Heterogeneous Computing
Abschlussprojekt - GPU Pregel

Kilian Bartz (Mknr.: 1538561)

Hinweis

Der zugehörige Code befindet sich im folgenden Github Repository: `https://github.com/kilianbartz/gpu_pregel`.

Einleitung - Takeaways der letzten GPU-Übung

In meinem letzten GPU Experiment im zweiten HetComp Übungsblatt konnte ich zwar einen deutlichen Performance-Gewinn der GPU gegenüber der Singlecore-Lösung erreichen, jedoch lag sie trotzdem um ein Vielfaches hinter der beeindruckenden Leistung der Multicore-Implementierung. Dies war vor allem diesen drei Punkten geschuldet:

- Die Multicore-Lösung (wie auch die Singlecore-Lösung) waren durch die Verwendung von Rust mit *rustfft*[1] und die dort eingesetzte AVX-Vektorisierung sehr effizient (sodass theoretisch pro Kern bis zu 16 Single-Precision Float Operationen parallel ausgeführt werden können), sodass die Vorteil der GPU mit ihren vielen Kernen bereits deutlich schrumpft. Zusätzlich bot *rayon*[2] eine sehr effiziente Multicore-Datenverarbeitung, welche kaum Overhead erzeugt. Im Gegensatz dazu war der Overhead der GPU Lösung vermutlich deutlich größer, da nach jeder Iteration mittels `hipDeviceSynchronize()` auf die GPU gewartet wurde und sehr viele Copy-Operationen (Host → Device und Devie → Host) durchgeführt werden mussten.
- Dadurch, dass ich AMDs HIP SDK (in eingeschränkter Form durch Windows) für die Implementierung der GPU Lösung verwendet habe, hatte ich mit schlechter Dokumentation und einem ohnehin sehr jungen, deutlich featureärmeren Framework zu kämpfen. Somit war mein Code für die GPU-Lösung sicherlich nicht optimal.
- Zur Evaluation habe ich eine AMD RX 6900 XT[3] verwendet, welche nur über 5120 Stream-Prozessoren in 80 Recheneinheiten verfügt. Neben der Software (HIP SDK) scheint auch die Hardware zur Beschleunigung von generischen Abläufen noch unausgereift, was man in der deutlich schwächeren Performance von PyTorch[4], einem stark optimierten Framework für Deep Learning, auf AMD GPUs erkennt. Konkret kam es bei mir zu einer starken Einschränkung aufgrund eines (nicht dokumentierten) Limits von nur 1000 batches pro FFT-Iteration, was die Zahl der nötigen Copy-Operationen stark erhöht hat und somit der Effizienz der gesamten Anweundung beeinträchtigt hat. Verglichen mit der Nvidia RTX 4090[5] sollte die AMD GPU mit über 46 TFLOPS mehr als die Hälfte der Leistung der Nvidia Karte (83 TFLOPS) bieten, diese hat jedoch mit 16384 CUDA Cores mehr Kerne, sodass komplexe Berechnungen sich in der Praxis besser parallelisieren lassen sollten.

Da mir aufgrund meiner Erfahrungen im KI-Bereich jedoch bewusst ist, wie viel leistungsstärker eine GPU verglichen mit einer CPU sein kann (z. B. in der Inferenz größerer Sprachmodelle), wollte ich in diesem Projekt an einer Anwendung, die sich optimal für die Parallelisierung auf einer GPU eignet, versuchen, einen starken Speedup gegenüber einer CPU-Lösung zu erreichen. Aufgrund meiner Erkenntnisse im 2. Übungsblatt versuchte ich dieses Mal, von Grund auf eine CUDA Implementierung zu schreiben, sodass ich unabhängig von eventuell suboptimalen Bibliotheken bin. Diese Implementierung habe ich dann (u. a.

[1] `https://docs.rs/rustfft/latest/rustfft/`
[2] `https://docs.rs/rayon/latest/rayon/`
[3] `https://www.amd.com/de/support/downloads/drivers.html/graphics/radeon-rx/radeon-rx-6000-series/amd-radeon-rx-6900-xt.html`
[4] `https://pytorch.org/`
[5] `https://www.nvidia.com/en-us/geforce/graphics-cards/40-series/rtx-4090/`

unterstützt durch die KI in Form von Claude 3.5 Sonnet) auf die CPU und in das AMD HIP SDK portiert. Durch dieses Vorgehen erhoffte ich mir eine bessere Code Qualität, da es zu CUDA ein Vielfaches mehr an Dokumentation und Beispielen gibt, und infolge dessen eine effizientere Implementierung, die durch eine direkte Portierung gut vergleichbar sein sollte.

