Hardware-unabhängige Beschleunigung von Medizinischer Bildverarbeitung mit OpenCL

Christian Siegl[1], Hannes G. Hofmann[1], Benjamin Keck[1], Marcus Prümmer[1], Joachim Hornegger[1,2]

[1]Lehrstuhl für Mustererkennung, Friedrich-Alexander-Universität Erlangen-Nürnberg;
[2]Erlangen Graduate School in Advanced Optical Technologies (SAOT), Erlangen
hannes.hofmann@cs.fau.de

Kurzfassung. Zur Bewältigung komplexer Berechnungen wird in der medizinischen Bildverarbeitung immer häufiger Spezialhardware eingesetzt. Die Open Computing Language offeriert die Möglichkeit eines gleichzeitig hardware-unabhängigen und performanten Programms. Dies wurde von uns am Beispiel der Bildrekonstruktion untersucht und gezeigt, dass sich mit Hilfe von OpenCL auf CPU-Systemen Leistungssteigerungen einfach erzielen lassen. Des weiteren wird ein hohe Unabhängigkeit der Implementierung von der Hardware erreicht und somit die Nutzung moderner Technologien, wie z.B. Grafikprozessoren, erleichtert. Die Laufzeit unseres Problems konnten wir auf einer Vierkern-CPU von 40 min auf 6,5 min reduzieren. Durch die Verwendung einer Grafikkarte und einfache Optimierungen wurde schließlich eine Laufzeit von 17 s erreicht.

1 Einleitung

In den letzten Jahren hat nicht nur die durchschnittliche Datenmenge medizinischer Datensätze zugenommen, sondern die zum Einsatz kommenden medizinischen Bildverarbeitungalgorithmen wurden auch komplexer. Um den dadurch gestiegenen Rechenaufwand effizient zu bewältigen wird oft Spezialhardware (Multi-core [1], GPUs, FPGAs) eingesetzt. Derartige Spezialhardware bietet zwar eine Vielzahl von Recheneinheiten, jedoch waren bisher hardwarespezifische Befehle (z.B. Intrinsics) oder herstellerabhängige Erweiterungen (z.B. CUDA) notwendig, um deren hohe Leistung zu nutzen.

Die Ende 2008 veröffentlichte Open Computing Language (OpenCL)[1] verspricht eine einheitliche, effiziente und portable Programmierung verschiedener leistungsstarker Hardwarearchitekturen.

In dieser Arbeit wird zuerst OpenCL kurz vorgestellt, um dann am Beispiel der 3D Rekonstruktion von C-arm CT deren Eignung für die medizinische Bildverarbeitung zu untersuchen. Unser Augenmerk liegt dabei auf dem unkomplizierten Umstieg zu OpenCL, der Portierbarkeit des Quelltextes bezüglich der Architektur und der dabei erreichten Leistung.

[1] http://www.khronos.org/opencl/

2 Methoden

Als Beispiel für einen medizinischen Bildverarbeitungsalgorithmus verwenden wir die 3D Rückprojektion von C-arm CT Daten. Die Methode nach Feldkamp et al. [2] findet in der Praxis weite Verbreitung und bedarf der rechenintensiven Verarbeitung großer Datenmengen. Zum Vergleich unterschiedlicher Implementierungen und deren Laufzeitmessung verwenden wir das von Rohkohl *et al.* vorgestellte RabbitCT Framework [3].

2.1 RabbitCT – Datensatz und Benchmark

Bei RabbitCT handelt es sich um ein Framework zum Vergleich verschiedener, optimierter Implementierungen der 3D Rückprojektion. Dazu wird ein Mess- und Auswertungsprogramm, eine Beispiel-Implementierung, und ein standardisierter Datensatz mit Geometrieinformationen öffentlich zur Verfügung gestellt[2]. Dies erlaubt einen fundierten Vergleich der Laufzeit verschiedener Implementierungen, sowie der von ihnen erreichten Bildqualität.

Die 496 Projektionsbilder der Größe 1240×960 des RabbitCT Datensatzes sind bereits vorverarbeitet und gefiltert. Die Aufnahmegeometrie wird in Form von Projektionsmatrizen bereitgestellt. Für unseren Vergleich verwendeten wir eine Volumenauflösung (x,y,z) von 512^3 Volumenelementen.

