FH MÜNSTER
University of Applied Sciences

Claus Grewe (Hrsg.)

Abschlussbericht FEP 2023/2024

FH Münster – Master Wirtschaftsinformatik

14. April 2024

Münster

Vorwort

Das Curriculum des Studiengangs Master of Science Wirtschaftsinformatik an der FH Münster beinhaltet das Wahlpflichtmodul „Forschungs- und Entwicklungsprojekt“. Das Modul ermöglicht es Studierenden, sich forschungsorientiert mit innovativen Themen auseinanderzusetzen. Die behandelten Themen werden von Jahr zu Jahr neu gewählt und umfassen sowohl aktuelle Forschungsgebiete als auch Innovationen, die den IT-Markt bzw. die Wirtschaftsinformatik gerade erreichen bzw. noch nicht durchdrungen haben.

Das Forschungs- und Entwicklungsprojekt im Wintersemester 2023/2024 befasste sich inhaltlich mit der Fragestellung, wie sich hochperformanter Code in systemfernen Programmiersprachen integrieren lässt? Ausgangspunkt der Betrachtung ist die Beobachtung, dass gängige Programmiersprachen wie Java, Python oder JavaScript einen hohen Grad der Abstrahierung von maschinennahen Details anstreben, was u. a. zu deren Verbreitung und Zuverlässigkeit beiträgt. Andererseits führt die Systemferne zur erschwerten Integration und Nutzung von parallelen SIMD-Strukturen, wie sie in modernen CPUs und GPUs heute üblicherweise vorliegen. Derzeitig wird in verschiedenen Projekten – sowohl im Umfeld von Java als auch in einer Working Group des World Wide Web Consortiums – an Vorschlägen zur verbesserten SIMD-Integration in der Sprache Java bzw. im Browser gearbeitet.

Ein Projektteam – bestehend aus den Mitgliedern Niklas Lohmann, Marcus Kligge und Jannis Theile – setzte sich daher mit den folgenden Fragestellungen auseinander:

- Wozu dienen die Schnittstellen *WebGPU* und *WebGL* und welche Leistungssteigerungen erzielen die Ansätze in browserbasierten Anwendungen?
- Wie lassen sich mit Hilfe der Schnittstellen *Foreign Functions & Memory API*, *Java Vector API* und *Java Native Interface* (JNI) SIMD-Operationen in Java-Programmen einbeziehen und worin unterscheiden sich die Ansätze?

Die Ergebnisse der Untersuchungen wurden in Form eigenständiger Beiträge verfasst und in diesem Abschlussbericht zusammengetragen.

Münster, im April 2024

Prof. Dr. Claus Grewe

Inhaltsverzeichnis

Leistungsvergleich von WebGL und WebGPU als Alternativen für die Implementierung einer Computer Vision Anwendung im Browser

Niklas Lohmann[1]

Abstract: Anwendungen setzen immer häufiger leistungsintensive Methoden aus dem Bereich der künstlichen Intelligenz ein, um beispielsweise dem Nutzer Aufgaben abzunehmen. Es existieren verschiedene JavaScript-Bibliotheken, die eine Implementierung und Ausführung derartiger Anwendungen innerhalb des Browsers ermöglichen. WebGL und WebGPU sind JavaScript-APIs, die die Nutzung von GPU-Beschleunigung innerhalb des Browsers ermöglichen. Im Rahmen dieser Untersuchung wird ein Leistungsvergleich der beiden APIs erstellt, welcher auf einer für den Browser entwickelten Computer Vision Anwendung basiert. Der Vergleich wird durch eine Implementierung und Messung desselben Anwendungsfalls außerhalb des Browsers erweitert. Innerhalb der Untersuchung erwiesen sich beide APIs als praktikabel zur Erzielung von GPU-Beschleunigung innerhalb des Browsers. WebGPU erzielte im Anwendungsfall bessere Leistungsergebnisse als WebGL und konnte mit der erzielten Leistung der nativen Ausführung mithalten.

Keywords: Leistungsvergleich, WebGL, WebGPU, GPU-Beschleunigung im Browser

1 Einleitung

Verschiedene Innovationen im Bereich der künstlichen Intelligenz (KI) haben den Grundstein für heute verfügbare »Large Language Models« gelegt, die für breite Teile der Gesellschaft über Chatbots und andere Ansätze abstrahiert zugänglich sind. Die Leistungsfähigkeit derartiger Modelle hat einen Hype ausgelöst, der auch auf Unternehmen übergegriffen hat. Eine Studie des »World Economic Forum« hat ermittelt, dass bis 2027 74,9 % aller Unternehmen KI-Technologie einführen wollen [Wo23]. Ein weiterer Trend, der die Technologielandschaft von Unternehmen verändert, ist der stetig steigende Verbreitungsgrad von Webanwendungen, obwohl diese in puncto Leistungsfähigkeit klassischen Desktopanwendungen häufig nachstehen.

Die vollständige Implementierung einer KI-Anwendung als Webanwendung gestaltet sich aufgrund der beschränkten Leistungsfähigkeit eines Browsers als schwierig. Ein verbreiteter Ansatz zur Lösung des Problems ist die Ausführung des leistungsintensiven Teils der Anwendung auf einem hochperformanten Server. Hierbei muss der Serverbetreiber die Rechenkosten tragen und kann diese (wenn überhaupt) nur indirekt an den Anwender weitergeben. Weiterhin werden die zu verarbeitenden Daten häufig beim Anwender erzeugt, müssen zum Server gesendet und anschließend zurück übertragen werden, was Latenzzeiten verursacht. Die Ausführung des leistungsintensiven Anwendungsteils innerhalb des Browsers

1 FH Münster, Fachbereich Wirtschaft, Wirtschaftsinformatik, niklas.lohmann@fh-muenster.de,

würde die geschilderten Probleme lösen. In der vorliegenden Untersuchung wird sich mit der aus dem geschilderten Szenario stammenden Fragestellung beschäftigt, wie eine KI-Anwendung performant in der systemfernen Umgebung des Browsers ausgeführt werden kann. Ein Ansatz, der sich hierzu anbietet, ist die Nutzung von GPU-Beschleunigung. Hierbei wird die Grafikkarte des Computers eingesetzt, um Berechnungen beschleunigt durchzuführen. Die JavaScript-APIs »WebGL« und »WebGPU« eignen sich zur Nutzung von GPU-beschleunigten Berechnungen in einer Webanwendung. Ziel der vorliegenden Ausarbeitung ist die Erstellung eines Leistungsvergleichs der genannten APIs im Kontext der Ausführung einer KI-Anwendung im Browser.

Für die Durchführung der Untersuchung wird eine konstruktive Methodik bestehend aus der Erstellung einer prototypischen Implementierung der beschriebenen Anwendung und einer anschließenden Evaluation des Prototypen durch Leistungsanalysen angewendet. Als Anwendungsfall soll die Segmentierung von Personen und Körperteilen in Bildern und Videos implementiert werden. Dieser ist der KI-Disziplin »Computer Vision« zuzuordnen und eignet sich aufgrund der hohen benötigten Leistungsintensität. Der Anwendungsfall weist einen hohen Praxisbezug auf, da Segmentierungen von Personen in Bildern die Grundlage für beispielsweise Gestensteuerungen per Kamera oder das Verblenden des Hintergrunds in Videokonferenzen bilden.

2 Grundlagen

2.1 GPU-Berechnungen

Berechnungen auf der Grafikkarte weisen eine Stärke bei der Ausführung gleichförmiger Aufgaben wie der Grafikberechnungen für das Einfärben mehrerer Pixel auf. Da derartige Aufgaben auch außerhalb der Grafikverarbeitung häufig eingesetzt werden, hat sich die Disziplin der »General Purpose Computation on Graphics Processing Unit« (GPGPU) entwickelt. Hierbei werden Berechnungen auf der GPU ausgeführt, die über den ursprünglichen intendierten Aufgabenbereich einer Grafikkarte hinausgehen. Im Kontext des »Machine Learnings« (ML) ist die Matrixmultiplikation eine häufig eingesetzte und gleichförmige Operation, die sich für die Ausführung auf einer Grafikkarte eignet. Um Berechnungen auf einer Grafikkarte durchzuführen, werden unter anderem »Shader« eingesetzt. Hierunter sind Programme zu verstehen, die in einer formalen Sprache definiert werden (der »Shading Language«), zur Laufzeit in eine der Grafikkarte verständlichen Maschinensprache übersetzt und durch Shadereinheiten der GPU ausgeführt werden. Shader werden in verschiedene Unterarten unterschieden: Ein »Fragment Shader« wird in der Grafikverarbeitung verwendet, um Farbwerte und andere Eigenschaften von Pixeln bzw. Fragmenten zu berechnen. Hierbei können »Texturen« eingesetzt werden, mit denen zusätzliche Daten in die Berechnungen integriert werden können. Eine Form zur Realisierung von GPGPU ist die Abbildung zu berechnender Operationen auf Fragment Shader und Texturen und die anschließende Ausführung auf einer Grafikkarte. Der »Compute Shader« ist losgelöst von Grafikberechnungen

und ermöglicht die Durchführung allgemeiner Berechnungen auf einer Grafikkarte. In diesem Kontext werden »Storage Buffers« als Datenstruktur eingesetzt, die es einem Compute Shader ermöglichen, Daten zwischen Berechnungen auszutauschen und zu speichern. [LGK23]

»WebGL« ist eine JavaScript-API mit der 3D-Berechnungen im Browser ohne den Einsatz zusätzlicher Plugins auf der Grafikkarte hardwarebeschleunigt ausgeführt werden können. WebGL ist auf die Berechnung von 3D-Grafiken ausgelegt, kann allerdings auch für GPGPU im Browser verwendet werden. »WebGPU« ist der Arbeitstitel eines zukünftigen Web-Standards sowie eine JavaScript-API für die Ausführung hardwarebeschleunigter Berechnungen auf der Grafikkarte innerhalb des Browsers. WebGPU wird als Nachfolger von WebGL gehandelt und wurde 2023 in den Google Chrome Browser integriert.

2.2 TensorFlow.js

Die JavaScript-Bibliothek »TensorFlow.js« ermöglicht die Implementierung verschiedener Typen von Aufgaben aus dem Bereich des maschinellen Lernens in nativen JavaScript und die anschließende Ausführung innerhalb des Browsers. TensorFlow.js ermöglicht die Verwendung verschiedener »Computational Backends« die unter anderem die verwendeten mathematischen Operationen ausführen und als Speicherort für Tensoren dienen. Ein Tensor ist ein mathematisches Objekt aus der linearen Algebra, welches im ML-Bereich häufig als Repräsentation für beispielsweise Bilder oder Videos eingesetzt wird. TensorFlow.js unterstützt unter anderem den Einsatz eines CPU-, WebGL- sowie WebGPU-Backends und kann deshalb in der Untersuchung eingesetzt werden. Das CPU-Backend führt Operationen in nativen JavaScript aus, was aufgrund der »Single-Thread«-Ausführung von JavaScript nicht auf die Erzielung einer hohen Leistung hindeutet. Das WebGL-Backend speichert verwendete Tensoren als Texturen und implementiert die mathematischen Operationen als Fragment Shader. Das WebGPU-Backend nutzt Storage Buffers als Speicherort und setzt zur Berechnung Compute Shader ein. [GooJ]

2.3 BodyPix

Das Computer Vision Modell »BodyPix« kann unter anderem in Kombination mit TensorFlow.js ausgeführt werden und ermöglicht die Segmentierung von Personen und Körperteilen innerhalb des Browsers. Hierbei werden die Pixel eines zu segmentierenden Bildes als Eingabemenge in Form eines zweidimensionalen Arrays verwendet und in semantische Bereiche gruppiert. Im Fall einer Personensegmentierung berechnet das Modell hierfür für jedes Pixel des Eingabearrays ein Wahrscheinlichkeitsmaß welches angibt, mit welcher Wahrscheinlichkeit es zum Körper der Person gehört. Durch einen wählbaren Schwellenwert der zu erreichenden Wahrscheinlichkeit wird ein Ausgabearray (bzw. eine Maske) erzeugt, dass die Zugehörigkeit eines Pixels zur Person binär angibt. Mit der Körperteilsegmentierung

des Modells können bis zu 24 Körperteile einer Person klassifiziert werden. Um dies zu ermöglichen, wird für jedes der 24 trainierten Körperteile ein weiteres Array berechnet, in welchem für jedes Pixel der Eingabemenge die Wahrscheinlichkeit angeben ist, ob es zum Körperteil, welches das Array repräsentiert, gehört. Durch eine Optimierungsfunktion wird ein Ausgabearray erzeugt, welches verschiedene Zahlen enthält, die entweder ein Körperteil oder den Hintergrund repräsentieren [Te19]. BodyPix kann als Aufsatz verstanden werden, welcher auf einem vortrainierten Modell zur »Image Classification« aufbaut. Das von BodyPix unterstützte Modell »MobileNetV1« ist für mobile Endgeräte optimiert und wird aus diesem Grund nicht näher betrachtet. Im Rahmen der Untersuchung soll BodyPix in Kombination mit dem Modell »ResNet50« [He15] verwendet werden.

2.4 Verwandte Arbeiten

Obwohl der erste »API-Proposal« für die heutige WebGPU-API bereits 2017 veröffentlicht wurde [Ap17], konnten keine relevanten Veröffentlichungen identifiziert werden, die WebGPU und WebGL in einem Leistungsvergleich im Kontext einer Computer Vision Anwendung im Browser evaluieren. Ein Grund hierfür könnte die erst kürzliche Integration in den Google Chrome Browser sein. Unbeachtet davon existieren zwei veröffentlichte Paper, die ähnliche Untersuchungen wie in dieser Ausarbeitung angestellt haben. Diese sollen nachfolgend kurz vorgestellt werden.