Anwendung dieses Projekts - GPU Pregel

In vielen Anwendungen sind Graphalgorithmen unersetzlich, z. B. für die Analyse von Social Media Netzwerken, in der Navigation, für Recommender Systeme (v. a. Clustering) oder zum Ranken von Webseiten einer Suchmaschine durch den PageRank Algorithmus [1]. Facebook nannte beispielsweise 2016 Graphverarbeitung öne of many important parts of the Data Infrastructure backend services at Facebook", da sie in vielen Recommender Systemen und für Infrastruktur-Optimierungen eine wichtige Rolle spielen [2]. Auch wenn sich sowohl Google als auch Facebook / Meta ziemlich verschlossen hinsichtlich der aktuell genutzten Techniken halten und kürzlich immer mehr auf Deep Learning und Embeddingansätze setzen [4, 8, 7, 3], ist davon auszugehen, dass Graphalgorithmen nach wie vor eine wichtige Rolle spielen.
Für mein Projekt habe ich zwei Graphalgorithmen ausgewählt, die ich implementiert habe:

1. PageRank. Dieser Algorithmus funktioniert sehr intuitive und lässt sich in vielen Anwendungsfeldern einsetzen (immer dort, wo die Wichtigkeit einzelner Knoten in einem Graphen berechnet werden soll). Zudem ermöglichen die Fließkommazahlen als Bewertungen der Knoten, die Effizienz von `double` und `float` Implementierungen zu vergleichen.

2. Single source shortest paths (SSSP) / Dijkstra Algorithmus. Dieser Algorithmus ist wohl erforscht und wird in vielen Anwendungen eingesetzt. Im Gegensatz zu all-pairs shortest paths (z. B. Floyd-Warshall) wird jedoch kein quadratischer Speicher ($\mathcal{O}(|V|^2)$) benötigt, während der Algorithmus trotzdem bessere Parallelisierung bietet, als ein einzelner kürzester Weg [6].

Wie parallelisiert man effizient Graphalgorithmen?

Hier habe ich meine Inspiration von Googles Pregel [6] genommen, welche ich dieses Semester in der Big Data Vorlesung kennengelernt habe. Hierbei handelt es sich um ein generelles Framework zur Graphverarbeitung, welches Graphalgorithmen auf ganzen Serverclustern verteilen kann. Die Grundidee ist es hier, dass jede Einheit (z. B. jeder physische CPU-Kern im Cluster) **aus der Sicht eines Knotens** im Graphen $G = (V, E)$ arbeitet. Die `compute`-Funktion eines Knoten $x \in V$ verarbeitet dabei eingehende Nachrichten aus der vorherigen Iteration, aktualisiert seinen internen Zustand und sendet Informationen an seine Nachbarn $y \in V$ mit $(x, y) \in E$ für die nächste Iteration. Es handelt sich also um ein verteiltes System, in dem die `compute`-Funktion aller Knoten parallel berechnet werden kann, jedoch am Ende einer Iteration ("superstep" genannt) eine Synchronisierung erfolgen muss, damit ein Knoten sämtliche Nachrichten seiner Vorgänger für den nächsten superstep erhält ("bulk synchronous parallel"). Die Knoten verhalten sich ähnlich zum Aktor-Modell: Nach einem superstep kann ein Knoten inaktiv werden (`voteToHalt`); durch den Erhalt einer Nachricht wird ein Knoten reaktiviert. Die gesamte Berechnung ist beendet, wenn alle Knoten inaktiv sind und keine Nachricht mehr unterwegs ist.

Beispiel

Berechnung des maximalen Knotenwerts im Graphen.