2.2 OpenCL – Open Computing Language

OpenCL ist ein offener Standard zur Programmierung heterogener Systeme. Ziel ist die Bereitstellung einer einheitlichen Programmierschnittstelle für effiziente und portable Programme. Die drei Kernkonzepte von OpenCL werden im folgenden vorgestellt. Für eine ausführliche Einführung in OpenCL verweisen wir auf die Webseite der Khronos-Group[1], welche den Standard verwaltet, sowie auf Tutorials und Anleitungen von NVIDIA und AMD.

Das **Plattformmodell** besteht aus einem *Host*, der Verbindung zu einem oder mehreren OpenCL *Devices* aufnimmt. Ein OpenCL *Device* ist unterteilt in eine oder mehrere *Compute Units*. Diese sind weiter unterteilt in ein oder mehrere *Processing Elements* auf denen die tatsächliche Programmausführung stattfindet. Dabei sind *Host* und *Device* logisch voneinander getrennt, was die Portabilität sicherstellt.

Das OpenCL **Ausführungsmodell** legt die Ausführung der *Kernel* fest. *Kernel* sind Programme, die in einem C-Dialekt programmiert und auf dem OpenCL *Device* parallel ausgeführt werden. Wenn ein *Kernel* vom *Host*-programm zur Ausführung an das OpenCL *Device* übergeben wird, wird ein Indexraum erzeugt. Für jede Stelle in diesem Indexraum wird eine Instanz dieses *Kernels* ausgeführt. Diese *Kernel*-Instanz wird *Workitem* genannt und ist identifizierbar über die Position im Indexraum. *Workitems* werden in Gruppen zu *Workgroups* zusammengefasst.

[2] http://www.rabbitct.com/

Das OpenCL **Speichermodell** basiert wie die beiden anderen Modelle auf der Trennung von *Host* und *Device*. Dabei gibt es verschiedene Arten von Speicher, die unterschiedliche Zugriffseigenschaften und Latenzen aufweisen.

3 Implementierung und Evaluierung

Neben Hauptprozessoren wurden aufgrund ihrer hohen Rechenleistung auch Grafikkarten untersucht.

3.1 Berechnung auf Hauptprozessoren (CPUs)

Für die Laufzeitmessung der CPU-basierten Implementierungen wurde ein Rechner mit einem Intel Core2 Extreme X9650 Vierkern-Prozessor mit 3.0 GHz Taktrate verwendet. Als Ausgangspunkt diente die zu RabbitCT gehörende Referenzimplementierung. Hierbei handelt es sich um eine einfache Umsetzung der mathematischen Rechenvorschrift. Die Implementierung wird lediglich mit dem Compiler übersetzt und von diesem optimiert. Dadurch wird bei der Ausführung nur ein Prozess verwendet, die Laufzeit liegt bei ca. 42 min (siehe Abb. 1).

Da 3 der 4 Prozessorkerne durch den einzelnen Prozess nicht verwendet werden, wurde der Algorithmus als Erstes parallelisiert und auf mehere Prozesse aufgeteilt. Ein einfach zu verwendendes, weit verbreitetes Werkzeug hierfür ist die Open Multi-Processing (OpenMP) Erweiterung, z.B. für C/C++. Diese verwenden wir, um auf einfache Weise alle 4 Kerne auszunutzen. Mit Hilfe von OpenMP wurde die äußere Schleife (z-Richtung des Volumens) des Algorithmus in mehrere Blöcke aufgeteilt, die nun jeweils von einem anderen Prozessor bearbeitet werden. Dies erbringt einen Geschwindigkeitsgewinn um Faktor 4,3 auf eine Laufzeit knapp unter 10 min.

