Neues aus Wissenschaft und Lehre

Jahrbuch der Heinrich-Heine-Universität Düsseldorf 2008/2009

HEINRICH HEINE UNIVERSITÄT DÜSSELDORF

d|u|p düsseldorf university press

Jahrbuch der Heinrich-Heine-Universität Düsseldorf

2008/2009

d|u|p

Jahrbuch der Heinrich-Heine-Universität Düsseldorf 2008/2009

Herausgegeben vom Rektor
der Heinrich-Heine-Universität Düsseldorf
Univ.-Prof. Dr. Dr. H. Michael Piper

Konzeption und Redaktion:
Univ.-Prof. em. Dr. Hans Süssmuth

d|u|p

Einbandgestaltung: Monika Uttendorfer
Titelbild: Leben auf dem Campus
Redaktionsassistenz: Georg Stüttgen
Beratung: Friedrich-K. Unterweg
Satz: Friedhelm Sowa, LaTeX
Herstellung: WAZ-Druck GmbH & Co. KG, Duisburg
Gesetzt aus der Adobe Times
ISBN 978-3-940671-33-2

Inhalt

Juristische Fakultät

Partnerschaften der Heinrich-Heine-Universität Düsseldorf

Gesellschaft von Freunden und Förderern der Heinrich-Heine-Universität Düsseldorf e.V.

Forschergruppen an der Heinrich-Heine-Universität Düsseldorf

Beteiligungen an Forschergruppen

Sofja Kovalevskaja-Preisträger

Graduiertenausbildung an der Heinrich-Heine-Universität Düsseldorf

Nachwuchsforschergruppen an der Heinrich-Heine-Universität Düsseldorf

Interdisziplinäre Forscherverbünde an der Heinrich-Heine-Universität Düsseldorf

Kooperation der Heinrich-Heine-Universität Düsseldorf und des Forschungszentrums Jülich

Ausgründungen aus der Heinrich-Heine-Universität Düsseldorf

Zentrale Einrichtungen der Heinrich-Heine-Universität Düsseldorf

Zentrale Universitätsverwaltung

Universitäts- und Landesbibliothek

Zentrum für Informations- und Medientechnologie

Sammlungen in der Universitäts- und Landesbibliothek Düsseldorf

Ausstellungen

Geschichte der Heinrich-Heine-Universität Düsseldorf

Forum Kunst

Chronik der Heinrich-Heine-Universität Düsseldorf

STEPHAN RAUB, JENS ECKEL, REINHOLD EGGER und STEPHAN OLBRICH

Fortschritte in der Forschung durch Hochleistungsrechnen – Kooperation von IT-Service, Informatik und Physik

Zu den Dienstleistungen des Zentrums für Informations- und Medientechnologie im Servicebereich „Hochleistungsrechnen“ gehören nicht nur Maßnahmen zum Betrieb komplexer Rechencluster, sondern auch fortgeschrittene Services zur Unterstützung der effektiven Nutzung. Letztere basieren unter anderem auf dem so genannten Monitoring der Hochleistungsrechner[1], mit dem die Intensität und die Effizienz der verschiedenen Rechenanwendungen systematisch ausgewertet werden. Darüber hinaus beobachtet und bewertet das Zentrum für Informations- und Medientechnologie – in enger Kooperation mit dem Lehrstuhl für IT-Management (Univ.-Prof. Dr. Stephan Olbrich) – kontinuierlich den jeweils aktuellen Stand der Technik der Rechnerarchitekturen.

Im Rahmen eines zweijährigen, von der IT-Firma Bull geförderten Forschungsprojekts wurde – neben anderen Teilprojekten wie zum Beispiel <myJAM/> – zu Projektbeginn die damals innovative GPGPU-Technologie („General-Purpose Computation on Graphics Processing Units“) der Firma nVidia in den Bull-Rechencluster des Zentrums für Informations- und Medientechnologie integriert und im Projektverlauf erprobt. Dazu wurde eine besonders rechenintensive Anwendung im Institut für Theoretische Physik der Mathematisch-Naturwissenschaftlichen Fakultät der Heinrich-Heine-Universität identifiziert. In Kooperation mit dem Lehrstuhl für Festkörperphysik (Univ.-Prof. Dr. Reinhold Egger) wurde das Programm ISPI (iterative Summation des Pfadintegrals)[2] deutlich beschleunigt. Inzwischen können deutlich größere Probleme in immer noch überschaubarer Zeit gelöst werden. Außerdem wird das Zentrum für Informations- und Medientechnologie in der kurzfristig geplanten Erweiterung des zentralen Hochleistungsrechners mehrere Rechenknoten der Firma Bull ergänzen, die mit aktuellen Intel-Quadcore-Prozessoren sowie integrierten GPGPU-Rechenbeschleunigern ausgestattet sind. Damit werden die Erfahrungen aus der prototypischen Pilotierung unmittelbar in den Betrieb und das erweiterte Serviceangebot überführt.