- Jeder Knoten startet aktiv und sendet jedem Nachbar seinen eigenen Wert.
- Im darauf folgenden superstep erhöht er seinen eigenen Wert w, falls der Wert in einer Nachricht w übertrifft.
- Falls sich w verändert hat, wird dieser Wert an alle Nachbarn gesendet
- Nach dem Senden: `voteToHalt`
- Nach dem das System terminiert, wurde der höchste Wert gefunden und befindet sich in jedem Knoten des Graphen.

Wie funktioniert die Kommunikation zwischen den Knoten?

Pregel / Apache Giraph verwenden im Hintergrund Hadoop MapReduce. Die Kommunikation funktioniert dort so, dass eine Map-Iteration (in diesem Fall also die `compute`-Funktion eines einzelnen Knotens) eine Menge an **Key-Value-Paaren** produziert. In diesem Fall ist der Key die ID des Nachbarknotens und der Value w der neue Knotenwert. Es werden also Paare $(y : w)$ für $(x, y) \in E$ als Ausgabe eines supersteps produziert. Vor dem nächsten superstep werden alle Paare $(x : w')$, die von Vorgängern von x produziert wurden, gesammelt und in der nächsten Iteration der `compute`-Funktion aggregiert, um den neuen PageRank Wert zu berechnen. Ein simples Beispiel ist in Abbildung 1 zu finden.

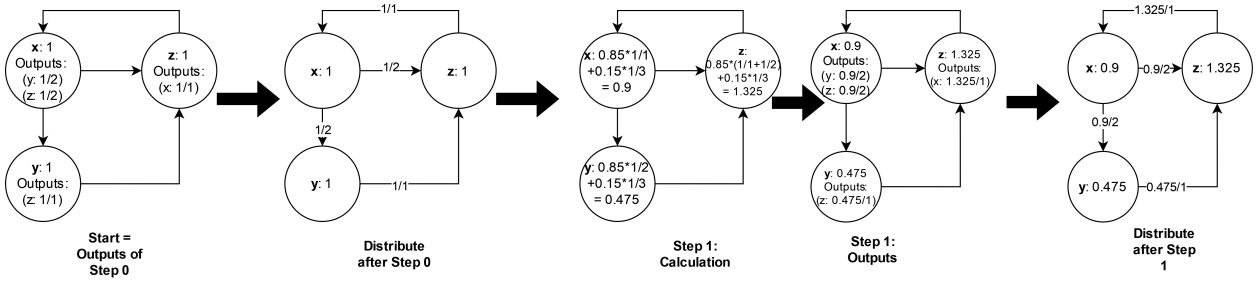

Abbildung 1: Simples PageRank Beispiel für 3 Knoten und einen superstep. Die Knoten x,y,z starten jeweils mit 1 als PageRank Wert. Zur Berechnung der PageRank Werte folgender Iterationen werden zunächst die Output Key-Value-Paare berechnet, welche dann an die entsprechenden Nachbarknoten verteilt werden. In der nächsten Iteration kann die entsprechende `compute`-Funktion auf diese Werte zugreifen, um den nächsten Wert zu berechnen.

Dieses Framework (bzw. seine Open Source Implementierung Apache Giraph[6]) wird verwendet, um Graphalgorithmen auf sehr großen Graphen auf Serverclustern zu berechnen. Auch wenn seine Effizienz durch die Synchronisierung nach jedem superstep leiden kann, wurde es 2013 erfolgreich von Facebook genutzt, um eine PageRank Iteration auf einem Graphen mit 10^{12} Kanten auf 200 Rechnern zu berechnen. Bei solch großen Graphen fällt die Synchronisierung dementsprechend wahrscheinlich kaum ins Gewicht, da jeder CPU-Kern so viele Knoten verarbeiten muss, dass der Aufwand für jeden Kern im Mittel sehr ähnlich ist.