Ein Forscherteam der Peking University hat eigenen Angaben zufolge 2019 die erste empirische Studie zur Ausführung von ML-Aufgaben (im Original wird der Begriff »Deep Learning« verwendet) im Browser erstellt. Im Rahmen der Studie wurde die Eignung von sieben JavaScript-basierten ML-Frameworks in Bezug auf die Unterstützung von ML-Teilaufgaben im Browser untersucht. Weiterhin wurde die Leistung der Frameworks für die Ausführung eines Modells zur Handschrifterkennung im Browser verglichen. Ebenfalls wurde ein Leistungsvergleich zwischen der Ausführung im Browser mit TensorFlow.js und der Ausführung unter nativen TensorFlow mit Python angestellt. Der Untersuchungsaufbau dieses Papers ist dem Aufbau der geplanten Untersuchung ähnlich, weist im Detail allerdings deutliche Unterschiede auf. Zum einen wurde ein anderer Anwendungsfall untersucht. Weiterhin wurde die Verwendung von WebGPU im Leistungsvergleich der ML-Frameworks nicht berücksichtigt. Generell wird die Nutzung von GPU-Beschleunigung nur beiläufig betrachtet. Der Fokus dieser Ausarbeitung hingegen liegt auf der Realisierung von GPU-Beschleunigung durch WebGL und WebGPU im Browser. [Ma19]

Hidaka et al. haben mit »WebDNN« ein weiteres JavaScript-basiertes Framework zur Ausführung von ML-Aufgaben im Browser implementiert. WebDNN unterstützt wie TensorFlow.js die Nutzung von WebGPU und WebGL zur Integration von GPU-Beschleunigung im Browser. WebDNN wurde im Rahmen eines Papers vorgestellt, welches auch Leistungsmessungen bei eingesetzter GPU-Beschleunigung enthält. Ein Vergleich der erzielten Ergebnisse mit den Ergebnissen der nachfolgenden Untersuchungen bietet sich aufgrund der unterschiedlichen Messumgebung und implementierten Aufgabenstellung nicht an. [Hi17]

3 Untersuchungsaufbau

Für die Planung der Untersuchung der eingangs definierten Fragestellung wurde sich an den Schritten zur systematischen Leistungsanalyse nach Jain orientiert [Ja91]. Abschnitt 3.1 beschreibt die sich aus verschiedenen Komponenten zusammensetzende Versuchsumgebung. In Abschnitt 3.2 wird eine Planung der durchzuführenden Versuche vorgenommen.

3.1 Versuchsumgebung

Versuchsprototyp: Als Versuchsprototyp wird nachfolgend eine Webanwendung zur Segmentierung von Personen und Körperteilen unter Verwendung von GPU-Beschleunigung im Browser verstanden. Für die Implementierung mit JavaScript wurden primär die in Abschnitt 2 beschriebenen Technologien TensorFlow.js und BodyPix verwendet. Um im Rahmen des Leistungsvergleichs die angestrebte GPU-Beschleunigung näher untersuchen zu können, wurde der Versuchsprototyp mit der Möglichkeit ausgestattet, die Berechnungen mit der CPU-, dem WebGL- oder dem WebGPU-Backend durchzuführen. Um die Messergebnisse nicht zu verfälschen, wurde der Einsatz von schwergewichtigen Web-Frameworks wie Angular ausgeschlossen und die Webanwendung aus modularen »Web Components« mit der Bibliothek »Lit« konstruiert.

Benchmark-Prototyp: Neben dem Versuchsprototyp wurde ein Benchmark-Prototyp zur Personensegmentierung implementiert, um Vergleichsmessungen anstellen zu können. Der Benchmark-Prototyp wurde in Python entwickelt und verwendet »TensorFlow« in der nativen Variante in Verbindung mit dem BodyPix-Modell. Die Ausführung des Benchmark-Prototypen erfolgt direkt auf dem Messsystem innerhalb einer Python-Laufzeitumgebung. Neben der Ausführung von Berechnungen auf der CPU wurde die Möglichkeit integriert, diese auf einer Grafikkarte des Herstellers »NVIDIA« unter der Verwendung von »CUDA« durchzuführen.

Messsysteme: Innerhalb der Versuchsumgebung werden ein Notebook und ein Desktop-PC als Messsysteme eingesetzt, welche ein unterschiedliches Leistungsniveau aufweisen. Die im Kontext der Untersuchung relevanten Eigenschaften eines Messsystems werden fortan als »Systemumgebung« bezeichnet und sind in der nachfolgenden Tab. 1 aufgeführt.

	Notebook	Desktop-PC
Modell	Lenovo T580	Eigenbau
CPU	Intel i7-8550U, 4-Cores á 1.80 GHz	Intel i5,-4670K 4-Cores á 3.40 GHz
GPU	NVIDIA GeForce MX150, 2 GB GDDR5	Radeon RX480, 8GB GDDR5
Software	Windows 10 22H2, Chrome 120 Ubuntu 18.04, Python 3.7	Windows 10 22H2, Chrome 120

Tab. 1: Eigenschaften der Messsysteme der Versuchsumgebung

Virtuelle Maschine: Für die Durchführung der Untersuchung wurde weiterhin eine virtuelle Maschine (VM) zur Bereitstellung verschiedener Unterstützungsdienste eingerichtet. Primärer Zweck der virtuellen Maschine ist es, mit einer »CI/CD-Pipeline« das Bauen des Versuchsprototyps durchzuführen und diesen über einen Webserver bereitzustellen. Weiterhin wurde ein Ergebnis-API-Server entwickelt und auf der VM bereitgestellt, um die von den Messsystemen übermittelten Messergebnisse in einer Datenbank zur weiteren Verarbeitung vorzuhalten.

3.2 Versuchsplanung

Für die Durchführung der Untersuchung gelten einige Rahmenbedingungen, die in Tab. 2 aufgelistet sind. Von besonderer Bedeutung ist die Unterscheidung der Versuchsarten »Statische Segmentierung« und »Echtzeit Segmentierung«. Bei der statischen Segmentierung wird für ein einziges Bild eine Personen- oder Körperteilsegmentierung berechnet. Jeder Versuch dieser Versuchsart muss 20 mal wiederholt werden, um eine statistische Verwertbarkeit zu gewährleisten. Bei der Echtzeit Segmentierung wird für ein aufgezeichnetes Video von 60 Sekunden Länge eine Personen- oder Körperteilsegmentierung durchgeführt. Die übrigen Rahmenbedingungen der Tabelle ergeben sich aus dem Kontext der Versuchsart oder wurden bereits zuvor erläutert.

	Statische Segmentierung	Echtzeit Segmentierung
Ausführungsumgebung	TensorFlow.js im Browser *CPU, WebGL, WebGPU*	TensorFlow.js im Browser *CPU, WebGL, WebGPU* TensorFlow mit Python *CPU, GPU mit CUDA*
Wiederholungen bzw. Zeitintervall	20	60 Sekunden
Leistungsmaß	Laufzeit	Frames per Second (FPS)
Image Classification Modell	ResNet50	ResNet50

Tab. 2: Rahmenbedingungen der durchgeführten Untersuchung

Für die Planung der durchzuführenden Versuche wurden im Rahmen eines explorativen Verfahrens Parameter für die Ausführung der Prototypen ermittelt, die versprechen, einen hohen Einfluss auf die Leistung auszuüben. Die Parameter wurden für die Versuchsplanung zu Faktoren mit definierten Faktorstufen spezifiziert, um diese in verschiedenen Versuchsanordnungen zu variieren. Die Faktoren »OutputStride« und »InternalResolution« können eingesetzt werden, um die Genauigkeit bzw. Qualität der Ergebnisse sowie die erreichbare Geschwindigkeit zu steuern. Die Faktoren weisen einen Zusammenhang auf: Die Erzielung einer höheren Genauigkeit geht mit einer Verminderung der Geschwindigkeit einher. Der gegenteilige Zusammenhang gilt auch. Tab. 3 listet die ermittelten Faktoren inkl. Faktorstufen auf.

	Faktor	Faktorstufen
S	Systemumgebung	Notebook, Desktop-PC
A	Aufgabe	Personensegmentierung (A 1), Körperteilsegmentierung (A 2)
OS	OutputStride	(Quality ↑, Speed ↓) 16, 32 (Quality ↓, Speed ↑)
IR	InternalResolution	(Quality ↓, Speed ↑) 0.3, 0.5, 0.7, 1.0 (Quality ↑, Speed ↓)

Tab. 3: Verwendete Faktoren und Faktorstufen

Um den tatsächlichen Einfluss eines Faktors bzw. einer Faktorkombination auf das je nach Versuchsart relevante Leistungsmaß zu bestimmen, wurden verschiedene teilfaktorielle Untersuchungen durchgeführt. Mit der Anfertigung von 2^k-Faktoranalysen konnte der Einfluss der Faktoren bzw. Faktorkombinationen auf das Leistungsmaß ermittelt werden. Tab. 4 listet Faktoren oder Faktorkombinationen aus den Analysen auf, die einen Einfluss von mehr als fünf Prozent in einer Versuchsanordnung aufweisen.

SSx / SST in %	SS{S}	SS{OS}	SS{IR}	SS{S; IR}	SS{OS; IR}
Statisch CPU	0,12%	9,68%	**83,02%**	0,06%	6,95%
Statisch WebGL	**46,96%**	2,10%	**24,59%**	**23,10%**	1,09%
Statisch WebGPU	**76,06%**	2,91%	9,91%	7,59%	0,56%
Echtzeit CPU	-	-	-	-	-
Echtzeit WebGL	**35,63%**	1,71%	**60,31%**	1,87%	0,07%
Echtzeit WebGPU	13,53%	0,90%	**84,94%**	0,46%	0,07%

Tab. 4: Übersicht relevanteste Faktoren und Faktorkombinationen der 2^k-Faktoranalysen

Aus den ermittelten Werten kann geschlussfolgert werden, dass die Systemumgebung und die InternalResolution maßgeblichen Einfluss auf die Variation haben. Weiterhin zeigen die Ergebnisse, dass sich die InternalResolution als Faktor zur Skalierung der Problemgröße eignet. Basierend auf den Erkenntnissen wird eine Versuchsplanung basierend auf der InternalResolution als Problemgröße gewählt. Für jede Stufe der InternalResolution soll eine Versuchsreihe gemessen werden, die alle weiteren Faktorstufen vollfaktoriell kombiniert. Jede Versuchsreihe soll als statische sowie als Echtzeit Segmentierung unter Verwendung aller Computational-Backends im Browser durchgeführt werden.

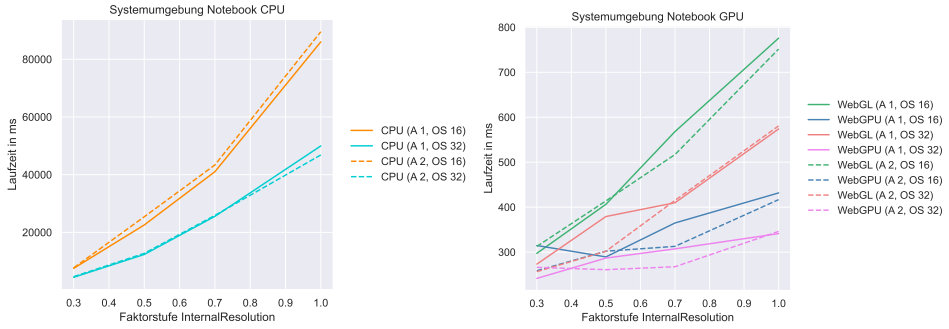
Für den Leistungsvergleich zwischen der TensorFlow.js-Ausführung im Browser und der Ausführung der nativen Version von TensorFlow unter Python gilt nachfolgender Versuchsaufbau: Für den Benchmark-Prototyp wurde nur die Echtzeit Personensegmentierung implementiert, weiterhin ist die InternalResolution seitens des BodyPix-Modells für die native Ausführung auf 0.5 festgesetzt. Die Faktorstufen des Faktors OutputStride sollen auf der CPU sowie auf der GPU mit Verwendung von CUDA gemessen werden. Da CUDA eine NVIDIA-Grafikkarte voraussetzt, kann nur das Notebook als Messsystem verwendet werden.

4 Leistungsvergleich

Für die definierten Versuchsreihen wurden Leistungsmessungen durchgeführt. Die Ergebnisse wurden vom Ergebnis-API-Server bezogen und innerhalb von »Jupyter-Notebooks« ausgewertet [Lo24]. Um Ausreißer zu glätten wurden für die Analysen Durchschnittswerte über die Wiederholungen eines Versuches (Statische Segmentierung) bzw. über den Zeitintervall der Durchführung eines Versuches (Echtzeit Segmentierung) gebildet.

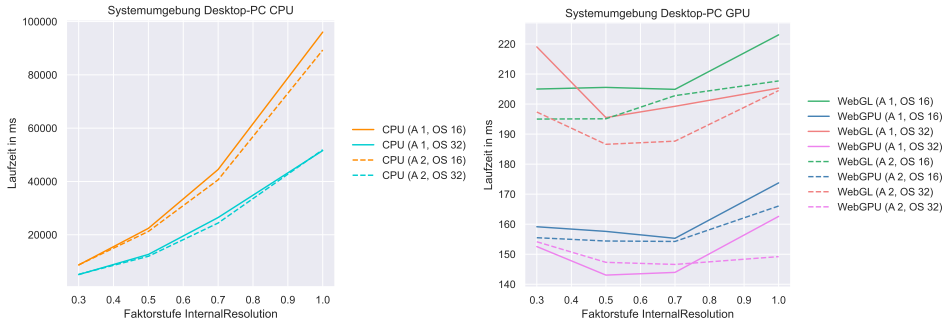
4.1 Statische Personen- und Körperteilsegmentierung im Browser

Abb. 1a, 1b, 2a und 2b visualisieren die Ergebnisse der Leistungsmessungen für die statische Personen- und Körperteilsegmentierung im Browser. Die Darstellung von CPU- und GPU-Ergebnissen der Messsysteme musste aufgrund von zu starker Divergenz des Leistungsmaßes in zwei Diagrammen erfolgen. In den Abbildungen sowie in nachfolgenden Abbildungen werden Abkürzungen aus Tab. 3 für verwendete Faktorstufen eingesetzt.



