tallet i [0,1].

- Har byttet ut peker til seed med en peker til et objekt av typen MTRand i diverse funksjoner. Seed brukes nå bare til å initialisere MTRand-objektet, og oppdateres ikke.

- Har lagt til "#include MersenneTwister.h" i mange filer.

2.3 Konvertering fra C til C++

Den nye tilfeldig-tall-generatoren, Mersenne Twister, er implementert i C++. For enklest mulig å kunne bruke Mersenne Twister-koden, er alle c-filer, bortsett fra malloc.c, omdøpt til cc-filer. Følgende endring måtte gjøres i forbindelse med dette:

- Har laget filen malloc.h ved å flytte de fire funksjonsdeklarasjonene for minneallokering/deallokering i malloc.c til malloc.h. Har inkludert malloc.h istedenfor de fire funksjonsdeklarasjonene i mange filer. Har også definert egen printError-funksjon i malloc.c.

2.4 Parallellisering

Ved parallellisering av koden ved hjelp av OpenMP er det viktig å huske på å bruke OpenMP-direktivet *threadprivate* for alle globale variable og OpenMP-setningen *private* om hver tråd skal ha sin egen verdi av variable som er deklarert utenfor *#pragma omp parallel*-blokken. Bruk av threadprivate og private skaper lett logiske feil som ikke oppdages av kompilatoren. Dersom det er mulig, er det bedre å unngå bruk av threadprivate og private (og også firstprivate og lastprivate) ved å bruke lokale variable, argumenter til funksjoner og lignende istedenfor (se [6]). Vi gjorde derfor følgende endringer knyttet til dette samtidig med at vi parallelliserte simuleringsprogrammet:

- Har lagt til #include <omp.h>.
- Har lagt til en del "#pragma omp"-setninger for å parallellisere for-løkkene.
- Har flyttet en del variabeldeklarasjoner til blokker lenger inn for å få en kopi av variabelen for hver prosess.
- Har lagt til kode for å generere *n* nokså uavhengige sekvenser av uniformt fordelte tall, der *n* er antall tråder. Sekvensen for tråd nummer *th_id* lages fra seedet som følger:
  mtrand = new MTRand(seed + tr_id);
  Merk at i filen MersenneTwister.h står det at "Not thread safe (unless auto-initialization is avoided and each thread has its own MTRand object)". Det betyr at når vi skal bruke Mersenne Twister i OpenMP-varianten av simuleringsprogrammet, må vi, som vi har gjort, angi start-seedet, og la hver tråd ha sitt eget MTRand-objekt.

3 Parallellisering ved hjelp av GPU

I dette kapittelet beskriver vi hvilke endringer som måtte gjøres i den opprinnelige simuleringskoden i forbindelse med parallelliseringen ved hjelp av GPU. Når vi har utviklet GPU-versjonen har vi forsøkt å ta hensyn til alle rådene som er beskrevet som "High-Priority Recommendations" i *CUDA C Best Practices Guide* [8]. Vi har også fulgt de fleste anbefalingene med lavere prioritet.

For å gjøre PC-en klar for utnyttelse av GPU-ene lastet vi ned *CUDA Toolkit* fra www-siden: http://developer.nvidia.com/object/cuda_3_2_downloads.html. I "*Windows getting started guide*" på denne siden finner man en veiledning for hvordan dette skal gjøres. Vi lastet også ned *GPU Computing SDK code samples*. Disse kodeeksemplene ble lagret på området …*NVIDIA GPU programming SDK* lokalt på PC-en.

3.1 Oppretting av et prosjekt i Visual Studio

GPU-versjonen av simuleringsprogrammet er utviklet i Visual Studio. Da vi opprettet et nytt prosjekt i Visual Studio tok vi utgangspunkt i templaten SimpleTemplate fra …*NVIDIA GPU programming SDK*\C\src[1]. Denne kopien endret vi som beskrevet i *CUDA_SDK_Release _Notes.txt* på området …*NVIDIA GPU programming SDK*\doc\. En av endringene besto naturlig nok i å inkludere kildekoden for det opprinnelige simuleringsprogrammet.