Übertragung dieser Idee auf die GPU

Die Idee hinter diesem Projekt ist es, die Funktionsweise von Pregel nach dem Motto "die Grafikkarte ist der Supercomputer des kleinen Mannes" auf eine GPU zu übertragen. Wie ein Servercluster verfügt auch

[6] `https://giraph.apache.org/`

eine GPU über zahlreiche, einzeln ansprechbare / programmierbare Prozessoren, die sich gut dafür eignen, skalierbare Probleme parallel zu verarbeiten. Zusätzlich ist die Speicherstruktur dadurch, dass jeder GPU-Kern auf die Graphstruktur im gemeinsamen Speicher zugreifen kann, deutlich simpler als bei Pregel, wo jeder Rechner nur eine Partition des Graphen kennt.
Meine Idee war es also, ein Framework (ähnlich zu einem Pregel-lite) zu implementieren, das es ermöglichen würde, mit der Angabe einer `compute`-Funktion als Kernel und eines Eingabegraphen beliebige Graphalgorithmen auf der GPU berechnen zu lassen.
Allerdings bereitet die Kommunikation zwischen den Knoten des Graphen, die in einem verteilten System wie Pregel zwangsläufig nachrichtenbasiert ist, auf eine GPU Probleme. Man kann sich durchaus eine passende Datenstruktur überlegen, um nachrichtenbasierte Kommunikation umzusetzen (siehe Abbildung 2), was ich es in meinen ersten AMD HIP Versuchen im Ordner *legacy* ausprobiert habe. Dort habe ich eine generische

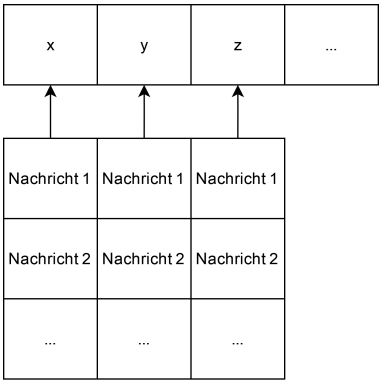

Abbildung 2: Mögliche Umsetzung einer nachrichtenbasierenden Kommunikation zwischen Graphknoten im gemeinsamen GPU-Speicher. Das Feld hat einen Eintrag pro Knoten, der auf den Kopf einer entsprechenden Nachrichten-Queue zeigt.

Klasse ThreadSafeVector implementiert, welche *atomic*-Operationen verwendet, um sicherzustellen, dass es während dem hinzufügen von Einträgen nicht zu Race-Conditions kommt.

Hinweis

atomic-Operationen (z. B. *atomicAdd* `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=atomicadd#atomicadd`) verwende ich auch in meiner finalen Implementierung. Sie lesen den Inhalt einer Speicheradresse, berechnen den neuen Wert und schreiben diesen zurück an die ursprüngliche Adresse. Die Atomizität wird dabei als spezielle Anweisung durch die Hardware garantiert und sollte so keine Performance kosten.

Zusätzlich habe ich eine generische `PregelVertex`-Klasse definiert, die beliebige Wertetypen (z. B. doubles, floats, ints), Kantentypen und Nachrichtentypen erlaubt.

ersterEntwurf.hip

Dieser erste Versuch veranschaulicht, wie ich mir das Framework vorgestellt habe. Nach der Initialisierung der Laufparameter (Anzahl der Iterationen, Anzahl der Knoten) und der Speicherinitialisierung würde man in einem simplen Schleife den `compute`-Kernel (hier für PageRank) und anschließend das Ergebnis ausgeben. Wie man hier sieht, erlauben es die generischen Klassen, für das eigentliche Problem sehr simplen Code zu schreiben. Vor allem die `compute`-Funktion ist durch das *vertex-centric* Paradigma sehr einfach verständlich. Es werden lediglich zwei Nachrichten-Queues (`messageBoardOld`, `messageBoardNew`) benötigt, wobei die erste für die Berechnung des aktuellen Wertes dient und die zweite für den nächsten superstep. Benutzt man stattdessen nur eine Queue, kann es passieren, dass der aktuelle Wert eines Vorgängers y in die Nachrichten-Queue von x geschrieben wird und in dieser Iteration einfließt, obwohl er erst im nächsten superstep relevant

wäre. So gut lesbar der Code allerdings war, gelang es mir nicht, das Beispiel mit nur 2 Knoten auszuführen (Treiber-Timeout nach einigen Sekunden).

claude.hip

Im zweiten Anlauf behielt ich das generische Framework bei, probierte jedoch die Klassen `host_vector` und `device_vector` aus der `thrust`-Bibliothek aus, da ich einen Fehler in meiner `ThreadSafeVector`-Implementierung vermutete. Wenngleich sich diese Implementierung zwar ausführen lässt, funktioniert hier die Distribution der PageRank-Werte für die nächste Iteration nicht, sodass nicht die korrekten Werte PageRank-Werte berechnet werden.