(a) Erzielte Laufzeit CPU (b) Erzielte Laufzeit GPU

Abb. 1: Ausführung statische Personen- und Körperteilsegmentierung (Notebook)



(a) Erzielte Laufzeit CPU (b) Erzielte Laufzeit GPU

Abb. 2: Ausführung statische Personen- und Körperteilsegmentierung (Desktop-PC)

CPU vs. GPU: Bei einer Betrachtung der Laufzeitkurven für die Versuche auf der Notebook- sowie Desktop-CPU zeigt sich jeweils ein erwartbarer Verlauf, der mit zunehmender Problemgröße ein stetiges Wachstum aufweist. Die Ausführungen auf der GPU weisen teilweise unerwartete Verläufe auf, bei denen die Laufzeit bei steigender Problemgröße gesunken ist. Dieses Phänomen ist in den Versuchen zweimal auf dem Notebook und mehrfach auf dem Desktop-PC aufgetreten. Eine Erklärung für dieses Verhalten konnte nicht gefunden werden.

Bei beiden Messsystemen ergeben sich zwischen CPU- und GPU-Ausführung bezogen auf die Laufzeit große Leistungsunterschiede. Die langsamste Faktorkombination benötigt auf der Notebook-CPU eine Laufzeit von über 80 Sekunden. Auf der CPU des Desktop-PCs wird eine Laufzeit von fast 100 Sekunden benötigt. Verdeutlicht werden die großen Unterschiede, wenn die Ergebnisse in Relation zur Ausführung mit der GPU gesetzt werden. Beide Messsysteme beenden die Berechnungen auf der GPU in unter einer Sekunde. Setzt man die genauen Laufzeiten der CPU- und WebGL-Ausführung ins Verhältnis, ergibt sich für das Notebook ein Avg.-Speedup in Höhe von *59,649*. Für den Desktop-PC beträgt der Avg.-Speedup *158,018*. Die Leistungssteigerungen von der CPU-Ausführung auf die WebGPU-Ausführung fallen noch größer aus: Im Falle des Notebooks wurde ein Avg.-Speedup von *91,893* erreicht, im Fall des Desktop-PCs von *205,103*. Als Schlussfolgerung des Vergleichs lässt sich festhalten, dass die CPU-Ausführung nicht praktikabel für einen Praxiseinsatz ist und eine hohe GPU-Beschleunigung durch die Nutzung von WebGL und WebGPU erreicht werden kann.

WebGL vs. WebGPU: Im Vergleich von WebGL und WebGPU ergeben sich mit steigender Problemgröße auf beiden Messsystemen Laufzeitergebnisse, die belegen, dass WebGPU in der Versuchsreihe schneller als WebGL ist. Dies zeigen auch die ermittelten Beschleunigungswerte. Auf dem Notebook konnte von der WebGL- auf die WebGPU-Ausführung ein Avg.-Speedup von *1,410* erreicht werden, auf dem Desktop-PC von *1,308*.

Der Vergleich der Laufzeitkurven von GPU-Ausführung auf dem Notebook sowie dem Desktop-PC zeigt bei Ausführung auf dem Notebook Kurvenverläufe, die bis auf die bereits genannten Ausreißer nach stetigen Wachstum aussehen. Bei den Versuchen auf dem Desktop-PC stellt sich ein sichtbares Wachstum der Laufzeiten erst ab Faktorstufe 0.7 der InternalResolution ein. Eine Erklärung hierfür könnte die leistungsfähige Grafikkarte des Desktop-PCs sein. Durch die hohe Leistungsfähigkeit könnte die Problemgröße in den niedrigen Faktorstufen zu klein skaliert sein, sodass die Berechnung der kleinen Stufen nicht anspruchsvoll genug ist.

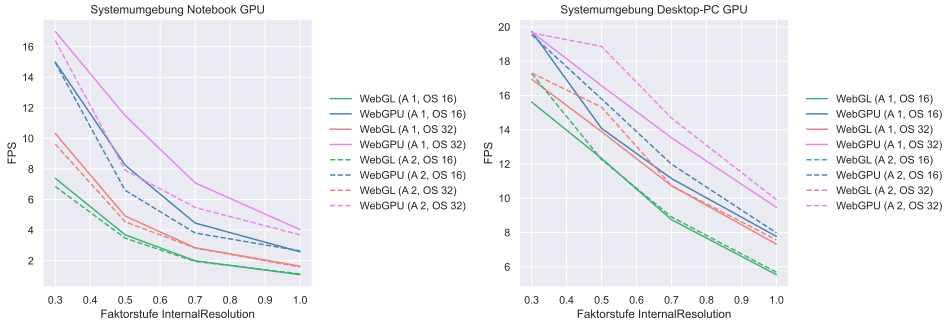
Personen- vs. Körperteilsegmentierung: Ob eine Personen- oder eine Körperteilsegmentierung durchgeführt wird, verursacht auf beiden Messsystemen bei einer CPU-Ausführung keine wesentlichen Laufzeitunterschiede. Die Laufzeiten liegen nah beieinander und teilweise fast gleich auf. Auf beiden Messsystemen zeigt sich bei der GPU-Ausführung eine Volatilität zwischen den Laufzeiten der Personen- und Körperteilsegmentierung. Auch wenn die Körperteilsegmentierung in der Theorie mehr Berechnungsschritte erfordert, spiegelt

sich dies nicht in den Laufzeiten wieder. Häufig nimmt bei der GPU-Ausführung sogar die Personensegmentierung höhere Laufzeiten in Anspruch.

OutputStride: Der Faktor OutputStride verhält sich auf beiden Messsystemen sowie unter CPU- und GPU-Ausführung wie spezifiziert. In einer bis auf die OutputStride identischen Faktorkombination lässt sich die Faktorstufe 32 schneller berechnen als Faktorstufe 16. Weiterhin lässt sich im Durchschnitt bei der GPU-Ausführung beobachten, dass WebGPU die Personen- und Körperteilsegmentierung mit einer leistungsintensiveren OutputStride von 16 schneller ausführt als WebGL mit der schnelleren OutputStride von 32.

4.2 Echtzeit Personen- und Körperteilsegmentierung im Browser

Abb. 3a und 3b zeigen die Ergebnisse der Leistungsmessungen für die Echtzeit Personen- und Körperteilsegmentierung im Browser.



(a) Erzielte FPS Notebook (b) Erzielte FPS Desktop-PC

Abb. 3: Ausführung Echtzeit Personen- und Körperteilsegmentierung

CPU vs. GPU: Es konnten keine Messungen für eine Echtzeit Segmentierung auf einer CPU durchgeführt werden, da auf beiden Messsystemen die Ausführung durch den Browser nach einer bestimmten reaktionslosen Zeit des Prototypen unterbrochen wurde. Als Schlussfolgerung ergibt sich, dass ohne die Nutzung von GPU-Beschleunigung der konkrete Anwendungsfall nicht realisierbar ist.

Auf beiden Messsystemen verhält sich die Ausführung der Echtzeit Segmentierung erwartbar, da die FPS-Anzahl mit steigender Problemkomplexität kontinuierlich sinkt. Ausreißer wie bei der statischen Segmentierung, in der die Leistungsfähigkeit trotz steigender Problemgröße steigt, sind in den Messungen nicht aufgetreten. Durch die deutlich häufiger wiederholte Segmentierung eines Bildes (60 Sekunden * FPS) könnte eine stärkere Glättung von Ausreißern eingetreten sein.

WebGL vs. WebGPU: Auch in dieser Versuchsreihe weist WebGPU eine höhere Leistung als WebGL auf. Bei der Ausführung der identischen Faktorkombination konnte mit WebGPU stets eine höhere FPS-Anzahl erzielt werden. Bei einer Anpassung der Berechnung des Speedups an die Höhe der FPS-Zahl ergibt sich für das Notebook von WebGL auf WebGPU ein Avg.-Speedup von *2,064*. Durch die Verwendung von WebGPU anstatt WebGL wurde auf dem Desktop-PC ein Avg.-Speedup von *1,230* erreicht.

Personen- vs. Körperteilsegmentierung: Die angestellten Messungen auf dem Notebook zeigen, dass die erreichten FPS-Werte für die Echtzeit Personensegmentierung über oder ungefähr gleichauf mit den FPS-Werten der Körperteilsegmentierung liegen. Die Messungen auf dem Desktop-PC zeigen ein gegenteiliges Bild, hier erreicht in vielen Faktorkombinationen die Körperteilsegmentierung eine höhere FPS-Anzahl. Das die Körperteilsegmentierung in der Theorie aufwendiger zu berechnen ist, kann auch durch diese Versuchsreihe nicht bestätigt werden.

OutputStride: Die Messungen der Echtzeit Personen- und Körperteilsegmentierung bestätigen ebenfalls, dass sich der Faktor OutputStride wie spezifiziert verhält. Hervorzuheben ist, dass wie bei der statischen Segmentierung WebGPU mit der leistungsintensiveren OutputStride 16 eine höhere FPS-Anzahl als WebGL mit einer OutputStride von 32 erreicht.

4.3 Echtzeit Personensegmentierung Browser vs. Python

Abb. 4 zeigt die Ergebnisse der Leistungsmessungen auf dem Notebook für den Vergleich der Ausführung einer Echtzeit Personensegmentierung im Browser zu der nativen Ausführung in einer Python Laufzeitumgebung.

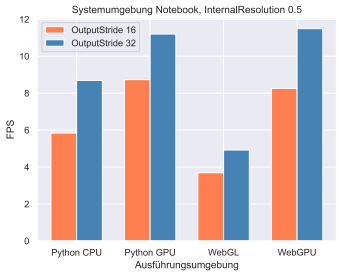


Abb. 4: Ausführung Echtzeit Personensegmentierung im Browser und in Python

Aus den Messungen geht hervor, dass die Ausführung in der Python Laufzeitumgebung mit der CPU als auch unter Verwendung der GPU eine höhere FPS-Anzahl als die GPU-Ausführung mit WebGL im Browser erreicht. Die Ausführung mit WebGPU im Browser konnte dahingegen eine höhere FPS-Anzahl als die Ausführung in Python auf der CPU erreichen. Weiterhin erreicht die WebGPU-Ausführung bei Verwendung einer OutputStride von 32 eine höhere FPS-Anzahl als die Python GPU-Ausführung mit CUDA. Bei einer Verwendung der leistungsintensiveren OutputStride von 16 liegt die mit WebGPU erreichte FPS-Anzahl nur knapp hinter der mit Python und CUDA erreichten FPS-Anzahl. WebGPU erweist sich auch in diesem Versuchsaufbau als schneller als WebGL. Beachtlich ist die Konkurrenzfähigkeit von WebGPU zur Python GPU-Ausführung mit CUDA.

5 Schlussfolgerungen

Im Rahmen der Untersuchung konnte gezeigt werden, dass ein erstellter Prototyp einer Computer Vision Anwendung sich im Browser nicht praxistauglich mit der CPU ausführen lässt. Sowohl bei der Verwendung von WebGL als auch bei der Verwendung von WebGPU innerhalb des Prototypen konnte eine Praxistauglichkeit innerhalb des Browsers festgestellt werden. Leistungsmessungen für den erstellten Prototypen zeigen, dass WebGPU im konkreten Anwendungsfall noch höhere Leistungssteigerungen zur CPU-Ausführung als WebGL ermöglichen kann. Bei einem Vergleich der WebGPU-Ausführung mit der Ausführung eines weiteren Prototypen außerhalb des Browsers lagen die erzielten Ergebnisse vom Leistungsniveau ungefähr gleich auf. Zusammenfassend konnte gezeigt werden, dass sich sowohl WebGL als auch WebGPU zur Nutzung von GPU-beschleunigten Berechnungen im Browser grundsätzlich eignen. Obwohl WebGPU in der vorliegenden Untersuchung eine höhere Leistung als WebGL erzielen konnte, reicht der Untersuchungsumfang längst nicht aus, um eine generelle Empfehlung auszusprechen. Die Beobachtungen dieser Untersuchung können als Anhaltspunkte verwendet werden, um die Übertragbarkeit auf andere Anwendungsfälle zu prüfen. Zukünftige Untersuchungen zu diesem Thema könnten die Nutzung von GPU-Beschleunigung mit WebGL und WebGPU innerhalb des Browsers für weitere Anwendungsfälle aus dem Bereich Computer Vision evaluieren und somit weitere Messdaten ermitteln, die die Aussagekraft der angestellten Untersuchung erhöhen könnten.

Literaturverzeichnis

[Ap17] Apple WebKit Team: WebGPU API Proposal, 2017, URL: https://webkit.org/wp-content/uploads/webgpu-api-proposal.html, Stand: 26. 01. 2024.

[GooJ] Google Brain Team: TensorFlow.js Guide, o.J. URL: https://www.tensorflow.org/js/guide, Stand: 25. 01. 2024.

[He15] He, K.; Zhang, X.; Ren, S.; Sun, J.: Deep Residual Learning for Image Recognition, 2015.

[Hi17] Hidaka, M.; Kikura, Y.; Ushiku, Y.; Harada, T.: WebDNN: Fastest DNN Execution Framework on Web Browser. Proceedings of MM'17, S. 1213–1216, 2017.

[Ja91] Jain, R.: The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. Wiley, New York, 1991.

[LGK23] Lehn, K.; Gotzes, M.; Klawonn, F.: Introduction to Computer Graphics: Using OpenGL and Java. Springer International Publishing und Imprint Springer, Cham, 2023.

[Lo24] Lohmann, N.: jupyter-notebooks-lohmann, 2024, URL: https://git.fh-muenster.de/fep-ws-23-24/jupyter-notebooks-lohmann, Stand: 28. 01. 2024.

[Ma19] Ma, Y.; Xiang, D.; Zheng, S.; Tian, D.; Liu, X.: Moving Deep Learning into Web Browser: How Far Can We Go? Proceedings of the 2019 World Wide Web Conference (WWW'19), S. 1234–1244, 2019.

[Te19] TensorFlow.js: BodyPix: Real-time Person Segmentation in the Browser with TensorFlow.js, 2019, URL: https://blog.tensorflow.org/2019/11/updated-bodypix-2.html, Stand: 25. 01. 2024.

[Wo23] World Economic Forum: Future of Jobs Report: Insight Report, 2023.

Comparison of WebGL and WebGPU as alternatives for implementing GPGPU computing in the browser

Marcus Kligge[1]

Abstract: General purpose computation on GPU (GPGPU) promises high performance for use cases that lend themselves to parallel computing. It could enable companies to move the computational load of their services into the browser on consumer devices and thus result in significant savings. Two alternatives for GPGPU in the browser are WebGL and the currently emerging WebGPU. This paper features a structural, qualitative and performance comparison between the two technologies, additionally considering plain JavaScript, C++ and CUDA for the performance comparison. It offers the insight, that WebGPU might be the more reasonable alternative for GPGPU in the browser, that for some cases can rival CUDA on desktop.