Ebenfalls auf Basis der Referenzimplementierung betrachten wir nun eine OpenCL Implementierung auf der CPU. Als Framework wird hierbei das ATI Stream SDK 2.1, welches OpenCL 1.0 unterstützt, verwendet. Jeder OpenCL *Kernel* bearbeitet eine (oder mehrere) x-y-Schicht(en) des Volumens. Allein der Einsatz von OpenCL bringt gegenüber OpenMP eine zusätzliche Geschwindigkeitssteigerung um Faktor 1,5 und eine Laufzeit von ca. 6,5 min. Der zusätzliche Geschwindigkeitsgewinn lässt sich dadurch erklären, dass der OpenCL Compiler die Vektoreinheiten besser ausnutzt als sein OpenMP Pendant.

3.2 Verwendung von Grafikprozessoren (GPUs)

Zur weiteren Beschleunigung greifen wir auf die Leistung einer Grafikkarte zurück. Für die Messungen wurde eine NVIDIA QuadroFX 5600 benutzt. Dabei wurde der CUDA 3.1 Treiber verwendet, der OpenCL 1.1 unterstützt.

Die auf der CPU entwickelte OpenCL Implementierung kann durch Änderung des OpenCL *Device* im *Host*-Programm ohne weitere Anpassungen auf der Grafikkarte ausgeführt werden, schöpft aber nicht deren volle Leistung aus. Grafikkarten bieten ein Vielfaches an Recheneinheiten und erfordern deshalb einen

höheren Grad an Parallelisierung. Daher wird der Algorithmus nicht nur in z-Richtung aufgeteilt, sondern über alle Volumenelemente. Diese GPU Implementierung mit OpenCL kann im Vergleich zur CPU einen Geschwindigkeitsgewinn um Faktor 1,3 auf ca. 5 min verbuchen. Die Performanz wurde zwar verbessert, allerdings fällt der Gewinn geringer als erwartet aus.

Ursache hierfür ist die spezifische Architektur von Grafikkarten. Zwar bieten GPUs eine enorme theoretische Rechenleistung, intensiver Zugriff auf den Grafikspeicher der Karte stellt allerdings einen Engpass dar. Um diesen Einfluss zu verringern wenden wir die folgenden Optimierungen für Grafikkarten an.

Zuerst haben wir den sogenannten *coalesced* Speicherzugriff verwendet. Greifen alle *Kernel* einer *Workgroup* gleichzeitig in einem bestimmten seriellen Muster auf den Speicher zu, dann kann die effektive Speicherbandbreite deutlich erhöht werden. Bei einer Transaktion kann dann ein größerer Datenblock, anstatt mehrerer Kleiner, zum bzw. vom Speicher übertragen werden.

Um diese Technik in unserem Fall anzuwenden, gehen wir von der beschriebenen 3D-Aufteilung des Problems zurück zu einer 2D-Aufteilung, parallelisiert über die x-z-Ebene. Wichtig ist hierbei, dass jeder *Kernel* nun eine Zeile des Volumens in y-Richtung und jede *Workgroup* einen Teil einer Zeile in x-Richtung bearbeitet. Durch diese Art der Paralleliserung wird sichergestellt, dass die *Kernel* innerhalb einer *Workgroup* sequentiell auf den Volumenspeicher zugreifen und der Zugriff somit *coalesced* ist. Die Geschwindigkeit wird hierdurch weiter um einen Faktor 2,9 auf eine Laufzeit von unter 2 min verbessert.

Durch den Einsatz von Textureinheiten wird schließlich eine weitere Technik zur Laufzeitverbesserung von Bildverarbeitungsalgorithmen angewendet. Moderne Grafikkarten bieten neben speziellen Textur-Caches auch Recheneinheiten zur Interpolation zwischen Texturelementen (Pixeln). In unserem Fall kann dieser besondere Speichertyp für die Projektionsbilder verwendet werden. Sind angeforderte Werte bereits im Cache vorhanden, so werden die Zugriffe auf den Hauptspeicher der Grafikkarte reduziert. Durch die Verwendung von Texturen