main.hip

Um dem Problem aus der vorherigen Implementierung zu begegnen, entschied ich mich hier, die Distribution der neuen PageRank-Werte in einen gesonderten Kernel auszulagern, der nach einer Synchronisation nach der Berechnung der neuen Werte erst ausgeführt wird. Außerdem verwendete ich hier erstmalig meinen`ImportGraph`-Header, der eine Kantenliste aus einer Textdatei liest und eine entsprechende Adjazenzliste aufbaut.
Allerdings ist hier das Ergebnis, dass zwar scheinbar ein einzelner PageRank-Wert berechnet wird, dies jedoch ca. 56s für 2 Iterationen auf dem web-NotreDame Graph (siehe Abschnitt 3) dauert. Während dieser Zeit friert außerdem Windows ein und es kommt bei jeglicher Interaktion mit dem Rechner zu einem Treibertimeout.

Finale Implementierung - PageRank

Meine finale PageRank Implementierungen sind mit *pagerank_...* beschriftet. Es handelt sich um die ursprüngliche CUDA Implementierung (*pagerank_double.cu*), sowie Portierungen zu AMD HIP und auf die CPU (singlethreaded bzw. mit OMP als Multithreading-Version). Während ich an `thrust`-Vektoren und der Zweiteilung der `compute`-Funktion festgehalten habe (Distribution in `pagerank_kernel`; Berechnung des neuen Wertes in `apply_damping_kernel`), ist der gesamte Algorithmus der Sauberkeit wegen in die PageRank Klasse ausgelagert worden. Diese umfasst sowohl die (Speicher-) Intialisierung als auch die `compute`-Funktion für einen einzelnen superstep. Hier werden *cudaEvents* verwendet, um korrekt die Dauer einer Iteration zu messen (selbst mit Synchronisierung ist eine Messung mittels `std::chrono` komplett unbrauchbar). Zudem wird nach jedem superstep die absolute Abweichung zu den vorherigen Werten berechnet, um nach Konvergenz vorzeitig abbrechen zu können. Dafür kommt die effiziente `thrust::transform_reduce`-Operation zum Einsatz.
Schaut man sich die Kernels an, so ist der größte Unterschied zu vorherigen Versuchen, dass das Graph hier **edge-centric** statt **vertex-centric** verarbeitet wird. Dies sorgt für eine bessere Parallelisierung, da Prozessoren, die Knoten mit wenigen Nachbarn verarbeiten, nicht mehr auf Knoten mit vielen Nachbarn warten müssen. Stattdessen können Kanten ohne eine feste Reihenfolge verarbeitet werden. Dies vereinfacht auch die Datenstruktur erheblich, da keine Nachrichten-Queues mehr benötigt werden müssen, sondern nur zwei Fließkommavariablen pro Knoten (für den alten und neuen PageRank-Wert). Deshalb gilt der Algorithmus nun auch als terminiert, wenn eine Konvergenz erreicht wurde, oder die maximalen Iterationen durchgeführt wurden. Ein aktiv/passiv Status, sowie Nachrichten werden nicht mehr beachtet. Es gilt zu beachten, dass es sicherlich auch Graphalgorithmen gibt, für die eine solche Verarbeitung nicht möglich ist - potenziell wird durch den edge-centric Ansatz also die Mächtigkeit des Frameworks eingeschränkt.

Finale Implementierung - SSSP

Meine finalen Implementierungen für das SSSP Problem sind mit *dijkstra_...* beschriftet. Hier gibt es jedoch nur eine Version pro Plattform, da nur mit ganzzahligen Distanzen gerechnet wird und somit in jeder Implementierung *int* zum Einsatz kommt. Auch hier habe ich wieder den CUDA-Code als Grundlage verwendet und mittels eines Sprachmodells auf die anderen Plattformen portieren lassen.
Der Dijkstra-Code für die GPUs folgt im Wesentlichen dem bekannten Algorithmus, wobei keine Queue verwendet wird, sondern stattdessen in jeder Iteration alle Kanten erneut untersucht werden, ob ein neuer

kürzester Weg gefunden wurde. Durch die *atomicMin*-Operation werden auch hier Race Conditions aufgrund zweier Kerne, die den Wert derselben Strecke aktualisieren wollen, verhindert. Eine `boolean`-Flag überwacht, ob mindestens eine Strecke in der aktuellen Iteration aktualisiert wurde, andernfalls terminiert der Algorithmus.
Im Gegensatz dazu folgt der CPU-Code der bekannten Implementierung mit einer PriorityQueue.