Einleitung

In den letzten Jahren konnten größere Leistungssteigerungen im Bereich des Hochleistungsrechnens insbesondere durch den Einsatz von „Compute-Beschleunigern“ realisiert werden. Einer der Trends ist der Einsatz von Grafikprozessoren (GPU, Graphics Processing Unit) der Firma nVidia, die ursprünglich für leistungsstarke 3-D-Beschleunigerkarten

[1] Vgl. auch „Werkzeuge für Monitoring und Management von Rechenclustern – Anforderungen und Entwicklung des Tools <myJAM/>“ in diesem Jahrbuch, S. 783–791

[2] Vgl. Weiss *et al.* 2008.

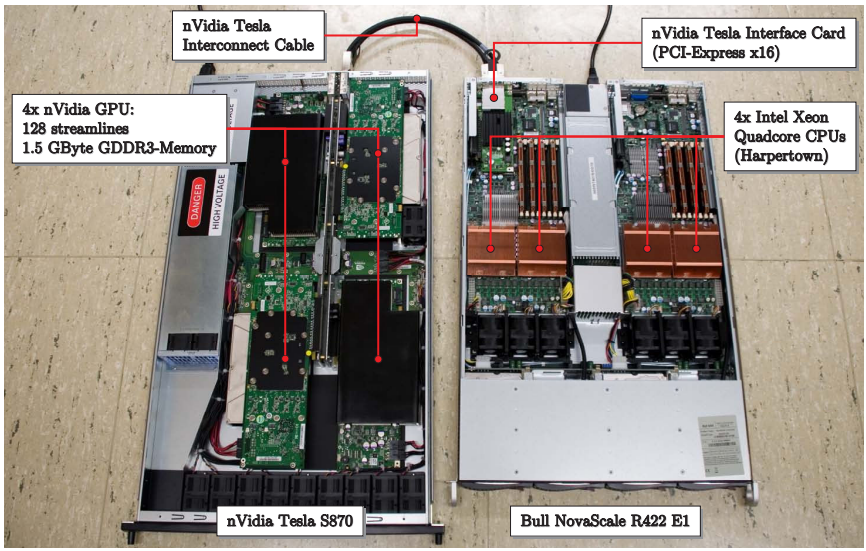

Abb. 1: Innenansicht und Verkabelung der nVidia TESLA

entwickelt wurden. Mit CUDA (Compute Unified Device Architecture) bietet nVidia ein einheitliches API (*Application Programming Interface* – Programmierschnittstelle), um GPGPUs quasi als Koprozessoren einsetzen zu können.

Die TESLA-Serie[3] von nVidia geht noch einen Schritt weiter und bietet die jeweils leistungsstärksten GPUs mit qualitativ hochwertigem Speicher als reine Compute-Beschleuniger an. TESLA-Beschleuniger besitzen erst gar keinen Videoausgang, obwohl es sich gleichwohl um vollwertige Grafikprozessoren handelt, was ihren dedizierten Einsatz als pure Rechenbeschleuniger unterstreicht.

Durch entsprechende hybride Programmieransätze lassen sich GPGPUs auch zur Beschleunigung diverser wissenschaftlicher Simulationen einsetzen. Neben der nativen hybriden Programmierung in CUDA spielen hier der Einsatz spezieller hybrider Softwarebibliotheken sowie von speziellen „CUDA-fähigen“ Compilern eine wichtige Rolle. Dieses soll hier exemplarisch an einer Applikation aus dem Bereich der theoretischen Festkörperphysik gezeigt werden.