3.2 Tilfeldig-tall-generator

Som beskrevet i kapittel 2.2, benytter vi Mersenne Twister til å generere tilfeldige tall i OpenMP-versjonen av simuleringsprogrammet. Når vi benytter OpenMP trenger vi *N* uavhengige sekvenser med tilfeldige tall, der *N* er antall tråder (vanligvis lik antall CPU-er på PC-en). Siden *N* typisk er et relativt lite tall (vanligvis 4, 8 eller 16), kunne vi benytte en sekvensiell versjon av Mersenne Twister, der vi lager *N* uavhengige sekvenser ved å initialisere tilfeldige-tall-generatoren med *N* ulike seed (*seed, seed+1, … , seed+N-1*). Denne strategien kan man imidlertid ikke bruke i forbindelse med GPU-programmering, fordi man da gjerne har tusenvis av tråder og følgelig trenger tusenvis av uavhengige sekvenser. For GPU-versjonen av simuleringsprogrammet har vi derfor valgt en variant av Mersenne Twister som er spesielt tilpasset GPU-programmering [10], og som sørger for at de *N* sekvensene blir uavhengige av hverandre selv om *N* er stor. Tilfeldig-tall-generatoren for GPU-er, MTGP-src-1.0.2, er lastet ned fra http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MTGP/index.html#footer.

Koden vi lastet ned er for linux (MTGP fins ikke for Windows). Vi måtte derfor tilpasse den litt til Windows. Vi la følgende filer til Visual Studio-prosjektet for GPU-versjonen knyttet til tilfeldig-tall-generatoren: mtgp32-cuda-common.c, mtgp32-cuda-tex.cu, mtgp32dc-param-11213.c, mtgp32-fast.c, mtgp32-fast.h og mtgp-cuda-common.c. Videre, ble følgende endringer ble gjort i disse filene:

- I flere av filene har vi kommentert ut *#include <inttypes.h>*, samt noen (linjer fra) funksjoner som ikke brukes. Som en erstatning for filen inttypes.h har vi lastet ned filen stdint.h fra nettet og lagt den til prosjektet.

[1] For å unngå problemer med linking av koden gjorde vi følgende: "Rebuild the NVIDIA GPU Computing SDK\OpenCL\common\oclUtils.sln". For å kunne bruke atomicAdd og andre atomic- funksjoner gjorde vi følgende: Velg Project->Properties->Configuration Properties->CUDA Runtime API->GPU og sett GPU Architecture(1) til sm_20.

- I mtgp32-cuda-common.c har vi endret litt slik at man kan settet start-seedet selv.

- I mtgp32-fast.h har vi lagt til __ foran inline.

- I mtgp32-cuda-tex.cu har vi fjernet main() og endret temper_single01() som følger: "return int_as_float(r) - 1.0f;" er erstattet av "return 2.0f - int_as_float(r);" for at funksjonen skal returnere verdier i intervaller (0,1] istedenfor i intervallet [0,1).

- I mtgp-cuda-common.c har vi endret avlesingen av *major* og *minor*.

3.3 Parallellisering

For å utnytte ressursene på grafikkortet best mulig, er det viktig å ha mange parallelle tråder. Det får vi til ved å la grupper av tråder ta seg av hver sin simulering. Det beste hadde vært om alle de ønskede simuleringene kunne utføres i parallell. Dette er dessverre ikke mulig pga. minnebegrensninger. Det er imidlertid mulig å utføre en andel av simuleringene i parallell. Vi valgte å la 16384 ($=2^{14}$) simuleringer gå i parallell[2]. Den delen av koden som skal parallelliseres må skrives i CUDA C (den resterende delen av koden kan være uendret). CUDA C inneholder det meste fra programmeringsspråket C, pluss en del tilleggsfunksjonalitet for å kunne utnytte GPU-ene. Følgende endringer er gjort i den opprinnelige simuleringskoden i forbindelse med parallelliseringen ved hjelp av GPU-programmering:

- Filene *tDist.c*, *C-vine.c* og *D-vine.c* er navnet om til henholdsvis *tDist.cu*, *C-vine.cu* og *D-vine.cu*. Dette er gjort fordi funksjoner med CUDA C-kode må ligge i .cu- filer.
- *main.c* er flyttet til *VinesGPU.cu*, og utvidet med noen få setninger for at vi skal kunne bruke GPU-en. Alle andre .cu-filer er inkludert i *VinesGPU.cu*.
- Vi har lagt til en ny fil, *global.h*, med definisjoner og en strukt som trengs i forbindelse med beregningene på GPU-en.
- Alle double-variable er endret fra double til float fordi float er mye mer effektivt enn double ved beregninger på GPU-ene. Vi har sjekket at simuleringsresultatene blir tilstrekkelig like før og etter at vi har gjort denne og andre endringer i koden.
- Mange heltall er gjort om til heltall.0f om det inngår i et float-uttrykk. Tilsvarende er "f" føyd til på slutten av mange desimaltall.
- ContTDist() og InvCondTDist() i tdist.cu returnerer float istedenfor å ha med en ut-variabel.
- I C-vine.cu og D-vine.cu har vi gjort følgende
    - *nuMat* og *rhoMat* blir lagret en-dimensjonalt i "constant memory" slik at vi kun behøver å lagre verdier som benyttes. Dette er nødvendig siden det ikke er så mye "constant memory"ved bruk av GPU.
    - *wVec* er lagret i "global memory" og er mye lengre enn i den opprinnelige versjonen, fordi vi i GPU-versjonen trekker mange uniformt fordelte variable av gangen.
    - *simData* er lagret i en en-dimensjonal array ("global memory") på GPU-en.
    - Det meste av innholdet i de to funksjonene *simulateStudentsTC()* og *simulateStudentsTD()* er flyttet til to kernel-funksjoner[3] som kalles fra henholdsvis *simulateStudentsTC()* og *simulateStudentsTD().*

[2] Antall tråder bør være delelig på 32, som er størrelsen på en "warp" i det grafikkortet vi bruker.

[3] Kernel-funksjoner er en betegnelse på funksjoner som inneholder spesialskrevet GPU-kode. Disse funksjonene kalles fra vanlige C-funksjoner og kjøres på GPU-en.

    - Den viktigste endringen i koden i de to kernel-funksjonene er at matrisen *vMat* er erstattet av en eller to vektorer. Disse vektorene er nå så korte som mulig. Dette er gjort av to grunner: (i) Det er viktig å allokere minst mulig minne i kernel-funksjonene, (ii) vi ønsker å allokere *vMat* lokalt for å unngå bruk av globalt minne som det tar lang tid å aksessere.
- I VinesGPU.cu har vi lagt til funksjoner i) for å (de)allokere og initialisere minne på GPU-en; ii) for å trekke uniformt fordelte tall; og iii) for å gjøre om *nuMat* og *rhoMat* fra to-dimensjonale matriser til en-dimensjonale vektorer.

3.4 Begrensninger

GPU-versjonen av simuleringsprogrammet som er beskrevet i dette dokumentet er en foreløpig versjon i den forstand at vi har lagt to begrensninger på funksjonaliteten sammenlignet med dagens versjon:

- Antall simuleringer må gå opp i 16384.
- Dimensjonen på den multivariate fordelingen man ønsker å simulere fra kan ikke være høyere enn ca 50. Denne begrensningen skyldes blant annet bruk av "constant memory", som det er relativt lite av. Hvis dimensjonen er høyere enn 50 stopper programmet med en melding om at "*Display driver stopped responding and has successfully recovered*"[4]. Det er mulig at nyere versjoner av driveren[5], eller andre grafikkort, vil tillate høyere verdier for dimensjonen.

[4] Maskinen fortsetter å virke med den nyeste driveren for grafikkortet (8.17.12.6779). Med den forrige versjonen (8.17.12.6078) ble skjermen svart, alt stoppet opp og vi måtte restarte PC-en når dette skjedde.

[5] Versjonsnummer for driveren finner vi ved å høyreklikke på My Computer og velge Manage -> System Tools -> Device Manager -> Diplay Adapters. Deretter: Høyreklikk på NVIDIA Quadro 600 og velg Properties -> Driver. I vinduet som kommer opp finner vi blant annet informasjon om driver version. De nyeste driverne kan lastes ned fra NVIDIAs websider.