Keywords: GPU Programming, GPGPU, WebGPU, WebGL, WGSL, GLSL, CUDA

1 Introduction

Mainstream business use cases like stock option pricing, and emerging technologies like AI both require substantial computational power. Historically, such computation-heavy tasks are mostly executed on expensive high-performance servers in the backend, as the computational power of CPUs in consumer devices is limited. An approach to increase this power is to utilize GPUs for the execution of highly parallel tasks.

Since integrated or dedicated GPUs are present in almost all consumer PCs and smartphones, the inception of WebGPU, a JavaScript API in the browser that offers capabilities specifically for general purpose computation on GPU (GPGPU), perpetuates the possibility for harnessing the GPUs power without the need for installing an application. This could enable companies to move the computational load of their services to the browser on consumer devices. The diminished need for hardware resources and therefore power consumption in their data centres could result in significant savings.

Besides the new WebGPU API, there is already WebGL as an established API for graphics rendering in the browser, which also offers the possibility for GPGPU computing. This warrants a comparison between the two technologies. Consequently, the goal of this paper is to compare and recommend either WebGL or WebGPU as a reasonable alternative for implementing GPGPU computing in the browser, based on multiple qualitative and quantitative traits. It also serves as an indicator whether GPGPU in the browser is a viable alternative to GPGPU in desktop applications, for example with CUDA.

[1] FH Münster, Fachbereich Wirtschaft, Wirtschaftsinformatik, mk381867@fh-muenster.de

2 General Purpose Computation on Graphics Processing Unit

The high computational power of GPUs stems from the multithreaded Single Instruction Multiple Data (SIMD) model, meaning the same instruction is executed on multiple different data elements simultaneously on different threads. Initially, this was used for graphics rendering, but it can also be exploited for general purpose computation, which defines the essence of GPGPU. This means executing non-graphical computations on the GPU, by writing code and applying data structures in a way that the GPU can interact with them. The GPU hardware handles parallel execution and thread management [He19].

Vertex and fragment shaders. The programs that can run on the GPU are called shaders and are written in shading languages. A typical shader program consists of a vertex shader and a fragment shader. The first shader is responsible for determining the position of each single vertex in a 3D clip space, based on which the rasterizer can interpolate the position of all the grid-aligned fragments in the screen space. The fragment shader code is then invoked once per fragment to determine its colour and depth based on its position, the content of the existing frame buffer, and the specific shader code written by the developer. Each fragment contains information to generate a single pixel in the frame buffer [Ba11]. To leverage this rendering pipeline for GPGPU, data passed to the GPU via textures, which are two-dimensional grids of RGBA values that can be filled with any data, is processed in the fragment shader stage. The fragment position is used as an index to access the data on which the calculations should be performed. If the number of fragments is the same as the number of passed data elements, and each fragment shader call handles a single fragment, then the code is effectively run once for each passed data element. Depending on the use case, the proportions of fragments and data elements can shift, so that one fragment shader call handles two or more data elements. As GPU shaders do not have a mechanism to return the computed values directly to the program that spawned them on the CPU, the results must be read from the targeted frame buffer [Ba11]. One RGBA value is made up of four channels, each channel can represent a distinct value. Instead of outputting RGBA values, where only one channel is filled with actual data, it is possible to compute four input values and generate one tightly packed output RGBA value per execution of the fragment shader code. This is called texture channel packing [Ar23].

Compute shaders. Newer graphics APIs for GPU programming also offer a compute shader stage, which is independent from the vertex and fragment shaders and specifically designed for GPGPU. Instead of executing the fragment shader code for each fragment in the screen space and thus all the passed data, the developer can explicitly specify the number of times the compute shader code is run. This is achieved by setting the number of GPU threads that execute the shader code. For each shader iteration, the iteration number is available as an id inside the program, which can be used as an index to access the data. Instead of wrapping the data in a texture as a necessary abstraction for the fragment shader, the data is passed to the device in a more direct manner using storage buffers. The results of the compute shader can be read from the storage buffer that was provided as output buffer [Ta23].

3 Overview of selected APIs for GPU programming

WebGL. The Web Graphics Library (WebGL) is a JavaScript API exposed by the HTML5 Canvas element for rendering 2D and 3D graphics in the browser without additional plugins. To communicate with the physical GPU, developers must write OpenGL ES 3.0 compliant GLSL shader code [Kh16] and pass it to the device via the JavaScript API. It is developed by The Khronos Group Inc. under the MIT License. Its initial release of version 1.0 was in February 2011, while the version 2.0, which is the main version being focused on and that will henceforth be synonymous with WebGL in this paper, was released in February 2017 [Kh23].

WebGPU. WebGPU is also a JavaScript API for graphics rendering and general computation in the browser. It is exposed by the Navigator object and relies on WGSL shader code as a means of communicating with the physical device. It is developed by the W3C GPU for the Web Community Group under the W3C Software and Document License. It is currently a W3C Working Draft with its initial release in Chrome version 113 in May 2023 [Wc23].

CUDA. The Compute Unified Device Architecture (CUDA) is a proprietary programming interface for general purpose computation by the NVIDIA Corporation and was first introduced in 2006. It is subject to the CUDA Toolkit End User License Agreement. CUDA programs are typically written in C or C++. In contrast to WebGL and WebGPU, CUDA programs are run directly on the desktop instead of in the browser [Nv23].

4 Comparison

4.1 Structural comparison

As seen in Figure 1, the structural design and way of interacting with the physical GPU is quite similar for WebGL and WebGPU, while their implementations are different. In WebGL, a WebGL2RenderingContext as the singular interface for interacting with the GPU is obtained from an HTMLCanvasElement. Each canvas element has one distinct rendering context to separate the WebGL programs from each other. The rendering context is used to create WebGL programs, to pass and compile the shader code, to set the global state, as well as to set attributes and uniform values for a specific render pass. In WebGPU, the navigator object offers an adapter from which the developer can acquire logical devices to access the physical GPU in a compartmentalized way. Each logical device is responsible for one WebGPU program. It is used to pass the WGSL shader code, create buffers, pipelines, and command encoders, as well as pass the recorded GPU commands to the GPU queue. Regarding the general model, WebGL and WebGPU programs both run in applications in the browser. The shader code and all additional data is passed down to their respective backends in the browser which are acting as an abstraction layer to the native GPU API. These native GPU API's process intermediate representations, meaning

Microsofts's Direct3D requires the High-Level Shading Language (HLSL) on Windows, The Khronos Group's Vulkan requires SPIR-V on Windows, Linux, and Android, and lastly Apple's Metal requires the Metal Shading Language (MSL) on macOS and iOS. The WebGL and WebGPU backends translate GLSL and WGSL respectively into these representations. Specifically, the Almost Native Graphics Layer Engine (ANGLE) by Google translates WebGL's GLSL code and the Tint compiler in Google's Dawn translates WebGPU's WGSL code. The translated shader code is passed to the GPU via the aforementioned native GPU API's. Which GPU API it is, is dependent on the specific physical device. The GPU driver compiles the abstraction layer's intermediate representations to another driver-specific intermediate representation, for example DirectX Byte Code (DXBC), which is then compiled to machine code at runtime during execution of the program on the GPU [Mo23].

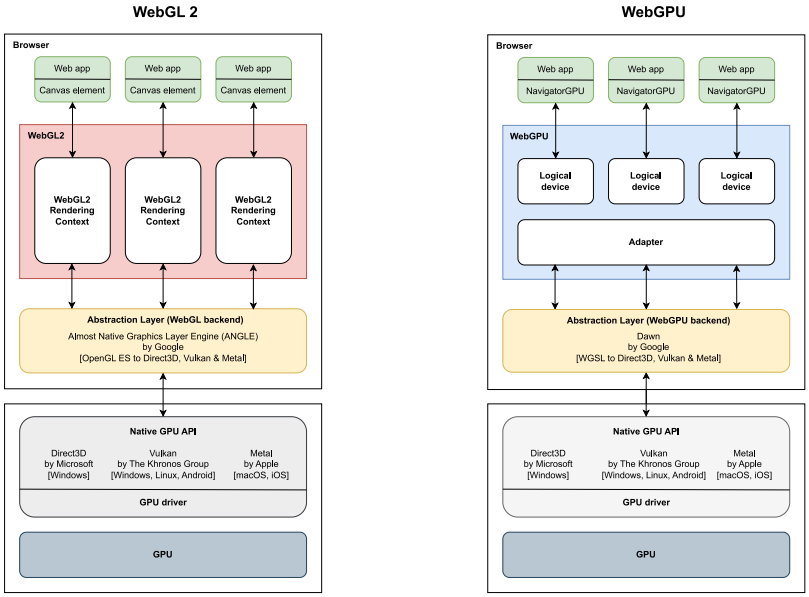


Figure 1: Structural design of WebGL and WebGPU

4.2 Qualitative comparison

Table 1 features a qualitative comparison of WebGL and WebGPU regarding several relevant criteria from the ISO/IEC25010 standard for software quality [Is23]. Each is awarded points from 0 to 10 for later visualisation. The comparison is based on various sources such as specifications, papers, and own implementations of three use cases[2].

[2] https://git.fh-muenster.de/fep-ws-23-24/webgpu-vs-webgl-kligge

	WebGL 2	WebGPU
Functional suitability	Convoluted GPGPU via vertex and fragment shaders. No shared memory across computations, can necessitate algorithm changes. No support for atomic data types. **(6)**	Straightforward GPGPU via compute shaders. Threads in the same block can use shared memory for efficiency. Support for atomic signed and unsigned integer types with built-in read, write and read-modify-write. **(9)**
Flexibility	Available on all major operating systems and browsers on desktop and mobile. Supports older GPU drivers (DirectX 11, native OpenGL). **(10)**	Not available on Linux, Android, or iOS. Only available in Chromium-based browsers on desktop. Requires newer GPU drivers (Vulkan, Metal and Direct3D 11). **(4)**
Reliability	Mature technology that exists since 2011 in version 1 and 2017 in version 2. Developed by established Khronos Group. **(9)**	Emerging technology that just now finds widespread support. Developed by established W3C GPU for the Web Community Group. **(5)**
Maintainability	Officially still a working draft. Few changes to public API or shading language. Low need for maintenance. **(9)**	Active working draft with frequent changes to public API and shading language. High need for maintenance. **(4)**
Resource utilisation	Offers 8-bit, 16-bit, 32-bit and 64-bit numerical data types for good memory efficiency. **(8)**	Offers only 32-bit numerical data types, possibly using more memory than necessary. **(3)**
Interaction capability	**GPGPU.** Indirections like wrapping input data in textures and output data being RGBA values with four channels. Optimizations like texture channel packing necessary. Degree of parallelism is determined by frame buffer size. **(6)** **Collaboration.** Lots of global state that can be affected by accident, bad for collaboration because of potential errors. **(4)** **Documentation.** Little documentation on GPGPU. **(4)** **Community & Ecosystem.** Active community and large ecosystem. Long list of engines, libraries and frameworks. **(8)** **Lines of code.** Verbose, mainly shader compilation and location lookup, buffer creation and setting attributes. **(5)** **Readability.** Negatively impact by need to understand graphics rendering, usage of global state and function names being not self-descriptive. **(5)** **Learnability.** Steep learning curve due to other factors in this list. No real support from IntelliSense because of constants pertaining to the WebGL2RenderingContext. Takes longer to learn than WebGPU. **(4)**	**GPGPU.** Inputs and outputs are storage buffers that are easy to process. Degree of parallelism is determined by workgroup size and number of workgroups, which can be set by the developer. **(10)** **Collaboration.** Stateless API with immutable pipelines and command encoders. Simplifies collaboration and code sharing. **(10)** **Documentation.** Dedicated documentation for GPGPU. Sometimes out of date due to high frequency of changes. **(6)** **Community & Ecosystem.** Active community. Limited but increasing support in libraries and frameworks. **(4)** **Lines of code.** 15% less verbose than WebGL, but still about 8 times more verbose than vanilla JavaScript. **(6)** **Readability.** Easily readable because of self-descriptive function names, explicit passing of data and stateless API function calls. **(8)** **Learnability.** Better learning experience. Good support from IntelliSense and dedicated documentation for GPGPU. **(9)**
Sources	Ar23; Ca23; Is23; Kh16; Kh23; Mo23; Ta23; Wc23	

Table 1: Qualitative comparison of WebGL and WebGPU

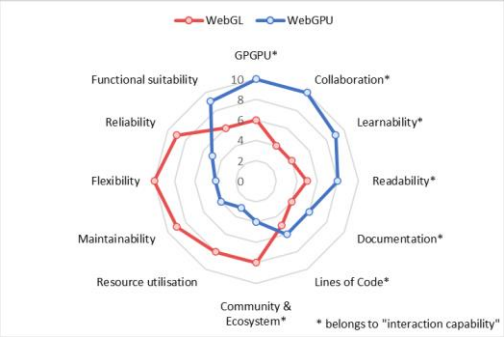


Figure 2: Radar chart of qualitative comparison results

The awarded points are visualised in a radar chart in Figure 2.

Summary. WebGPU has better support for GPGPU through compute shaders and dedicated documentation. The interaction capability of the technology is higher, especially considering the learnability. This can be seen by the fact that the area for interaction capability-related properties is significantly bigger for WebGPU in the radar chart. WebGL, on the other hand, is more mature with less ongoing changes to the API, the shading language, and its documentation. It is also a lot better supported in different operating systems and browsers and requires less memory during execution because of data types with different memory utilisation. It is therefore leading in software quality features such as reliability, flexibility, maintainability, and resource utilisation.

4.3 Performance comparison

Due to the lack of recent performance comparison featuring both WebGL and WebGPU, experiments have been carried out using the software and hardware in Table 2.