Die Heinrich-Heine-Universität war die erste Universität in Deutschland, die eine TESLA S870 (Abb. 1) gekauft und installiert hat. In diesem Beschleuniger stecken gleich vier Grafikprozessoren der G80-Serie, wie man sie von High-End-3-D-Karten kennt. Damit ist die gesamte Tesla etwa 400-mal schneller als heutige Desktop-PCs und kann bereits als ein kleiner Hochleistungsrechner angesehen werden, der aber im Gegensatz zu konventionellen Superrechnern bereits mit „Steckdosenstrom“ auskommt.

[3] Benannt nach Nikola Tesla (1856–1943).

Speed-up für Physiker

In der Molekularelektronik wie auch in nanostrukturierten Bauelementen ist man primär an elektrischen Transportgrößen (wie zum Beispiel der Stromspannungscharakteristik) interessiert. Auf theoretischer Seite ist diese Fragestellung sehr reizvoll (aber auch schwierig), da zum einen die Effekte der Coulombkräfte zwischen Elektronen in solchen Nanostrukturen sehr wichtig sein können und zum anderen das System nicht mehr im thermodynamischen Gleichgewicht ist. Erst in den letzten Jahren wurden Zugänge entwickelt, die eine zuverlässige numerische Analyse dieser Frage erlauben. Ein wichtiger Schritt in diese Richtung wurde mit der Entwicklung iterativer Pfadintegralsimulationen und der Implementierung dieser Methode in dem Programm ISPI am Lehrstuhl für Festkörperphysik des Instituts für Theoretische Physik (Univ.-Prof. Dr. Reinhold Egger) der Heinrich-Heine-Universität getan.

Diese eine Anwendung hat auf dem GAUSS-Cluster über 400.000 CPU-Stunden Rechenzeit (das sind etwas mehr als 45 Jahre) in weniger als sechs Monaten genutzt. Es entstand eine Kooperation zwischen dem Lehrstuhl für IT-Management von Univ.-Prof. Dr. Stephan Olbrich (zugleich Direktor des Zentrums für Informations- und Medientechnologie) und dem Lehrstuhl von Professor Egger mit dem Ziel, ISPI durch den Einsatz der nVidia TESLA zu beschleunigen.

Ignoriert man den kompletten physikalischen Hintergrund von ISPI, so wird die Laufzeit allein von einem Parameter K bestimmt. Grob gesprochen handelt es sich dabei um die Anzahl von Zeitschritten, die für die Berechnung jedes weiteren Zeitschrittes mit berücksichtigt werden. Die Laufzeit für den originalen ISPI-Code wächst extrem schnell mit steigendem K (Abb. 2a). Rechnungen mit $K > 6$ waren nie durchgeführt worden, da für publizierbare Ergebnisse sehr viele Rechnungen mit jedem K erforderlich sind. Für physikalisch interessante Probleme wäre sogar K im Bereich von $K = 12$ notwendig. Das erfordert jedoch Rechnungen, die mit dem alten Code undurchführbar wären.

Wo ist nur die (Rechen-)Zeit geblieben?

Am Anfang jedes Optimierungsprozesses steht verpflichtend ein gewissenhaftes und gründliches Benchmarking und Profiling. Ziel ist es herauszufinden, in welchen Programmteilen die meiste Rechenzeit verbraucht wird und welche Teile überhaupt optimierbar sind. Darüber hinaus soll das Skalierungsverhalten des Codes abgeschätzt werden.

Unser Profiling mit dem Ziel, die Programmteile zu identifizieren, die die Laufzeit maßgeblich bestimmen, lieferte ein klares Ergebnis (Abb. 2b): Bereits ab einem $K = 3$ verbringt das Programm über 70 Prozent in einem einzigen Rechenschritt; ab einem $K = 4$ sind es sogar mehr als 90 Prozent. Somit hatten wir eine eindeutige Programmregion, bei der sich eine Optimierung lohnte.

In dem identifizierten Programmteil werden drei quadratische, komplexwertige Matrizen miteinander multipliziert:

$$\underline{\underline{\mathbf{L}}} = \underline{\underline{\mathbf{B}}} \cdot \underline{\underline{\mathbf{C}}}^{-1} \cdot \underline{\underline{\mathbf{D}}} \tag{11}$$

mit

$$\underline{\underline{\mathbf{L}}}^{n\times n}, \underline{\underline{\mathbf{B}}}^{n\times n}, \underline{\underline{\mathbf{C}}}^{n\times n}, \underline{\underline{\mathbf{D}}}^{n\times n} \in \mathbb{C} \tag{12}$$