4 Resultater

I dette kapittelet har vi sjekket hvor mye CPU-tidene blir redusert med de parallelliserte programmene. Vi genererte simuleringer fra seks ulike par-copula konstruksjoner:

- 5-dimensjonal D-vine med alle par-copulaer lik Student's t-copulaen
- 5-dimensjonal C-vine med alle par-copulaer lik Student's t-copulaen
- 25-dimensjonal D-vine med alle par-copulaer lik Student's t-copulaen
- 25-dimensjonal C-vine med alle par-copulaer lik Student's t-copulaen
- 50-dimensjonal D-vine med alle par-copulaer lik Student's t-copulaen
- 50-dimensjonal C-vine med alle par-copulaer lik Student's t-copulaen

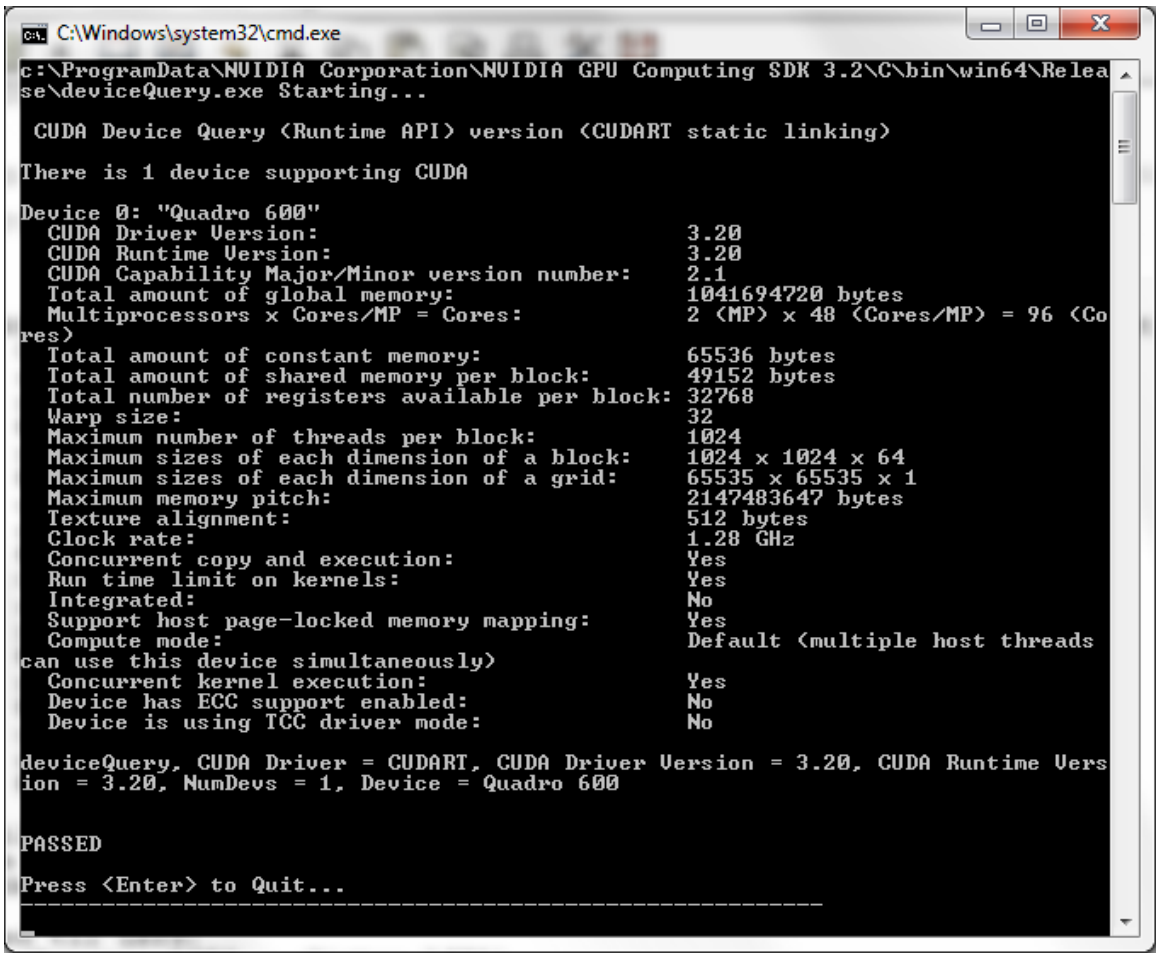
I de 5-dimensjonale strukturene er korrelasjonsparameterne på første nivå satt til henholdsvis 0,8, 0,6, 0,4 og 0,4. For alle par-copulaer på nivå 2-4 er korrelasjonsparameteren satt til 0,2. Frihetsgradsparameterne på første nivå er satt til henholdsvis 8, 6, 4 og 4 og til 10 for de resterende par-copulaene. For de 25- og 50-dimensjonale strukturene er alle korrelasjoner satt til 0,2 og alle frihetsgradsparametere satt til 6.