Operating System	Windows 10 Home 22H2, Build 19045.3803
Processor	AMD Ryzen 5 4600H, 6 cores, 12 logical processors L1-Cache: 384KB, L2-Cache: 3.0 MB, L3-Cache: 8.0 MB
RAM	16 GB, 3.2 MHz
Integrated GPU	AMD Radeon RX Vega 6 with driver version 31.0.21905.1001 (384 stream processors)
Dedicated GPU	NVIDIA GeForce GTX 1650 with driver version 31.0.15.4612 (896 CUDA cores)
Native GPU driver	Direct3D 11, DirectX 12
Browser	Google Chrome Version 120.0.6099.72 (x64) with V8 JavaScript Engine v12.0.267.8

Table 2: Specification of Lenovo IdeaPad Gaming 3 15ARH05

Use Case	**Variable**
A) i-j-k NxN matrix multiplication	Matrix length
B) Monte Carlo sim.: calculating European call option prices using the Black Scholes model [BL73]	Number of time intervals
C) Monte Carlo sim.: approximating pi	Number of points

Table 3: Implemented use cases

For the experiments, three use cases have been implemented, each with a variable to alter, in order to increase the workload between runs, as seen in Table 3. Each is implemented using the different outlined GPU APIs and JavaScript. For comparison, they are also implemented using C++ and CUDA. Table 4 shows how the experiments are conducted. The GPGPU implementations get run for each GPU device present in that column. For use case A, there are two WebGL implementations, one uses the described texture channel packing and must reorder the results on the CPU, while the other filters the padded zeros from the output on the CPU. After the success in the first experiment, the only WebGL implementation for use case B uses texture channel packing. The use case C does not get evaluated in this paper, as the results from A and B are more relevant, but the implementations for use case C did give a deeper insight into WebGL and WebGPU, so that the knowledge gained from them influence the qualitative comparison in section 4.2.

Name	**Implementation**	**Environment**	**GPU Device**
Native JS	JavaScript V8	Browser	
WebGL	JavaScript V8 & WebGL 2	Browser	AMD, NVIDIA
WebGPU	JavaScript V8 & WebGPU	Browser	AMD, NVIDIA
C++	C++ 17	Desktop	
CUDA	C++ 17 & CUDA 12.3	Desktop	NVIDIA

Table 4: Experiment overview

Metrics. To compare the performance, the CPU runtime, GPU runtime, and total runtime are collected. The GPU runtime is the duration from the call of the GPU, i.e. `drawArrays` or `device.queue.submit`, to reading the results, i.e. `readPixels` or `buffer.mapAsync(GPUMapMode.READ)`. The CPU runtime is the rest of the execution time including context setup, input preparation and result handling, while the total runtime is the sum of CPU runtime and GPU runtime. Resource cleanup after shader execution, as well as creating the arrays filled with random numbers for the matrix multiplication are not included in the runtime. The duration is measured in milliseconds using `Date.now()` in JavaScript, or `chrono::high_resolution_clock::now()` in C++.

Automation. For the browser-based implementations, there is a web application, where parameters can be set, and experiments run any number of times. For this, a start and end value for the variable from Table 3 is set, the range is divided into a set number of steps, and the experiment is executed with the same input parameters for a set number of iterations per step. The results are collected in an array that can be downloaded as a csv file. The execution of the desktop-based implementations is automated via batch scripts that create a csv file with the results. For all the experiments, code that is not necessary for execution but could influence the runtime is omitted. The collected results are analysed in a Jupyter Notebook. For each run, a regression function is calculated to visualize the results in relation to the other runs. The use case implementations, measurements and the evaluation notebook are available in the accompanying GitLab repository.

Matrix multiplication. Figure 3 shows the regression functions for all runtimes of all implementations of use case A. The implementations show large differences in runtime.

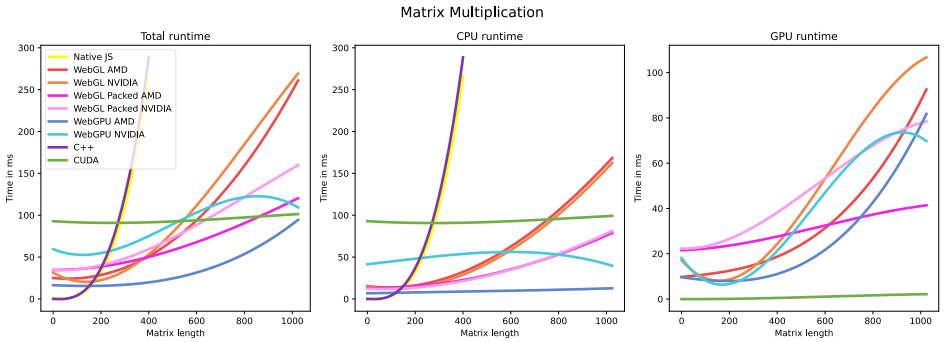


Figure 3: Matrix multiplication results

GPU vs. CPU. The implementations without GPU support, Native JS and C++, are exceptionally fast for small matrices with a matrix length up to 150 but are a lot slower for matrix lengths over 300. They are faster for small matrices, because they do not need to communicate with the native GPU API and transfer data between CPU and GPU. This overhead from communication and data transfer can be seen in the CPU runtime plot of the GPU supported implementations. For a 1x1 matrix, they show an overhead between 6 and 92 ms. However, the CPU-based implementations cannot keep up with the parallel computing power of the GPU for larger matrices.

Packed vs. Unpacked. The WebGL implementations that use texture channel packing and reorder the results on the CPU have a different total runtime than those which filter the padded space from the results on the CPU. The former have a higher initial GPU runtime, because they synchronously compute four values instead of just one. However, their GPU runtime does not increase as rapidly as for the non-texture-packed implementations, because they only need to compute quarter the number of fragment shaders. This means for larger matrices the implementations using texture channel packing are faster. The difference in GPU runtime for larger matrices would suggest that computing a single fragment shader that calculates four values synchronously is faster than computing four fragment shaders in parallel. A reason for this could be that the overhead of creating three new threads takes a longer duration than synchronously executing three more values on the existing thread.

AMD vs. NVIDIA. There are differences in runtime behaviour for the same code between the two tested GPUs. For WebGPU, the AMD GPU is faster until matrix length 1060, from where NVIDIA is faster due to its irregular behaviour, where the runtime goes down after matrix length 854. This behaviour can be seen in Figure 4 with AMD on the left, where the total runtime has a standard deviation of 21.3ms, and NVIDIA on the right, where the total runtime has a standard deviation of 28.4ms. There are two main possibilities for the irregularities around matrix length 854 on NVIDIA: Firstly, the AMD Ryzen 5 4600H processor has an 8MB L3 cache, which can fit two input matrices and one output matrix of 32-bit values each with matrix length 816. Higher matrix lengths would

cause communication with the main memory which would be slower and rather negatively affect the runtime. Secondly, the GeForce GTX 1650 has 896 CUDA cores and therefore can process 896 threads simultaneously. However, not all CUDA cores are identical and so do not have the same speed and latency. Using almost all available CUDA cores around matrix length 854 could mean using slower cores as well, while every CUDA core that is used for the second iteration of threads after 896 is probably a fast core again, which is why the runtime drops. That said, this does not happen for the WebGL implementation and therefore might me related to WebGPU internals. For WebGL, the AMD GPU is also faster and has less deviation than the NVIDIA GPU. A possible reason why AMD is faster, even though it can only process 384 threads simultaneously, while NVIDIA manages 896, is that the AMD GPU is integrated in the processor and therefore has a closer physical proximity and thus lower latency.

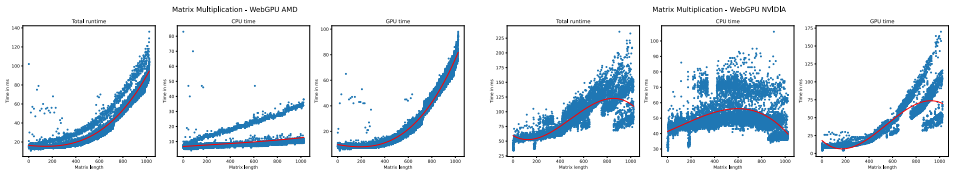


Figure 4: Runtime behaviour WebGPU AMD (left) vs. NVIDIA (right)

Browser vs. Desktop. When comparing WebGL and WebGPU as browser-based implementations to CUDA as a desktop-based alternative, the former perform better for smaller matrix lengths up to around 500. This is due to a large CPU runtime overhead of CUDA, which is 92 ms for a matrix length of 1, while the overhead for WebGL and WebGPU is between 6 ms and 41 ms. However, the CPU runtime of CUDA only barely rises with increased workload. The GPU runtime of CUDA is the lowest out of all implementations with only 2 to 3 ms for 1024x1024 matrices, which also only barely rises, while the browser-based implementations show a quadratic increase in duration. The larger initial overhead in CUDA could be due to the initialization of the GPU and the launching of the kernel, which is the executable function on the GPU. While the browser-based implementations also must communicate with the GPU, the connection is already made because the graphical interface of the browser itself is already powered by the NVIDIA GPU in "high performance mode". As such, there is no additional overhead for initialization upon starting the matrix multiplication execution. In contrast, the reason why CUDA is much faster for larger matrix lengths is its conformity to the hardware. Both CUDA and the GPU it uses are made by NVIDIA and their collaboration is therefore optimized. On the other hand, WebGL and WebGPU use a more generic approach and their shader code is translated through multiple intermediary representations, as seen in Figure 1. This translated code is likely not optimized for the specific NVIDIA hardware and thus results in higher runtimes.

WebGL vs. WebGPU. WebGPU is faster than WebGL without texture channel packing on both the NVIDIA and the AMD GPU. The reason for this is the difference in CPU runtime because of the result filtering on the CPU that WebGL must do to achieve the

same tightly packed results as WebGPU. Apart from the observed anomaly for WebGPU NVIDIA, the GPU runtime of WebGL and WebGPU is similar with only milliseconds difference. For the WebGL Packed implementations, the CPU runtime is higher than for WebGPU, because the WebGL implementations using texture channel packing must reorder the results on the CPU. However, their incline of the GPU runtime is significantly lower than for the WebGPU AMD implementation, ignoring the WebGPU NVIDIA anomaly. This is why ultimately the WebGPU implementations perform better than the WebGL implementations for matrix lengths up to 1231, the intersection of the regression lines of WebGPU AMD and WebGL Packed AMD. For a larger workload, WebGL is faster but only if the developer employs optimization techniques like texture channel packing. The reason for this was described in the "Packed vs. Unpacked" section.

Summary. For small workloads up to matrix length 157, the implementations without GPU support are fastest, because they have no overhead for communication with the physical GPU. For medium workloads up to matrix length 1057, WebGPU on AMD has the lowest total runtime, as the GPU initialization overhead is the lowest out of all GPU supported implementations. When ignoring the WebGPU NVIDIA anomalies, CUDA is fastest for large workloads with matrix lengths over 1057, because of its optimization for the NVIDIA hardware. If a desktop-based implementation is not feasible, the optimized WebGL implementation using texture channel packing is the fastest from a matrix length of 1231 onward. This is likely because synchronously computing four values on the same thread is faster than creating three additional threads.

Monte Carlo Simulation for stock option pricing. Figure 5 shows the regression functions for all runtimes of all implementations of use case B. As these are approximations of the actual experiment results, some lines in the graphs start below zero on the y-axis.

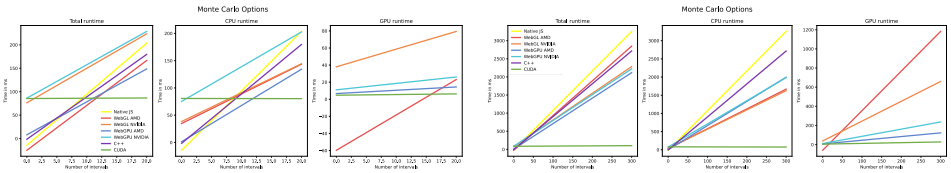


Figure 5: Monte Carlo Simulation for stock option pricing results
(left: intervals 0 – 20, right: intervals 0 – 300)

GPU vs. CPU. In general, the difference between the implementations with GPU support and the ones without is smaller than for the matrix multiplication. The former tend to be faster than the implementations without GPU support, except for WebGL on AMD, which is slightly slower than C++. A possible reason is that the Monte Carlo Simulation heavily relies on random numbers. CUDA is a lot faster than the others because the `<curand>` random generator of CUDA is a lot faster than `Math.random` or `<random>` in JavaScript and C++. For 100 million random numbers, CUDA takes 275 ms while JavaScript needs 1650 ms and C++ 2900 ms. CUDA is also significantly faster than the implemented

random number generator function using Box-Muller-Transform for WebGL and WebGPU, which do not offer built-in random functions. Box-Muller-Transform takes about 600 ms for 1 million random numbers. The browser-based GPU implementations do benefit from their parallel computing power, but the slower random number generation results in their runtimes being close to or even slower than the CPU-based approaches.

AMD vs. NVIDIA. Same as for the matrix multiplication, the standard deviation of the NVIDIA experiments is higher. The initial communication overhead for one interval is 90 ms for the NVIDIA experiments, which can be seen in Figure 5 on the left. Otherwise, the runtime difference is not as evident as for the matrix multiplication, because the NVIDIA experiments are faster than those performed on AMD in WebGL but slower in WebGPU.

Browser vs. Desktop. It is obvious that CUDA is by far the fastest implementation. A big influence is the described runtime difference for random number generation. Same as for the matrix multiplication, CUDA has a 90 ms communication overhead at the start, but is its runtime again barely rises. The CPU runtime for CUDA also barely rises. While WebGPU and WebGL must create and manage large JavaScript arrays, which is slow and affects the CPU runtime, CUDA benefits from its fast low-level approach.

WebGL vs. WebGPU. WebGPU is faster than WebGL on both AMD and NVIDIA. Especially the difference in runtime to the WebGL AMD experiment is significant, however it is not evident why that is, as the WebGPU AMD experiment is the fastest out of the four. It could suggest, that WebGL is more performant on AMD GPUs, which would contradict the results from the matrix multiplication experiments.

Table 5 shows the different speedups resulting from experiments of use cases A and B.