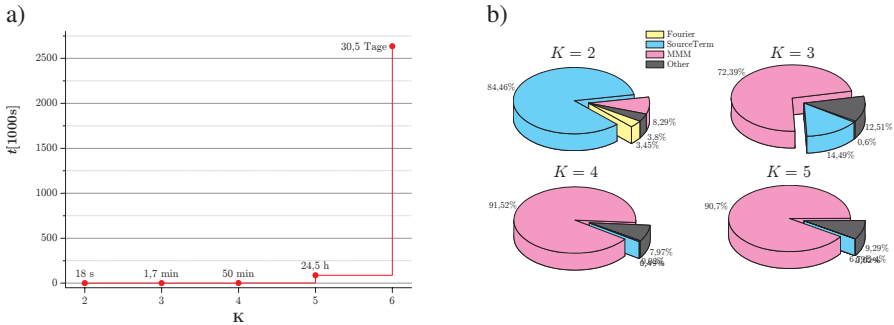

Abb. 2: Skalierung und Profiling des originalen ISPI

und der elementweisen Schreibweise

$$L_{ij} = \sum_{k}^{n} \sum_{l}^{n} b_{ik} \cdot c_{kl} \cdot d_{lj}. \tag{13}$$

```
1       do i = 1, n
2         do j = 1, n
3           do k = 1, n
4             do l = 1, n
5               L(i,j) = L(i,j) + B(i,k)*C(k,l)*D(l,j)
6             end do !. of l
7           end do !. of k
8         end do !. of j
9       end do !. of i
```

Abb. 3: Fortran-Code der Vierfachschleife

Der erste Ansatz zur Berechnung von (13) wäre die exakte Umsetzung der Doppelsumme für alle L_{ij}, was auf eine vierfach geschachtelte Schleife hinausläuft (Abb. 3). Und genauso war es im originalen ISPI auch implementiert. Von einem rein mathematischen Standpunkt aus ist das völlig korrekt. Jedoch skaliert der Rechenaufwand dieser Vierfachschleife mit $\mathcal{O}\left(n^4\right)$ und verursacht eine Menge Multiplikationsoperationen und etliche Zugriffe auf zweidimensionale Felder. Für den Fall, dass alle Matrizen die Dimension 24 mal 24 besitzen (was eher klein ist), sind mit dieser Methode mehr als 660.000 Multiplikationsoperationen und mehr als 1,6 Millionen Feldzugriffe zur Berechnung notwendig (Tab. 1, Spalte „eine Vierfachschleife“).

Obwohl Fortran das Akronym für „FORmula TRANslating system“ ist, sollte man genau das, nämlich eine beliebige Gleichung eins zu eins in ein Programm umsetzen, „nicht“ machen. Als ein *Proof of Concept* ist das völlig akzeptabel, aber wie man an diesem Beispiel gut erkennt, kann der entstandene Code sehr schlecht skalieren und so den Schritt hin zu physikalisch relevanten Systemgrößen versperren. Darüber hinaus verringert sich durch die schiere Anzahl der Multiplikationen die Genauigkeit der errechneten Werte.

	eine Vierfachschleife		zwei Dreifachschleifen	
n	Multiplikationen	Feldzugriffe	Multiplikationen	Feldzugriffe
8	8.192	20.480	1.024	4.096
12	41.472	103.680	3.456	13.824
16	131.072	327.680	8.192	32.768
20	320.000	800.000	16.000	640.000
24	663.552	1.658.880	27.648	110.592

Tab. 1: Vergleich der Anzahl der Multiplikationen und der Feldzugriffe der beiden Schleifenvarianten

Berechnung einer Matrix-Matrix-Matrix-Multiplikation

Eine gängige Praxis, um die Komplexität eines Problems zu verringern, ist, das neue Problem in bereits bekannte und damit verstandene Subprobleme zu zerlegen, für die optimierte Lösungen existieren. Ein offensichtlicher Schritt zur Berechnung von (11) ist die Zerlegung in zwei Matrix-Matrix-Multiplikationen:

$$\begin{aligned} \underline{\underline{\mathbf{L}}} &= \underline{\underline{\mathbf{B}}} \cdot \underbrace{\underline{\underline{\mathbf{C}}}^{-1} \cdot \underline{\underline{\mathbf{D}}}}_{\underline{\underline{\mathbf{T}}}} \\ \underline{\underline{\mathbf{T}}} &= \underline{\underline{\mathbf{C}}}^{-1} \cdot \underline{\underline{\mathbf{D}}} \\ \underline{\underline{\mathbf{L}}} &= \underline{\underline{\mathbf{B}}} \cdot \underline{\underline{\mathbf{T}}} \end{aligned}$$

```
 1       do i = 1, n
 2         do j = 1, n
 3           do k = 1, n
 4               T(i,j) = T(i,j) + C(i,k) * D(k,j)
 5           end do !. of k
 6         end do !. of j
 7       end do !. of i
 8
 9       do i = 1, n
10         do j = 1, n
11           do k = 1, n
12               L(i,j) = L(i,j) + B(i,k) * T(k,j)
13           end do !. of k
14         end do !. of j
15       end do !. of i
```

Abb. 4: Fortran-Code der zwei Dreifachschleifen

Obwohl dieser Schritt wirklich „sehr“ trivial ist, ist der Effekt auf die Recheneffizienz doch dramatisch. Zunächst wird durch diese Zerlegung die Komplexität von $\mathcal{O}\left(n^4\right)$ auf $\mathcal{O}\left(n^3\right)$ reduziert, da zur Berechnung jetzt nur noch zwei dreifach verschachtelte Schleifen nötig sind (Abb. 4). Darüber hinaus verringert sich dadurch sowohl die Anzahl der Multiplikationen als auch die Anzahl der Feldzugriffe dramatisch (siehe Tab. 1, Spalte „zwei Dreifachschleifen“). Das bewirkt nicht nur eine sehr stark verkürzte Rechenzeit, sondern

auch eine Verbesserung der Genauigkeit durch die geringe Anzahl an Fließkomma-Multiplikationen.

Die Matrix-Matrix-Multiplikation ist ein sehr gut verstandenes Problem, für das es hochoptimierte Algorithmen gibt. Ausgehend von (14) ist es sehr einfach, einen beliebigen Algorithmus für die zwei Matrix-Matrix-Multiplikationen zu benutzen. Die *Basic Linear Algebra Subprograms* (BLAS) sind eine Sammlung von Operationen aus dem Bereich der linearen Algebra und Gegenstand ständiger Optimierungen durch Forschung und Industrie. Die Funktion zur generellen Matrix-Matrix-Multiplikation (GEMM) ist eine der am meisten benutzten Operationen aus dieser Sammlung – nicht nur für Benchmarkzwecke, sondern auch in Anwendungen aus der Quantenmechanik – und wird daher seit ihrer Erstimplementierung 1979 ständig optimiert.

Was jedoch, wenn man für einen neuen Ansatz keine Funktion zur Hand hat, die seit 30 Jahren ständig optimiert wird? In diesem Fall ist man auf einen Compiler angewiesen, der einen selbstgeschriebenen Code bestmöglich optimieren kann. Die neueste Generation der PGI-Compiler der Firma The Portland Group[4] geht sogar noch einen Schritt weiter als eine normale Optimierung und generiert direkt CUDA-Code aus gewohnten Hochsprachen wie FORTRAN und C/C++, der dann auf aktuellen Beschleunigern, wie zum Beispiel der nVidia TESLA, ausgeführt werden kann.

```
1 !$acc region
2       do i = 1, n
3         do j = 1, n
4           do k = 1, n
5               T(i,j) = T(i,j) + C(i,k) * D(k,j)
6           end do !. of k
7         end do !. of j
8       end do !. of i
9
10      do i = 1, n
11        do j = 1, n
12          do k = 1, n
13              L(i,j) = L(i,j) + B(i,k) * T(k,j)
14          end do !. of k
15        end do !. of j
16      end do !. of i
17 !$acc end region
```

Abb. 5: Fortran-Code der zwei Dreifachschleifen mit PGI-Compiler-Anweisungen zur Erzeugung von CUDA-Code

Die Handhabung dieser Funktion ist denkbar einfach: Die Programmregion, die auf der TESLA ausgeführt werden soll, wird einfach zwischen den `!$acc`-Anweisungen eingeschlossen – mehr ist nicht nötig (Abb. 5). Der PGI-Compiler bestimmt selbsttätig, welche Arrays vom Speicher des Rechenknotens in den Speicher der TESLA hinein- und anschließend wieder zurückkopiert werden müssen. Außerdem generiert der Compiler alle notwendigen Anweisungen zum Speichermanagement auf der TESLA.