For hver av de 6 strukturene genererte vi 999 424[6] simuleringer med de tre ulike versjonene av simuleringsprogrammet. Alle versjonene ble kjørt på en PC med 8 prosessorer (Dell Optiplex 980, Intel (R) Core(TM) i7 CPU 860 @ 2,80 GHz 2,79 GHz, 16 GB RAM, 64 bit operativsystem). Maskinen har et grafikkort fra NVIDIA (NVIDIA Quadro 600)[7]. Diverse detaljer om dette kortet er vist i Figur 1, mens Tabell 1 viser CPU-tider for henholdsvis den opprinnelige simuleringskoden, OpenMP-versjonen med en og åtte tråder, og GPU-versjonen. Alle tidene er for 64-bits-versjonen av programmene (kompilert med x64). Vi oppnår omtrent samme prosentvise forbedring av CPU-tiden for 5, 25 og 50 variable. CPU-tidene for OpenMP- og GPU-versjonen er på henholdsvis 17 og 7% av CPU-tiden for den opprinnelige versjonen av programmet.

Tabell 1 viser forøvrig at det opprinnelige simuleringsprogrammet er noe raskere enn OpenMP-versjonen med en tråd. Dette skyldes sannsynligvis at tilfeldig-tall-generatoren som benyttes i dagens versjon av simuleringsprogrammet er ca. 10 ganger raskere enn Mersenne Twister. En annen årsak til at den parallelliserte versjonen er beregningsmessig noe mer krevende, kan være at selve parallelliseringen krever CPU-tid.

[6] Egentlig ønsket vi å benytte 1000000 simuleringer, men fordi antall simuleringer med GPU-versjonen må være delelig med 16384, valgte vi dette antallet i stedet.

[7] Vi vet ikke noe om hvordan CPU-tider endrer seg hvis man benytter et annet grafikkort. Vi vet heller ikke om GPU-versjonen av programmet kan kjøre på et grafikkort med mindre minne enn det *NVIDIA Quadro 600* har.



Figur 1 *Diverse informasjon om grafikkortet NVIDIA Quadro 600.*

Antall variable	5		25		50	
Program	**C-vine**	**D-vine**	**C-vine**	**D-vine**	**C-vine**	**D-vine**
Original-versjon	7,7 (100%)	10,0 (100%)	237 (100%)	340 (100%)	965 (100%)	1401 (100%)
OpenMP-versjon 1t	8,1 (105%)	10,4 (104%)	238 (100%)	344 (101%)	971 (101%)	1418 (101%)
OpenMP-versjon 8t	1,5 (19%)	1,9 (19%)	42 (18%)	59 (17%)	163 (17%)	247 (18%)
GPU-versjon	0,7 (9%)	0,8 (8%)	16 (7%)	23 (7%)	65 (7%)	94 (7%)

Tabell 1 *Sammenligning av CPU-tider for 999 424 simuleringer for de tre versjonene av simuleringsprogrammet (CPU-tid for skriving til fil er ikke inkludert).*

5 Referanser

[1] H. Joe, “Multivariate Models and Dependence Concepts”, Chapman & Hall, London, 1997.

[2] K. Aas, C. Czado, A. Frigessi og H. Bakken, “Pair-copula constructions of multiple dependence, *Insurance: Mathematics and Economics*, 44(2), 182-198.

[3] D. Kurowicka og R. M. Cooke, “Distribution free continuous Bayesian Belief Nets”, *4th Int. Conf. on Mathematical Methods in Reliability Methodology and Practice*, Santa Fe, Mexico, 2004.

[4] OpenMP Reference Sheet for C/C++, http://www.plutospin.com/files/OpenMP_reference.pdf.

[5] OpenMP C and C++ Application Program Interface, http://www.openmp.org/mp-documents/cspec20.pdf.

[6] 32 OpenMP Traps For C++ Developers, http://www.viva64.com/content/articles/parallel-programming/?f=32_OpenMP_traps.html&lang=en&content=parallel-programming#ID0EB2AK.

[7] NVIDIA CUDA C Programming Guide, http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf.

[8] NVIDIA CUDA C Best Practices Guide, http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf.

[9] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator", *ACM Transactions on Modeling and Computer Simulation*, 8(1):3-30, 1998.

[10] M. Saito “A Variant of Mersenne Twister Suitable for Graphic Processors”, Submitted for publication in 2010, http://arxiv.org/abs/1005.4973.