Use case	**A) 200**	**A) 800**	**B) 300**
JavaScript	1.00	1.00	1.00
WebGL AMD	1.12 (0.83 packed)	13.57 (23.97 packed)	1.14
WebGL NVIDIA	1.42 (0.80 packed)	11.72 (17.83 packed)	1.43
WebGPU AMD	2.04	39.94	1.54
WebGPU NVIDIA	0.58	17.83	1.47
C++	0.87	0.96	1.20
CUDA	0.35	22.24	31.59

Table 5: Speedup comparison for different experiments and workloads

5 Conclusion

Structurally, WebGL and WebGPU are very similar, both translating their shader code to intermediary representations that can be used by the native GPU API. Qualitatively, both have different strengths. WebGL is more mature and has widespread support in operating systems, browsers, and libraries, while WebGPU is easier to learn and implement because of dedicated GPGPU-related documentation and a stateless and explicit API with good

IntelliSense support. Performance-wise, it comes down to the specific use case. Generally, for a use case that is well-suited for parallel computation, WebGL and WebGPU can both offer an immense speedup compared to JavaScript in the browser, especially for larger workloads. On average, WebGPU performs better than WebGL without additional optimizations. In reference to the goal of this paper, WebGPU could be the more reasonable alternative for GPGPU computing in the browser because of better interaction capability, learnability, and out-of-the-box performance. It is a viable alternative to CUDA on desktop for certain applications and could enable companies to move the computational load of their services to consumer devices. Future research in this topic may offer deeper insights into suitability for different GPU alternatives and explain runtime anomalies.

6 References

[Ar23] Arm Limited: Real-time 3D Art Best Practices; Texture channel packing. 2023. URL: developer.arm.com/documentation/102449/0200/Texture-channel-packing.

[Ba11] Bao, H.; Hua, W.: Basics of Real-Time Rendering. In: Advanced Topics in Science and Technology in China; Real-Time Graphics Rendering Engine, pp. 7 – 20. 2011.

[Bl73] Black, F.; Scholes, M.: The Pricing of Options and Corporate Liabilities. In: Journal of Political Economy 81, No. 3, pp. 637 – 654. The University of Chicago Press. 1973.

[Ca23] Can I use… Authors: Browser support tables for modern web technologies. 2023. URL: caniuse.com/webgl2; caniuse.com/webgpu.

[He19] Hennessy, John L.; Patterson, David A.: Computer Architecture. A Quantitative Approach. 6th Edition, Morgan Kaufmann, 2019.

[Is23] ISO International Organization for Standardization: ISO/IEC 25010:2023 (E) - Systems and software engineering - System and software quality models. Geneva. 2023.

[Kh16] The Khronos Group Inc.: GLSL Specification Version 3.00. 2016. URL: registry.khronos.org/OpenGL/specs/es/3.0/GLSL_ES_Specification_3.00.pdf.

[Kh23] The Khronos Group Inc.: WebGL 2.0 Specification. Edited by Jackson, Dean; Gilbert, Jeff. 2023. URL: registry.khronos.org/webgl/specs/latest/2.0/.

[Mo23] Mozilla Corporation: MDN Web Docs – Web APIs. 2023. URL: developer.mozilla.org/en-US/docs/Web/API.

[Nv23] NVIDIA Corporation & affiliates: CUDA Toolkit Documentation. 2023. URL: docs.nvidia.com/cuda/.

[Ta23] Tavares, Gregg: WebGPU Fundamentals. 2023. URL: webgpufundamentals.org/webgpu/lessons/webgpu-fundamentals.html.

[Wc23] W3C GPU for the Web Working Group: WebGPU & WGSL Specification. 2023. URL: w3.org/TR/2024/WD-webgpu-20240125/; w3.org/TR/2024/WD-WGSL-20240126/.

All referenced URLs have been last visited on 27.01.2024.

Vergleich von Java Vector API, Foreign Functions & Memory API und Java Native Interface anhand einer Matrixmultiplikation und der Monte Carlo Optionspreisberechnung

Jannis Theile[1]

Abstract: Um rechenintensive Aufgaben zu beschleunigen ist es nötig, CPU-Architekturen und deren Ressourcen möglichst optimal zu verwenden. SIMD-Instruktionen bieten eine effektive Nutzung der vorhandenen CPU-Ressourcen. Dieses Paper beschreibt einen Vergleich zwischen den Technologien Java Vector API, Foreign Functions & Memory API und des Java Native Interface. Verglichen werden die grundlegenden Strukturen, qualitativen Merkmale und die Performance der Ansätze unter zusätzlicher Berücksichtigung ihrer Nutzung von SIMD-Instruktionen. Dabei werden die Eignung und die Leistungsfähigkeit anhand einer Matrixmultiplikation sowie der Berechnung von Monte Carlo Optionspreisen überprüft und bewertet. Das Paper bietet einen Einblick in die Leistungsfähigkeit der Java Vector API und legt dar, dass diese nicht nur bestehende SIMD-Implementierung ersetzen, sondern auch durch ihren Einsatz den Overhead der Kommunikation mit externen Bibliotheken überflüssig machen kann. Gleichzeitig ist kritisch zu hinterfragen, ob eine Neuimplementierung zugunsten des Performancegewinns gerechtfertigt oder die externe Anbindung durch die Foreign Functions & Memory API mit deutlich geringeren Implementierungsaufwand verbunden ist.

Keywords: SIMD, Java Vector API, Foreign Functions & Memory API, Java Native Interface

1 Einleitung

Hardwarearchitekturen in Form von Server- und Clienthardware werden stetig weiterentwickelt, um den Anforderungen für rechenintensive Fragestellungen gerecht zu werden. Zu den aktuellen Themen gehören rechenintensive Machine Learning Modelle, Big-Data Analysen oder aufwendige mathematische Berechnungen. Dabei fällt auf, dass insbesondere interpretierte Sprachen nicht mit der rasanten Weiterentwicklung mithalten können. Interpretierte Sprachen wie z.B. Java haben sich durch Abstraktionsschichten immer weiter von der Hardware entfernt und schöpfen nicht das volle Potenzial der CPU-Architekturen aus. Um Vektoroperationen aus Java heraus explizit und nicht nur durch Auto-Vektorisierung zu nutzen, mussten Entwickler auf Integrationsschnittstellen systemnaher Sprachen in systemferne Hochsprachen zurückgreifen. Dies konnte z.B. mit dem Java Native Interface umgesetzt werden. Dies hat allerdings zur Folge, dass die Komplexität der Implementierung steigt und ein Overhead bei der Kommunikation zwischen den Plattformen entsteht. Um diesen Anforderungen zu begegnen, wurde die

[1] FH Münster, Fachbereich Wirtschaft, Wirtschaftsinformatik, jt999045@fh-muenster.de

Java Vector API eingeführt, mit der es möglich sein soll, plattformunabhängige Vektorberechnungen durchzuführen. Daneben gibt es mit der Foreign Functions & Memory API eine weitere neue Möglichkeit native Bibliotheken und Speicher außerhalb der Java Virtual Machine zu reservieren, um SIMD-Instruktionen hardwarenah zu nutzen. Bei der Recherche wurden keine Papers gefunden, welche einen Vergleich dieser Technologien beschreiben. Lediglich Analysen zur Performance z.B. zur Java Vector API oder dem Java Native Interface existieren. Bei der Java Vector API wird der Performanceunterschied in Java zwischen der Java Vector API, der Auto-Vektorisierung und der Implementierung ohne Nutzung von Vektorisierung beschrieben [Ba23]. Des Weiteren gibt es ein Paper, welches einen Performancevergleich zwischen Java und dem Java Native Interface durchführt [Ha23].

2 SIMD

Bereits in den 1970er gab es die Idee in einem einzelnen Befehlsstrom mehrere Datenströme zu verarbeiten. Dabei kann SIMD (Single Instruction, Multiple Data) in einem Taktzyklus der CPU mehrere Datenströme mit dem gleichen Befehl ausführen [He14]. Dadurch konnte die Parallelität auf Befehlsebene erhöht werden. Gerade bei rechenintensiven wie z.B. mathematischen Berechnungen bietet SIMD eine erhebliche Steigerung der Verarbeitungsgeschwindigkeit. Weiterhin werden die CPU-Ressourcen effizienter genutzt, wodurch die CPU-Last verringert werden kann. Mit der Einführung von Advanced Vector Extensions (AVX) wurden neben 256-Bit Registern auch neue arithmetische Funktionen hinzugefügt. Damit die Verarbeitung von Daten effektiv ist, ist es Voraussetzung, dass eine große Menge strukturierter Daten vorliegt [He14]. Viele Programmiersprachen und Plattformen unterstützen die Verwendung von SIMD-Operationen. In C++ können Entwickler AVX-Intrinsics verwenden, welche spezielle Funktionen bereitstellen, um SIMD-Register und Instruktionen gezielt ansprechen zu können. In diesem Paper wird die Vector ISA (Instruction Set Architecture) AVX2 betrachtet, da dieses auf der vorliegenden Hardware verfügbar ist. Dabei lässt sich AVX2 in die Kategorien Arithmetik, Tauschoperationen, Laden von Daten, Speichern von Daten, bitweise Operationen, Vergleichsoperationen, Logische Operationen, kryptografische Operationen, String Operationen, Zufallsoperationen, Statistische Operationen und Umwandlungsoperationen einteilen. In der verwendeten ISA AVX2 stehen somit 191 Funktionen zur Verarbeitung von Daten zur Verfügung. [In23]

3 Überblick von Java Vector API, Foreign Functions & Memory API und Java Native Interface

Java Vector API. Die Java Vector API wurde erstmals als Inkubator-Version in Java 16 eingeführt. Die Technologie verspricht eine native Unterstützung für das Verwenden von Operationen auf Vektorregistern. Eine Plattformunabhängigkeit ist gegeben, sodass Implementierungen auf den CPU-Architekturen x86-64 und AArch64 umsetzbar sind. Mit

der Java Vector API wird nicht das Ziel verfolgt, bestehende Auto-Vektorisierung-Algorithmen zu verbessern. Ein Vector wird in der Java Vector API als abstrakte Klasse *Vector<E>* dargestellt. Die Kombination aus Elementtyp und Form stellt die Vektorspezies dar, welche die Vektorenlänge abbildet. Dabei kann der Vector z.B. mit einem ganzzahligen Elementtyp instanziiert werden. Je nach CPU-Architektur werden Vektorgrößen von 64, 128, 256 und 512-Bit unterstützt. Angenommen die CPU-Architektur bietet einen 256-Bit großen Shape an, so können 8 Integer-Werte a 32-Bit auf 8 Lanes gleichzeitig verarbeitet werden. Weiterhin können Operationen auf Teilmengen angewendet werden, indem die Eingabewerte durch eine boolesche Maskierung gefiltert werden. Die Vektoroperationen lassen sich in lane-wise Operationen und cross-lane Operationen unterteilen. Dabei wird bei der lane-wise Operation eine skalare Operation auf eine einzelne Spur in einem oder mehreren Vektoren durchgeführt. Bei der cross-lane Operation wie z.B. beim Sortieren werden hingegen spurenübergreifende Operationen verwendet. [Op23] [Or23a]

Foreign Functions & Memory API. Die Foreign Functions & Memory API aus Java 21 erlaubt es, nativen Code wie C oder C++ außerhalb der Java Laufzeitumgebung aufzurufen und ist eine Alternative zum Java Native Interface. Die API verspricht eine einfachere und sicherere Integration von nativem Code und ermöglicht Operationen auf dem OFF-Heap Speicher durchzuführen. Um Speicher außerhalb der JVM zu reservieren, wird eine Arena definiert, welche den Lebenszyklus des allokierten Speichers definiert und ebenfalls die Anzahl der Threads festlegt, die auf die erstellten Speichersegmente zugreifen dürfen. Dadurch kann die Lebensdauer eines Speichersegmentes bestimmt werden. In dieser Arena können Speichersegmente erstellt werden, welche die Struktur des allokierten Speichers darstellen. Dabei bildet ein Speichersegment eine Abstraktion eines dahinterliegenden Speicherbereiches ab. Um eine fremde Methode einer C++ Bibliothek aufzurufen, wird ein FunctionDescriptor verwendet, durch welchen die Methodensignatur repräsentiert wird. Dabei können einfache Datenprimitive oder komplexe Datenstrukturen als Speichersegmente übergeben werden. [Of23] [Or23b]

Java Native Interface. Das Java Native Interface ermöglicht den Aufruf von Funktionen, die in C oder C++ geschrieben sind. Dabei bietet JNI Mechanismen an, um Datentypen zu konvertieren und Daten an die Methode zu übergeben. Es ist eine explizite Verwaltung des Speichers für native Methoden nötig, um welche sich der Entwickler kümmern muss. Java Native Interface spezifische Methoden sind über Interface-Pointer erreichbar. Dabei werden Java-Typen wie z.B. double auf den nativen Typ jdouble abgebildet und die Daten kopiert. Java-Objekte hingegen werden per Referenz übergeben. Damit Objekte, die innerhalb des nativen Codes nicht vorzeitig von dem Garbage Collector freigegeben werden, kennt die JVM alle Objekte, die an den nativen Code übergeben wurden. Dabei wird zwischen lokalen und globalen Referenzen unterschieden. Lokale Referenzen sind für die Dauer des nativen Aufrufs gültig, während globale Referenzen explizit freigegeben werden müssen. Weiterhin ist es möglich javaseitig definierte Methoden zu suchen und aus der externen Bibliothek aufzurufen. Dabei werden ebenfalls spezielle Typsignaturen verwendet, um z.B. den Rückgabetyp zu definieren. [Ki23] [Or93]

4 Vergleich

4.1 Struktureller Vergleich

Da die Java Vector API in Java implementiert ist, werden keine externen C/C++ Bibliotheken benötigt, um SIMD-Operationen auszuführen. Deshalb wird nur ein struktureller Vergleich der Foreign Functions & Memory API sowie dem Java Native Interface vorgenommen.

Foreign Functions & Memory API vs Java Native Interface. Zunächst ist festzuhalten, dass bei dem Java Native Interface ein natives Programmiermodell verfolgt wird und somit die Speicherallokation auf C++-seitig erfolgt. Des Weiteren muss zunächst aus einer Java-Klasse und deren Methode eine Header-Datei generiert werden muss. Die generierte Header-Datei gibt die Methodensignatur an, welche implementiert werden muss. Aus den Implementierungen wird eine Bibliothek erstellt, welche in Java geladen wird. Danach können die nativen Methoden aufgerufen werden.