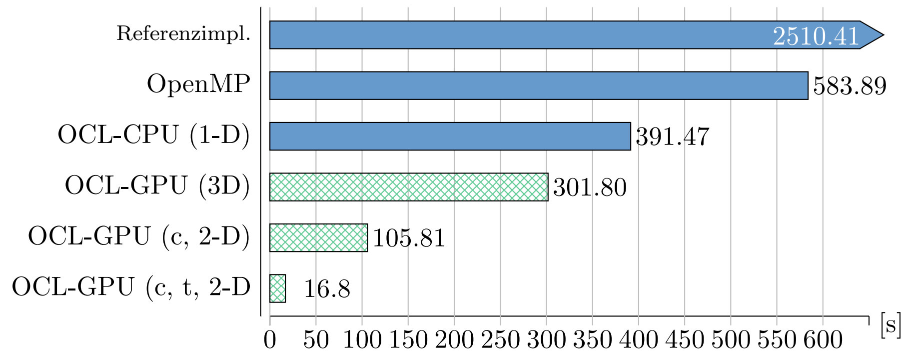

Abb. 1. Laufzeit verschiedener Programme in [s]. Kürzere Balken sind besser. (c) bedeutet *coalesced* Speicherzugriff, (t) bedeutet Benutzung von Textureinheiten. Die oberen drei Balken (blau) nutzen die CPU, die unteren drei (grün) die GPU.

können 4 Speicherzugriffe und deren bilineare Interpolation in Software durch einen einzigen Zugriff auf die Textur ersetzt werden. Letztendlich konnte durch die Verwendung von Texturen ein weiter Geschwindigkeitszuwachs um Faktor 6,3 auf eine Laufzeit von unter 17 s erreicht werden.

4 Diskussion

Im Rahmen dieser Arbeit spielte die Erzielung der maximalen Leistung eine sekundäre Rolle. Primär sollte die einfache Verwendung von OpenCL gezeigt werden, mittels der sich sowohl eine beträchtliche Leistung als auch ein einfacher Wechsel zwischen unterschiedlicher Hardware erreichen lässt. Die Ergebnisse in Abbildung 1 zeigen eindrucksvoll den durch die Benutzung von OpenCL möglichen Geschwindigkeitsgewinn. Da die OpenCL *Kernel* alle im gleichen C-Dialekt geschrieben sind, ist der Unterschied zwischen ihnen marginal und ermöglicht sowohl eine schnelle Realisierung als auch einen einfachen Umstieg zwischen Hardwarearchitekturen. Hierbei anzumerken ist der Umstand, dass die gezeigten Optimierungen Wissen über die Architekturen voraussetzen und nutzen, was die Portabilität einschränkt. Erwähnenswert ist auch die Tatsache, dass die verwendete Grafikkarte kein Modell der neuesten Generation ist, und hier weitere Geschwindigkeitszuwächse möglich sind.

5 Zusammenfassung und Ausblick

Zusammenfassend kann festgestellt werden, dass OpenCL ein sehr vielversprechendes Framework darstellt. Wenngleich noch Optimierungen für verschiedene Plattformen notwendig sind, ist der Code im allgemeinen sehr portabel. Dabei bietet OpenCL bereits auf der CPU Geschwindigkeitsvorteile gegenüber dem etablierten Framework OpenMP. Der Umstieg auf die Grafikkarte ist einfach realisierbar und birgt weitere enorme Geschwindigkeitszuwächse.

Wir gehen davon aus, dass OpenCL zukünftig ein weites Anwendungsspektrum und großes Interesse findet. Eine weitere Verbesserung der Compiler ist zu erwarten, wodurch eventuell auch die beschriebenen Optimierungen automatisiert werden.

Literaturverzeichnis

1. Hofmann HG, Keck B, Hornegger J. Accelerated C-arm Reconstruction by Out-of-Projection Prediction. In: Deserno TM, Handels H, Meinzer HP, Tolxdorff T, editors. Bildverarbeitung für die Medizin 2010. Berlin; 2010. p. 380–384.
2. Feldkamp LA, Davis LC, Kress JW. Practical Cone-Beam Algorithm. Journal of the Optical Society of America. 1984;A1(6):612–619.
3. Rohkohl C, Keck B, Hofmann HG, Hornegger J. RabbitCT—An Open Platform for Benchmarking 3-D Cone-beam Reconstruction Algorithms. Medical Physics. 2009 Sep;36(9):3940–3944. Available from: http://link.aip.org/link/?MPH/36/3940/1.