[4] Vgl. http://pgroup.com (29.10.2009).

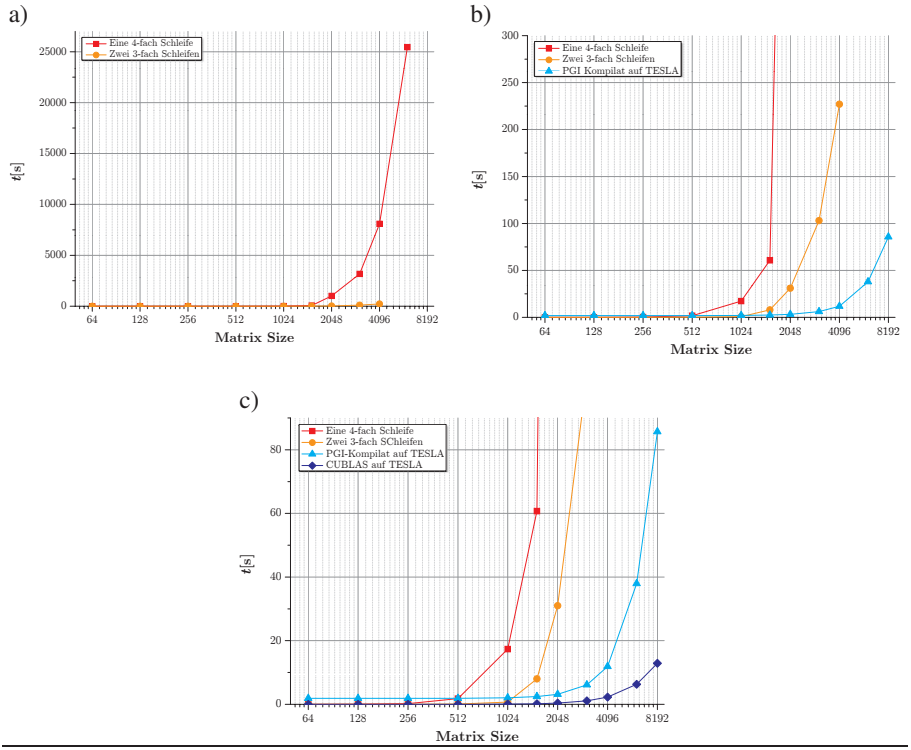

Abb. 6: Vergleich des Skalierungsverhaltens der unterschiedlichen Algorithmen für die Matrix-Matrix-Matrix-Multiplikation

Benchmarking

Das Benchmarking der drei oben beschriebenen Möglichkeiten einer Matrix-Matrix-Matrix-Multiplikation wurde auf einer Bull NovaScale R422-E1 mit Intel Xeon 5462 (Harpertown) Dualcore CPU (2,8 Gigahertz) und angeschlossener nVidia TESLA S870 durchgeführt. Gemessen wurde die Laufzeit der verschiedenen Algorithmen in Abhängigkeit von der Dimension der Matrizen.

Abbildung 6a zeigt den deutlichen Unterschied zwischen der Vierfachschleife und den zwei aufeinanderfolgenden Dreifachschleifen. Zusätzlich ist in Abbildung 6b der Vergleich mit der PGI-compilierten CUDA-Version der zwei Dreifachschleifen dargestellt. Mit nur wenigen Handgriffen kann also unter Verwendung des hochspezialisierten PGI-Compilers aus normalem Fortran-Code ein Programm erzeugt werden, das die Fähigkeiten der TESLA nutzt, um eine dramatische Beschleunigung des Programms zu erreichen.

Der Vollständigkeit halber ist in Abbildung 6c noch die Leistung der CUBLAS-Bibliothek dargestellt, mit der nVidia eine Teilmenge der BLAS-Funktionen CUDA-beschleunigt bereitstellt.

Speicher – „There and Back Again“

Wie eingangs bereits dargestellt, sind die Beschleuniger der Firma nVidia aus deren Grafikkarten hervorgegangen. Dieses Design bedingt, dass der Beschleuniger seinen eigenen Arbeitsspeicher hat und die GPU Zugriff nur allein auf diesen Speicher besitzt. Damit also auf dem Beschleuniger gerechnet werden kann, müssen zunächst die relevanten Daten vom Arbeitsspeicher des Computenode (Host-Memory) in den Arbeitsspeicher des Beschleunigers (Device-Memory) kopiert werden.