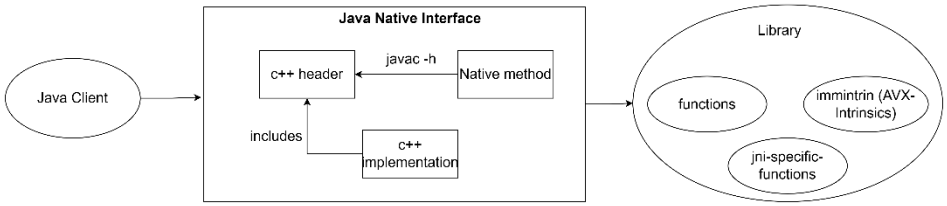


Abbildung 1: Aufbau Java Native Interface

Im Gegensatz zum Java Native Interface wird keine Deklaration der nativen Methode benötigt. Es wird ein reines Java-Entwicklungsmodell verfolgt. Dadurch erfolgt die Speicherallokation und Deallokation auf der Seite von Java. So können die reservierten Speichersegmente an die auszuführende Methode übergeben werden, welche dann die AVX-Intrinsics aufrufen. Es werden keine weiteren spezifischen Funktionen benötigt.

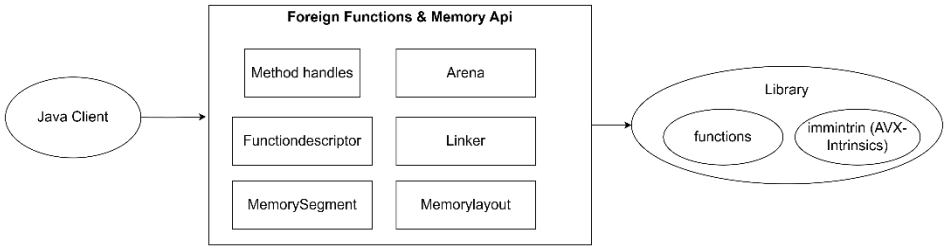


Abbildung 2: Aufbau Foreign Functions & Memory API

4.2 Qualitativer Vergleich

Um den qualitativen Vergleich von JVA, FFI und JNI durchzuführen, werden folgende Kriterien betrachtet. Dabei geht es vor allem darum, die Unterschiede kritisch aufzuzeigen. Der Vergleich basiert auf Dokumentationen, Artikeln und der eigenen Erfahrung bei der Umsetzung von verschiedenen Szenarien und deren Implementierung.

Funktionalität. Zur Funktionalität lässt sich festhalten, dass alle drei Ansätze Möglichkeiten bieten, unterschiedliche Anwendungsfälle umzusetzen. Es kann bei allen drei Ansätzen auf gängige Datentypen zurückgegriffen werden, sodass die Nutzung von verschiedenen AVX-Intrinsics möglich ist. Einige Besonderheiten sind dennoch zu beachten. Es entsteht zusätzlicher Aufwand, um komplexere Datenstrukturen, wie z.B. verschachtelte Arrays zu implementieren, da die AVX-Intrinsics nur mit eindimensionalen Datenstrukturen umgehen können. Die Abbildung solcher Strukturen ist javaseitig mit einem geringeren Implementierungsaufwand als in C++ verbunden.

Stabilität. Da bei der Java Vector Api das Java-Typsystem und die automatische Speicherverwaltung durch den Garbage Collector verwendet wird, gilt JVA als sehr stabil. FFI bietet im Vergleich zu JNI eine bessere Stabilität aufgrund der Typsicherheit auf Java-Seite. Ebenso kann der Garbage Collector aus Java bei der Verwaltung von Speicher zum Einsatz kommen, falls die Arena als automatische Arena definiert wurde. Bei JNI hingegen muss der Speicher mit JNI-spezifischen Methoden selbst verwaltet werden, sodass die Stabilität negativ beeinflusst werden kann. Da bei der Foreign Functions & Memory Api ebenfalls ein javaseitiges Programmiermodel vorliegt, ist eine bessere Stabilität aufgrund der geringeren Fehleranfälligkeit bei der Allokation des Speichers gegeben. Zusätzlich kommt es jedoch auf die Kenntnisse des Entwicklers im Bereich der Speicherallokation und Speicherdeallokation an.

Benutzbarkeit. Bei der Benutzerbarkeit werden Aspekte wie Lesbarkeit, Dokumentation und die Lines of Code dargestellt. Zur Lesbarkeit ist zu sagen, dass mit allen drei Ansätzen eine gute Lesbarkeit gewährleistet werden kann. Bei JNI ist es jedoch die Verwendung vieler komplexer Datentypen mit Mehraufwand verbunden. Da die Java Vector Api und die Foreign Memory Api noch relativ jung sind, gibt es im Vergleich zum Java Native Interface weniger Dokumentationen und Anwendungsbeispiele. Die Dokumente die verfügbar sind, bieten jedoch eine klare Struktur und eine detaillierte Beschreibung der verfügbaren Funktionen. Anwendungsbeispiele gibt es jedoch nur wenige. Betrachtet man die unterschiedlichen Implementierungsansätze, ist zu erwarten, dass das Ansprechen von externen Bibliotheken mit einem höheren Implementierungsaufwand verbunden ist. Dies belegt die Implementierung der Matrixmultiplikation und die Optionspreisberechnung. Hierbei fällt auf, dass je nach Anwendungsfall ebenso die Implementierung mit der Java Vector API im Vergleich zu Java ohne Vektorisierung zu einem höheren Implementierungsaufwand führt. Dies ist dadurch zu erklären, dass der Entwickler sich zunächst um das Laden und Speichern der Daten kümmern muss und ebenso um die verbleibenden Elemente, welche nicht durch die AVX-Intrinsics verarbeitet werden

können. Vergleicht man JNI und FFI, so ist erkennbar, dass sich die Lines of Code ähneln. Unterschiedlich ist jedoch die Verteilung der LOC in C++ und Java, da das Reservieren des Speichers bei FFI im Java Code stattfindet, während bei JNI das Allokieren des Speichers in C++ durchgeführt wird. Bei JNI wird in Java lediglich die Methode definiert und die Bibliothek geladen. Abhängig vom Kenntnisstand des Entwicklers in C++, kann das Allokieren von Speicher bei FFI den Aufwand im Vergleich zu JNI verringern. Für die Nutzung von AVX2-Operationen in C++ ist es ebenso notwendig, eine 32-Byte Ausrichtung zur gewährleisten, damit die Daten optimal in den Vektorregistern verarbeitet werden können [Un23]. Dies erhöht z.B. bei der Matrixmultiplikation in C++ mithilfe der Foreign Functions & Memory Api die Komplexität, da Anpassungen nötig sind, um javaseitig den Speicher mit der passenden Ausrichtung zu allokieren. Es ist festzuhalten, dass die Java Vector API je nach Anwendungsfall ca. die doppelte Anzahl Lines of Code benötigt. Beim Vergleich von FFI und JNI fällt auf, dass das javaseitige Programmiermodell bei FFI dafür sorgt, dass der C++ Code reduziert und der Java-Code erhöht wird.

Implementierung (LOC)	Java ohne JVA	JVA	FFI		JNI	
Programmiersprache	Java	Java	Java	C++	Java	C++
Matrixmultiplikation	7	13	19	16	3	33
Optionspreisberechnung	14	29	6	38	3	40

Tabelle 1: Implementierungen LOC

Portierbarkeit. Zu der Portierbarkeit lässt sich sagen, dass die Java Vector Api aufgrund der Plattformunabhängigkeit der JVM mit wenig Aufwand portierbar ist. Bei FFI und JNI sind zusätzliche Schritte und plattformspezifischer Code nötig, um die Portierbarkeit zu gewährleisten. Bezogen auf die AVX-Intrinsics sind zudem die CPU-Architekturen zu prüfen, sofern externe Bibliotheken angesprochen werden. Bei der Verwendung der Java Vector Api wird, falls die benötigte CPU-Architektur nicht vorhanden ist, eine gewisse Abwärtskompatibilität bereitgestellt. Das bedeutet, dass das Programm weiterhin ausführbar ist, jedoch im Hintergrund nicht die optimalen SIMD-Instruktionen nutzen kann.

Einsatzgebiete. Die Einsatzgebiete der drei Technologien sind sehr vielfältig, wobei FFI und JNI ein ähnliches Ziel und somit auch ähnliche Einsatzzwecke verfolgen. JVA bietet sich für rechenintensive, wissenschaftliche und mathematische Berechnungen sowie für Datenanalysen an. JNI und FFI eignen sich für das Anbinden von externen Systemen und Bibliotheken sowie hardwarespezifische Funktionen zu nutzen, welche in Java nicht verfügbar sind. Für FFI ist es ebenso möglich, GPU-Runtimes anzusprechen und dort den Speicher zu allokieren, um anspruchsvolle Berechnungen ebenso auf der GPU durchzuführen.

Wartung. Wird die Wartung der Technologien verglichen, so ist zu erkennen, dass die JVA einfacher zu warten ist als JNI und die FFI, da keine externen Bibliotheken angesprochen werden. Aufgrund der spezifischen JNI-Operationen, die in der externen Bibliothek verwendet werden, ist der Wartungsaufwand im Vergleich zu FFI höher, da FFI keine JNI-spezifischen Methoden besitzt, sondern auf Java-Methode zurückgreifen kann.

4.3 Performancevergleich

Der Performancevergleich wurde auf der Hardware durchgeführt, welche in Tabelle 2 spezifiziert wurde. Dabei ist besonders darauf zu achten, dass die CPU-Flags den Wert AVX2 besitzen, damit die CPU-Architektur die SIMD-Instruktionen für 256-Bit Vektorregister unterstützt. Die Cache-Grenzen wurden ebenfalls bei den Messungen der Performance berücksichtigt. Die Messungen wurden mit dem Java Microbenchmark Harness durchgeführt. Dabei wurde pro Messung jeweils eine Warmup-Iteration vor den anschließenden Iterationen durchgeführt.

Gerät	HP ProBook x360 440 G1
Betriebssystem	Ubuntu 18.04.6 LTS
Prozessor	Intel® Core™ i7-8550U 1.80GHz 4 Kerne 8 logische Prozessoren
RAM	16 GB
Integrierte GPU	Intel UHD Graphics 620
Relevante CPU-Flags	sse, sse2, sse3, sse4, sse4, fma, avx, avx2
Caches	L1: 64KB, L2:256KB, L3: 8192KB – Gesamt: 8512KB

Tabelle 2: Hardwarespezifikationen

Um die Performance der Technologien zu vergleichen, wurden zunächst die grundlegenden Aufrufgeschwindigkeiten miteinander verglichen. Dazu wurden Messungen der Downcall und Upcall Aufrufe beim Java Native Interface und der Foreign Functions & Memory API durchgeführt. Bei einem Downcall handelt es sich um einen Aufruf einer Funktion aus einer externen Bibliothek. Bei einem Upcall wird hingegen zunächst eine Funktion einer externen Bibliothek aufgerufen, die dann eine übergebene Funktion aus Java aufruft. Für den Downcall-Aufruf wird die Übertragungsgeschwindigkeit eines Arrays mit Elementen des Typs Double gemessen. Sobald die Daten im Speicher vorliegen, wird die Messung beendet. Für das Messen des Upcalls wurde der Quicksort-Algorithmus implementiert. Der Upcall ruft eine native Java-Methode auf, welche zwei Zahlen miteinander vergleicht. Dazu müssen bei FFI zunächst Elemente aus den an Java übergebenen Speichersegmenten ausgelesen werden, bevor die Double-Elemente miteinander verglichen werden können. In einem weiteren Schritt wurden anwendungsnähere Szenarien betrachtet. Zum Vergleich der Performance werden die ijk-Matrixmultiplikation und die Durchführung der Monte Carlo Simulation von

Optionspreisen anhand des Black Scholes Modells herangezogen [Bl73]. Bei der Matrixmultiplikation wurde dabei die Größe der Matrix pro Durchlauf variiert. Für die Performancemessungen wurden nur Matrizen der Größe NxN genutzt. Bei der Optionspreisberechnung wird die Anzahl der Simulation anhand der Pfade pro Messung erhöht. Die externen C++-Bibliotheken, welche durch FFI und JNI aufgerufen werden, wurden dabei mit dem GCC-Compiler und den Compileroptionen -std=c++17 mavx2 -shared erstellt, sodass die AVX2-Instruktionen des Prozessors unterstützt werden.

Downcall. Um das Java Native Interface und die Foreign Functions & Memory Api grundlegend zu vergleichen wurde die Dauer der Übertragungszeit an die externe Bibliothek gemessen. Maximal wurden 20.Mio Elemente vom Typ Double übertragen, sodass die Übertragungszeit bei ca. 100ms lag. Es ist festzuhalten, dass sich die Performance des Downcalls laut Abbildung 3. bei beiden Technologien ähnelt.

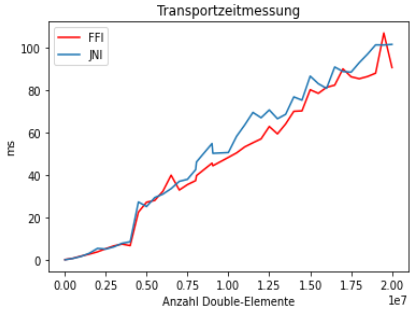


Abbildung 3: Messung der Übertragungszeit von Arrays vom Typ Double

Upcall. In Abbildung 4. wurde der Upcall von JNI und FFI unter Benutzung des Quicksort-Algorithmus verglichen. In Abb. 4 ist ersichtlich, dass JNI bis zu einer zu sortierenden Arraygröße von ca. 12500 Elemente performanter ist als FFI. Ab diesem Schnittpunkt ist FFI mit einem Speed-Up von ca. 1,5 schneller bei der Sortierung als JNI, sodass sich Upcall-Szenarien performanter mit FFI umsetzen lassen.