Evaluation

Zur Evaluierung der PageRank und SSSP Implementierungen habe ich echte Graphen aus der *Stanford Large Network Dataset Collection* [5] verwendet. Konkret kamen die gerichteten Graphen "web-NotreDame", "web-BerkStan" (beides Graphen, welche die Links zwischen den Webseiten der entsprechenden Universitäten darstellen) und "soc-LiveJournal1" (Graph repräsentiert Freundschaften im sozialen Netzwerk LiveJournal) zum Einsatz. Eine Übersicht der Graphen findet sich in Tabelle 1. Als Hardware wird eine Nvidia V100 (32GB VRAM) eingesetzt, um die CUDA-Version zu testen, eine AMD RX 6900XT für die HIP-Version und ein AMD 7700X für die CPU-Version.

Data set link	**# Knoten**	**# Kanten**	**K / SCC**
`https://snap.stanford.edu/data/web-NotreDame.html`	325,729	1,497,134	0.166
`https://snap.stanford.edu/data/web-BerkStan.html`	685,230	7,600,595	0.489
`https://snap.stanford.edu/data/soc-LiveJournal1.html`	4,847,571	68,993,773	0.790

Tabelle 1: Übersicht der wichtigsten Merkmale der Eingabegraphen. K / SCC bezeichnet den Anteil der Knoten, die sich in der größten starken Zusammenhangskomponente befinden.

Hinweis

Neben den oben aufgeführten Graphen wollte ich auch den riesigen "Friendster social network and ground-truth communities" Graph[a] mit ca. 65M Knoten und 1.8B Kanten in die Evaluation aufnehmen. Die zugehörige Kantenliste war als Textdatei ca. 22GB groß. Jedoch habe ich den Ladevorgang auf allen Geräten nach 20min abgebrochen, da der VRAM der AMD GPU sicherlich nicht für eine Verarbeitung ausgereicht hätte.

[a] `https://snap.stanford.edu/data/com-Friendster.html`

Vorgehensweise

In allen Implementierungen startet jeder Knoten mit $\frac{1}{\#Knoten}$ als Initialwert. Nachdem ich alle PageRank Implementierungen evaluiert habe, konnte ich erfolgreich verifizieren, dass die Ausgaben alle identisch waren, jede Implementierung hat also erwartungsgemäß dieselben Ergebnisse berechnet. Anschließend habe ich die jeweils mit dem Knoten mit dem höchsten PageRank das SSSP Problem berechnet. Zum Einsatz kamen hier Kantengewichte, die mittels *create_weights.py* einmalig zufällig generiert wurden.

Ergebnisse

Die Laufzeiten der einzelnen Implementierungen ist in Tabelle 2 und in Abbildung 3 zu finden.
Wie zu erwarten ist der **Vorteil der GPUs** für die PageRank-Berechnung erst bei soc-LiveJournal als **sehr großem Graphen** deutlich zu sehen. Dies ist auch der einzige Graph, bei dem die AMD GPU einen klaren Vorteil (etwa die 14-fache Leistung) gegenüber der CPU erreicht. Vor allem der Ausreißer beim "web-Berkley-Stanford" Graph, bei dem die AMD GPU fast 10x so lange braucht, wie die CPU zeigt, dass **AMD GPUs** im Computing-Bereich noch einige **Schwächen** hat und scheinbar nicht optimal mit allen Eingabegraphen klar kommt. Dem gegenüber ist die Nvidia GPU jedes Mal deutlich (zwischen 10x und 15x) der CPU überlegen und schlägt v. a. bei kleineren Graphen auch deutlich die AMD GPU.