Bezogen auf die Matrix-Matrix-Matrix-Multiplikation in ISPI müssen also zunächst die drei Matrizen generiert (wobei $\underline{\underline{\mathbf{C}}}$ obendrein noch invertiert werden muss) und dann in das Device-Memory kopiert werden. Dann kann auf der TESLA die eigentliche Matrix-Matrix-Matrix-Multiplikation erfolgen, um anschließend die Lösung wieder zurück in das Host-Memory zu kopieren. Ein schematischer zeitlicher Ablauf ist in Abbildung 7a dargestellt.

Der Zugriff der GPGPUs auf ihren lokalen Speicher erfolgt zwar mit 76,8 Gigabyte pro Sekunde. Der Datentransfer zwischen Host- und Device-Memory erfolgt jedoch über die PCI-Express(x16)-Schnittstelle der TESLA und ist damit auf $16 \cdot 250$ Megabyte pro Sekunde limitiert. Je nach Szenario kann der Datentransfer damit schnell zum Nadelöhr werden.

Durch spezielle Routinen in den nVidia-Laufzeitbibliotheken ist es jedoch möglich, den Datentransfer „asynchron“ zum Programmfluss durchzuführen, das heißt Daten zu kopieren, während sowohl CPU als auch GPGPU Rechnungen durchführen.

Abbildung 7b zeigt schematisch eine Möglichkeit, sich diesen asynchronen Datentransfer für ISPI zunutze zu machen: Sofort, nachdem die erste Matrix $\underline{\underline{\mathbf{D}}}$ generiert ist, wird ihr Transfer in das Device-Memory angestoßen. Während der Transfer läuft, wird die zweite Matrix $\underline{\underline{\mathbf{C}}}$ generiert, invertiert und anschließend sofort in das Device-Memory transferiert. Sobald auch $\underline{\underline{\mathbf{C}}}^{-1}$ vollständig im Device-Memory ist, kann schon die erste Matrix-Matrix-Multiplikation erfolgen.

Schon direkt nach der Invertierung von $\underline{\underline{\mathbf{C}}}$ kann die letzte Matrix $\underline{\underline{\mathbf{B}}}$ generiert werden, simultan zum Transfer von $\underline{\underline{\mathbf{C}}}^{-1}$ und der anschließenden Matrix-Matrix-Multiplikation auf der TESLA. Ist $\underline{\underline{\mathbf{B}}}$ vollständig generiert, kann diese Matrix sofort in das Device-Memory übertragen werden, unabhängig davon, ob die erste Matrix-Matrix-Multiplikation schon abgeschlossen ist.

Ist sowohl $\underline{\underline{\mathbf{B}}}$ vollständig im Device-Memory und ist die erste Multiplikation abgeschlossen, kann die zweite Multiplikation und danach der Transfer der Lösung $\underline{\underline{\mathbf{L}}}$ zurück in das Host-Memory erfolgen.

Die hier beschriebene Vorgehensweise, bei der die Datengenerierung, der Datentransfer und die Berechnung zeitlich überlappt stattfinden, kann zu einer deutlichen Verkürzung der Programmlaufzeit führen. Die Realisierung solcher so genannten asynchronen Algorithmen ist ein Gegenstand intensiver Forschung und Entwicklung am Lehrstuhl für IT-Management. Allerdings erhöht dieser Ansatz nicht immer die Effizienz; entweder weil die Matrizen zu klein sind, als dass ein solcher Aufwand Beschleunigung bringen würde, oder weil die Matrizen so groß sind, dass nicht alle Daten gleichzeitig in das begrenzte Device-Memory passen würden und völlig andere Wege gegangen werden müssten.