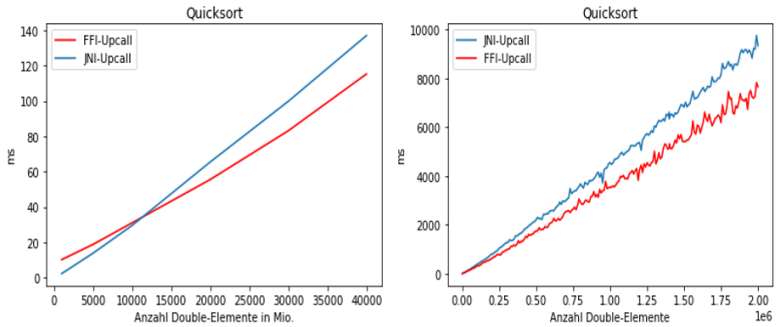


Abbildung 4: Vergleich des Upcalls für Quicksort bei JNI & FFI

Optionspreisberechnung. Nachdem die grundlegenden Aufrufzeiten von FFI und JNI beschrieben wurde, wird nun ein Monte Carlo Simulation zur Optionspreisberechnung als Anwendungsszenario betrachtet. Hierbei wurden Implementierungen mit der Java Vector Api, ohne die Java Vector Api und mit dem Aufruf von AVX2-Intrinsics aus JNI sowie FFI umgesetzt. Die Optionspreisberechnung wurde mit Float- und Double-Elementen mit einer maximalen Anzahl von einer Million Pfadberechnungen durchgeführt. Im rechten Teil der Abbildung 5 ist zu sehen, dass es bei der nativen Implementierung ohne die Java Vector Api kein Laufzeitunterschied ersichtlich ist. Dabei ist es gleichgültig, ob die Implementierung mit Float- oder Double-Elemente betrachtet wird. Bei der Nutzung von AVX2-Intrinsics auf Java oder C++-Seite kann durch Verwendung des Float-Datentyps im Vergleich zum Double-Datentyp die doppelte Menge an Rechenoperationen in einem CPU-Zyklus durchgeführt werden, da acht Float-Elemente à 4 Byte gleichzeitig in einem 256-Bit Vektorregister verarbeitet werden können. Wie schon in Abbildung 3 zu sehen war, ist die Aufrufzeit bei JNI und FFI ähnlich. Dieses Verhalten ist auch im linken Teil der Abb. 5 zu erkennen, da JNI und FFI ca. 1250ms für die Durchführung der Optionspreisberechnung benötigen. Dadurch kann ein Speedup für JNI von 1,4x und für FFI von 1,3x im Vergleich zur Implementierung ohne AVX-Intrinsics in Java erreicht werden. Ebenso ist zu erkennen, dass JVA einen Speedup von 4x beim Datentyp double sowie von 8x beim Datentyp float erreicht. Verglichen zu JNI und FFI ist die JVA ca. um das 2,6-fache schneller. Die ist auf die Aufruf- und Übertragungszeit sowie die Vorbereitungszeit der Daten zurückzuführen. Zusammengefasst lässt sich beschreiben, dass sich rechenintensive Berechnungen sehr performant über die Java Vector Api ausführen lassen, aber auch das Aufrufen einer externen Bibliothek unter Verwendung von SIMD-Instruktionen zu einer Beschleunigung beitragen kann. Die initiale Startzeit liegt bei der Foreign Functions & Memory Api mit 0,11ms am höchsten. Danach gleicht sich jedoch die Performance an die vom Java Native Interface an.

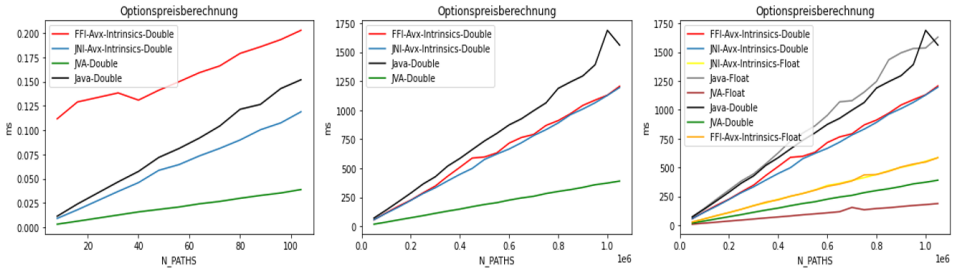


Abbildung 5: Vergleich Optionspreisberechnung

Matrixmultiplikation. Bei der Matrixmultiplikation wurden nur Matrixgrößen NxN betrachtet die die gleiche Anzahl an Zeilen und Spalten besitzen. Es wurden Matrixmultiplikationen mit den Datentypen Float und Double implementiert, um zu zeigen, ob die Speedup`s im Vergleich zur Optionspreisberechnung anhand des Datentyps erklärbar und auf weitere Anwendungsfälle übertragbar sind. Dabei wurde ebenfalls die Cache-Grenze ermittelt und in rot eingefärbt. Die Cache-Grenze ergibt sich aus zwei

Eingabematrizen und einer Ausgabematrix der Größe NxN sowie dem Bytefaktor für Double(8-Byte) und Float-Elemente(4-Byte). So liegt die Cache-Grenze für den Matrizen des Typs Double auf der vorliegenden Hardware bei 595 und für den Typ Float bei 842. Betrachtet man zunächst Abb. 6, so ist zu erkennen, dass die Foreign Functions & Memory Api die höchste initiale Startzeit mit ca. 0,08 Millisekunden besitzt. Auffällig ist, dass die Implementierung in Java ohne die Vector API schneller ist als die Foreign Functions & Memory API sowie das Java Native Interface. Dies kann zwei Gründe haben, die für eine endgültige Klärung weitere Messungen und Untersuchungen notwendig machen. Zum einen kann die Performance durch das Übertragen von 3 Matrizen und das Vorbereiten des Speichers verlangsamt werden und zum anderen kann die Matrixmultiplikation besonders gut von der Auto-Vektorisierung des Compilers verarbeitet werden. In der Abb.6 mit den Matrixgrößen bis 400 lässt sich ein Speedup durch die Java Vector Api im Vergleich zu Java ohne die Vector Api von 2,4x festhalten.

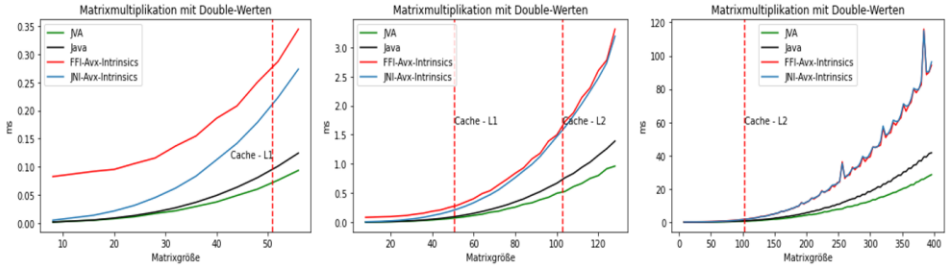


Abbildung 6: Matrixmultiplikation für Matrixgrößen im Intervall von 4 bis 400

In Abb.7 wurden Matrixmultiplikationen im Intervall von 400-700 durchgeführt. Dabei ist zu erkennen, dass die Umsetzungen mit der Foreign Functions & Memory Api und des Java Native Interface langsamer sind als die Implementierungen mit und ohne Vector Api. Generell zeigen sich keine neuen Erkenntnisse bei der Betrachtung des Datentyps float. Zu Erkennen ist jedoch, dass bei dem Ansprechen externer Bibliotheken vermehrt Unregelmäßigkeiten bei den Messungen auftauchen. Wird die Cachegrenze in den Ergebnissen von Abb.7 betrachtet, sind keine außergewöhnlichen Sprünge aufgrund der längeren Zugriffszeiten in den RAM zu sehen.

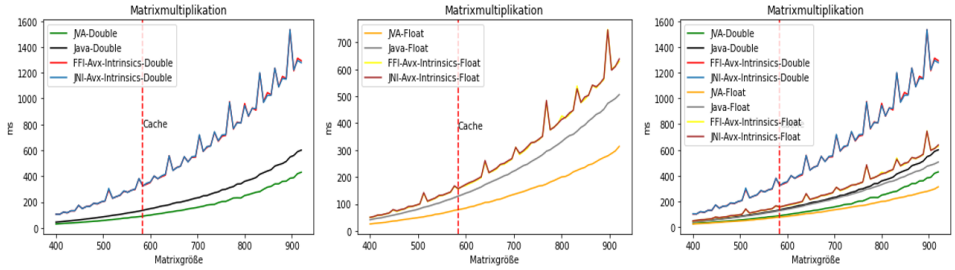


Abbildung 7: Matrixmultiplikation für Matrixgrößen im Intervall von 400-700

Nachfolgend sind in Tabelle 3 die Speedup´s der unterschiedlichen Fragestellungen und Datentypen verglichen anhand der Implementierung in Java ohne die explizite Nutzung der AVX-Intrinsics zusammengefasst.

Szenario	Matrixmultiplikation		Optionspreisberechnung	
	Double	Float	Double	Float
Java ohne JVA	1	1	1	1
Java Vector API	1,3	1,6	4	8
Java Native Interface	0,4	0,8	1,4	2,8
Foreign Functions & Memory API	0,4	0,8	1,3	2,7

Tabelle 3: Speedups

5 Fazit

In dieser Arbeit wurden ein Strukturvergleich, Qualitätsvergleich und Performancevergleich durchgeführt. Der Strukturvergleich zeigt, dass bei der Foreign Functions & Memory Api ein Java-First Programmiermodell verfolgt wird, sodass das Speichermanagement in Java stattfindet. Bei JNI liegt ein Native-First Programmiermodell vor, sodass die JNI-spezifischen Funktionen in C++ für die Verwaltung von Speicher genutzt werden. Der Qualitätsvergleich zeigt auf, dass sowohl die Java Vector Api als auch die Foreign Functions & Memory Api und das Java Native Interface die Ausführung effizienter SIMD-Instruktionen ermöglichen. Je nach Anwendungsfall ist jedoch durch FFI und JNI zusätzlicher Implementierungsaufwand nötig, um die externen Bibliotheken, welche SIMD-Instruktionen nutzen, anzusprechen. Die Kommunikation mit einer externen Bibliothek sollte zukünftig über die Foreign Functions & Memory Api realisiert werden, da diese im Vergleich zu JNI eine bessere Stabilität, Flexibilität, Wartbarkeit sowie ein besseres Programmiermodell zur Speicherverwaltung bieten. Anhand des Performancevergleichs ist zu erkennen, dass keine generelle Aussage zum Performancegewinn bei der Anbindung externer Bibliotheken getroffen werden kann. Bei der Matrixmultiplikation ist die Anbindung einer externen Bibliothek mit SIMD-Instruktionen langsamer als die Implementierung in Java ohne die Java Vector Api. Lediglich die Java Vector API führt hier zu einer Performancesteigerung. Bei der Monte Carlo Simulation für Optionspreise lohnt sich hingegen eine Anbindung der externen Bibliothek, da Speedup´s von 1,4x bis 2,8x erreichbar sind. Ebenfalls bietet die Java Vector Api einen Speedup von 4x – 8x. Die Java Vector Api kann die Performance für viele Anwendungsgebiete steigern. Es lässt sich festhalten, dass sich insbesondere rechenintensive mathematische Berechnungen anbieten. Gleichzeitig muss jedoch auch der Implementierungsaufwand betrachtet werden, um abzuwägen, ob eine Neuimplementierung der externen Bibliothek gerechtfertigt ist. Falls nicht kann, eine Anbindung mithilfe des Jextract-Tools über die Foreign Functions & Memory Api erfolgen, welche die Speicherverwaltung in Java anhand einer C++-Header-

Datei generieren kann. Abschließend ist festzuhalten, dass eine Performancesteigerung immer mit einem höheren Implementierungsaufwand und Wartungsaufwand verbunden ist. Zudem müssen Entwickler sich mit Plattformspezifika vertraut machen, um für bestimmte Anwendungsfälle die optimale Performance zu gewährleisten. Jedoch können durch diesen Mehraufwand die vorhandenen CPU-Ressourcen, durch die Nutzung von SIMD-Instruktionen, optimal verwendet werden.

6 References

[Ba23] Basso Matteo, Rosa Andrea, Omini Luca, Binder Walter. Java Vector API: Benchmarking and Performance Analysis. 2023. In Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction (CC 2023). Association for Computing Machinery, New York, NY, USA, 1–12.

[Ha23] Nassim Halli, Henri-Pierre Charles, Jean-François Méhaut. Performance comparison between Java and JNI for optimal implementation of computational micro-kernels. ADAPT 2015.: The 5th International Workshop on Adaptive Self-tuning Computing Systems, Jan 2015, Amsterdam, Netherlands. URL: https://hal.science/hal-01277940

[He14] Patterson, David A.; Hennessy, John L.: Rechnerorganisation und Rechnerentwurf. Die Hardware/Software- Schnittstelle. 5. Auflage. De Gruyter Oldenbourg, 2014.

[In23] Intel. URL: https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-9/intrinsics-for-avx2.html (zuletzt abgerufen am 04.01.2023)

[Un23] University of Alaska Fairbanks. URL: https://www.cs.uaf.edu/courses/cs441/notes/avx/ (zuletzt abgerufen am 20.01.2024)

[Ki23] Sharan, Kishori.; Beginning Java 8 APIs, Extensions and Libraries.: Swing, JavaFX, JavaScript, JDBC and Network Programming APIs, 2014.

[Of23] OpenJDK: JEP 442 Foreign Functions & Memory API. 2023. URL: https://openjdk.org/jeps/442. (zuletzt abgerufen am 30.12.2023)

[Or23a] Oracle. URL: https://download.java.net/java/early_access/loom/docs/api/jdk.incubator.vector/jdk/incubator/vector/Vector.html (zuletzt abgerufen am 28.12.2023)

[Or23b] Oracle. Foreign Functions & Memory API URL: https://docs.oracle.com/en/java/javase/21/core/foreign-function-and-memory-api.html#GUID-FBE990DA-C356-46E8-9109-C75567849BA8. (zuletzt abgerufen am 30.12.2023)

[Or93] Oracle. Java Native Interface URL.: https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/types.html. (zuletzt abgerufen am 20.01.2024)

[Ov23] OpenJDK: JEP 448 Vector API (Sixth Incubator). 2023. URL: https://openjdk.org/jeps/448. (zuletzt abgerufen am 28.12.2023)