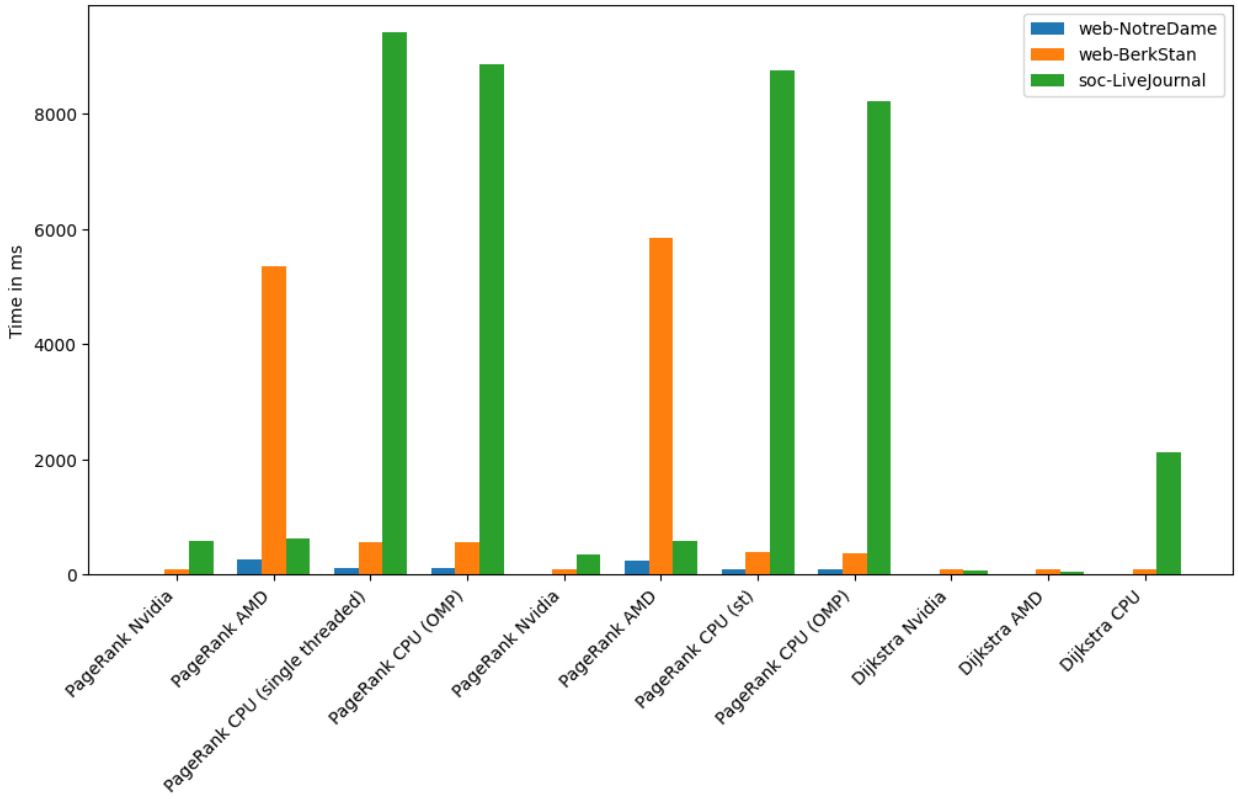

Abbildung 3: Vergleich der Laufzeiten der verschiedenen Implementierungen basierend auf den Laufzeiten aus Tabelle 2

Die Multicore CPU Lösung (OMP) lohnt sich nur geringfügig beim größten Graphen und bringt in den anderen Fällen nur eine Verschlechterung.
Hinsichtlich des Vergleichs **Double vs. Float** habe ich erwartet, nur einen leichten Performance-Vorteil für Floats zu finden. Ihre Berechnung kann zwar schneller (z. B. für AMD GPUs fast doppelt so schnell) durchgeführt werden, jedoch verbringt der PageRank Algorithmus wahrscheinlich einen Großteil der Zeit eher damit, durch Listen / Arrays zu iterieren. Überraschenderweise ist dies für den kleinsten Graphen auch der Fall, auf dem Berkley-Stanford Graphen benötigen die GPUs mit Floats sogar länger (die CPU deutlich kürzer) und auf dem größten Graphen profitiert nur die Nvidia GPU deutlich von Floats. Der Performancegewinn durch Floats scheint also **stark eingabeabhängig** zu sein und schwer zu schätzen.
Mit Blick auf SSSP (Dijkstra) sind beide GPUs der CPU überlegen, zeigen jedoch erneut erst beim größten Graphen ihr Potenzial mit einem Speedup von ca. 36. Dies ist auch der einzige Test, in dem die AMD GPU besser abschneidet als die Nvidia GPU.