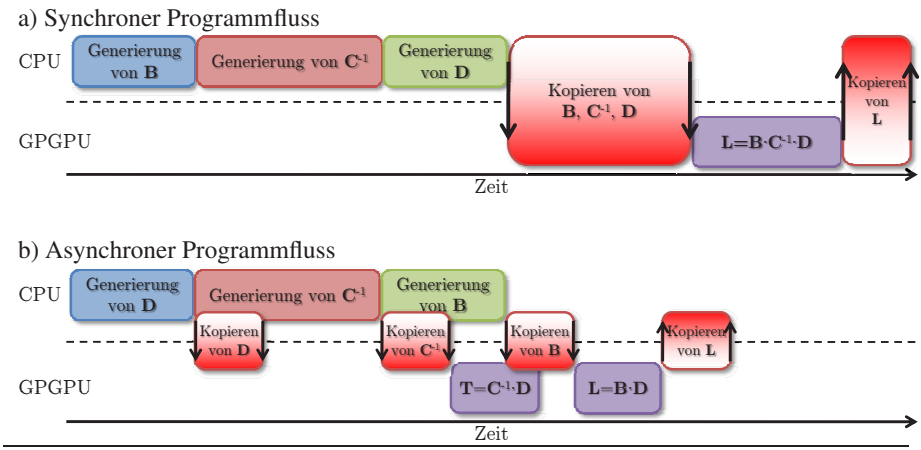

Abb. 7: Gegenüberstellungen von möglichem synchronem und asynchronem Programmfluss bei der Matrix-Matrix-Matrix-Multiplikation

Interessant erscheint hier die Möglichkeit, dass der PGI-Compiler die Befehle zur Verfügung stellt, in Abhängigkeit von erst zur Laufzeit bekannten Größen in den TESLA-Code zu verzweigen oder nicht.

Zusammenfassung und Ausblick

Dieses Beispiel zeigt zwei Dinge: (1.) Bereits bestehende und optimierte Bibliotheken zu nutzen bleibt der primäre Ansatz. Jeder, der auf leistungsstarke Software angewiesen ist, sollte ausführlich nach geeigneten Bibliotheken suchen, bevor er anfängt, selbst eine Methode zu schreiben. (2.) Code sollte nicht einfach von einer Formel in ein Stück Code übersetzt werden. Es kann sich lohnen, sein Problem so umzuformulieren, dass hochperformante Bibliotheken genutzt werden können.

Existieren für einen neuartigen Ansatz keine Bibliotheken, müssen die numerischen Algorithmen selbst entwickelt werden. Durch Compiler wie den PGI-Compiler ist es sehr einfach möglich, aktuelle Compute-Beschleuniger auch dann zu benutzen, wenn noch keine optimierten Bibliotheken zur Verfügung stehen.

Für ISPI konnten wir schon jetzt Erkenntnisse aus unserer Forschung nutzen. Nach der kompletten Umstellung des ISPI-Codes wurde die Matrix-Matrix-Matrix-Multiplikation komplett neu implementiert. Rechnungen mit einem $K = 6$ sind in annehmbarer Rechenzeit durchführbar (für $K = 6$ circa 40 Stunden im Vergleich zu 30 Tagen vorher), obwohl noch längst nicht alle möglichen Beschleunigungen implementiert wurden.

Jedoch wird nun – nach der Optimierung der ehemals geschwindigkeitsbestimmenden Matrix-Matrix-Matrix-Multiplikation – die Rechenzeit von einer anderen Operation dominiert: Mehr als zwei Drittel verbringt das Programm mit der Inversion der Matrix $\underline{\underline{C}}$ und der Berechnung von Determinanten. Beides sind Funktionen, die auf der so genannten LU-

Faktorisierung von Matrizen beruhen. Deren Portierung nach CUDA ist alles andere als trivial[5].

Somit bleiben die geschickte Ausnutzung von asynchronen Datentransfermethoden und die Verlagerung von möglichst vielen Rechenschritten auf die TESLA weiterhin ein spannendes Forschungsgebiet in der Kooperation zwischen der Theoretischen Physik und der Informatik.

Durch die rasante Verbreitung der Beschleunigertechnologie wird sie bald in fast allen Bereichen der *Simulation Sciences* eine wichtige Rolle spielen, auch wenn die Portierung von bestehenden Anwendungen jedes Mal aufs Neue eine Herausforderung darstellt. Doch diese Herausforderung wird mit zunehmender Verfügbarkeit von Entwicklungswerkzeugen wie dem PGI-Compiler oder der von nVidia angekündigten „Nexus"-Entwicklungsumgebung immer besser zu bewältigen sein.

Literatur

VOLKOV, V. und J. DEMMEL (2008). „LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs". Technical Report No. UCB/EECS-2008-49. http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-49.pdf (07.07.2009).

WEISS, S., J. ECKEL, M. THORWART und R. EGGER (2008). „Iterative real-time path integral approach to nonequilibrium quantum transport", *Physical Review B* 77(24), 195316/1–12.

[5] Volkov *et al.* (2008).

ISBN 978-3-940671-33-2