Abschließende Bemerkungen

Meinen ursprünglicher Plan, ein modulares Framework wie Pregel zu schaffen, in dem beliebige parallelisierbare Graphalgorithmen ausgeführt werden können, musste ich leider schnell revidieren. Ein nachrichtenbasiertes, vertex-centric System funktioniert schlecht auf einer GPU. Stattdessen hatte ich Erfolg damit, möglichst primitive Datenstrukturen (Arrays von Zahlen) zu verwenden und konnte damit tatsächlich in den zwei Graphprobleme, PageRank und SSSP, sehr effizient (zumindest unter Verwendung von Nvidia Hardware) lösen. Die Performance auf meiner AMD GPU war stark von der Eingabe abhängig - hoffentlich bessert sich dies in den nächsten Jahren mit der Reife von AMD HIP.
Obwohl der Ansatz, den ich in meinem Projekt verfolgt habe, Graphalgorithmen edge-centric mit der Hilfe

von `thrust`-Datenstrukturen zu lösen vermutlich nicht auf jedes Graphproblem übertragbar ist, finde ich, dass man trotzdem das Potenzial einer "Grafikkarte als Supercomputer des kleinen Mannes" erkennen kann.

Implementierung	web-Notre-Dame	web-Berkley-Stanford	soc-LiveJournal
PageRank Doubles Nvidia	11	88	589
PageRank Doubles AMD	271	5362	625
PageRank Doubles CPU (single threaded)	112	564	9411
PageRank Doubles CPU (OMP)	115	570	8857
PageRank Float Nvidia	9	96	350
PageRank Floats AMD	251	5837	593
PageRank Floats CPU (st)	87	398	8755
PageRank Floats CPU (OMP)	84	371	8210
Dijkstra Nvidia	1	84	71
Dijkstra AMD	1	97	59
Dijkstra CPU	1	101	2116

Tabelle 2: Ergebnisse der Evaluation meiner Graphalgorithmen-Implementierungen. Bei den Werten handelt es sich um die Laufzeiten der Graphalgorithmen (ohne Aufbau der Adjazenzlisten) bis zur Terminierung in ms.

Literatur

[1] Brin, S., and Page, L. The anatomy of a large-scale hypertextual web search engine. Computer networks and ISDN systems 30, 1-7 (1998), 107–117.

[2] Kabiljo, M., Logothetis, D., Edunov, S., and Ching, A. A comparison of state-of-the-art graph processing systems — engineering.fb.com. `https://engineering.fb.com/2016/10/19/core-infra/a-comparison-of-state-of-the-art-graph-processing-systems/`, 2016. [Accessed 24-09-2024].

[3] Lada, A., Wang, M., and Yan, T. How machine learning powers Facebook's News Feed ranking algorithm — engineering.fb.com. `https://engineering.fb.com/2021/01/26/ml-applications/news-feed-ranking/`, 2021. [Accessed 24-09-2024].

[4] Lerer, A., Wu, L., Shen, J., Lacroix, T., Wehrstedt, L., Bose, A., and Peysakhovich, A. Pytorch-biggraph: A large-scale graph embedding system. CoRR abs/1903.12287 (2019).

[5] Leskovec, J., and Krevl, A. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[6] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. Pregel: a system for large-scale graph processing. In Proceedings of the 2010 international conference on Management of data (New York, NY, USA, 2010), pp. 135–146.

[7] Medvedev, I., Wu, H., and Gordon, T. Powered by AI: Instagram's Explore recommender system — ai.meta.com. `https://ai.meta.com/blog/powered-by-ai-instagrams-explore-recommender-system/`, 2019. [Accessed 24-09-2024].

[8] Zha, D., Feng, L., Tan, Q., Liu, Z., Lai, K.-H., Bhushanam, B., Tian, Y., Kejariwal, A., and Hu, X. Dreamshard: Generalizable embedding table placement for recommender systems, 2022.