Neue Mikroarchitektur für zukünftige CPUs

SIMD-Vektor-Verarbeitung mit ARMv9

11.06.2021 | Autor / Redakteur: Frank Riemenschneider * / Michael Eckstein

Arm hat mit ARMv9 eine neue Mikroarchitektur vorgestellt, die einige Verbesserungen gegenüber Armv8 verspricht. Dazu zählt die Standardisierung der sogenannten Scalable Vector Extension 2 (SVE 2). Nicht nur für KI-Anwendungen könnten diese von großem Nutzen sein.

Scalable Vector Extension 2 (SVE 2): Für Entwickler wird das Schreiben und Optimieren von Arm-Code einfacher, insbesondere auch bei Anwendungen für maschinelles Lernen, Spracherkennung u.s.w.
(Bild: gemeinfrei / Pixabay)

Die SIMD-Verarbeitung, also das Verarbeiten von mehreren Daten gleichzeitig durch eine Instruktion, wurde ursprünglich speziell für wissenschaftliche High-Performance-Computing-Workloads entwickelt. Intel und AMD waren hier sehr erfolgreich, zumal maschinelles Lernen und Datenanalyse in der Wirtschaft immer mehr zum Mainstream werden und an Bedeutung gewinnen. Die SIMD-Befehlssätze von Intel entwickelten sich mit der Zeit immer weiter, von MMX über SSE2 und AVX bis zum aktuellen Stand AVX-512 mit 512 Bit breiten Vektor-Registern.

Die Architektur Armv8 wurde vor acht Jahren veröffentlicht und war die erste 64-Bit-Architektur von Arm. CPUs, die auf der ARMv7-Architektur oder vorangegangenen Releases basierten, konnten lediglich 32-Bit-Werte verarbeiten, der letzte 32-Bit-Prozessor war der Cortex-A15.

Arm implementierte die sogenannten Neon-SIMD-Erweiterungen, deren Register jedoch auf eine fixe Breite von 128 Bit limitiert sind. Dann kam der A64FX-Prozessor von Fujitsu. Die Japaner wollten einen Supercomputer auf Arm-Basis bauen, für dessen Fähigkeiten die Vektorverarbeitung extrem wichtig ist. Fujitsu ging in der Folge eine Partnerschaft mit Arm ein, um Arm-Prozessoren um einen neuen Befehlssatz namens SVE, Scalable Vector Extension, zu erweitern.

Im Januar 2016 wurde dann die Architektur Armv8.2-A angekündigt: Die mit Fujitsu entwickelte Scalable Vector Extension (SVE) ist eine optionale Erweiterung der Armv8.2-A-Architektur (und folgender), die es Chip-Designern erlaubt, mit Arm-CPUs variable Vektorlängen von 128 bis 2048 Bit zu implementieren. Die Erweiterung ist komplementär zu den Neon-Erweiterungen und ersetzt diese nicht.

Dies ist insofern wichtig, als dass der für den Fugaku-Supercomputer, der nach einem Upgrade jetzt 7.630.848 Cores beinhaltet, entwickelte Code z. B. auf Armv8-MPUs von NXP oder Renesas u. a. nicht laufen würde, weil ihnen einfach die Unterstützung für SVE-Befehle fehlt.

SVE wird zum Standard

In ARMv9 hingegen wird diese Art von Instruktionen Teil des Standards. Ein großer Teil von Arms Erfolgsstory besteht darin, dass die Firma Prozessoren für die billigsten Embedded-Geräte bis hin zu den teuersten Super-Computern anbieten kann, die alle in der Lage sind, den gleichen Code auszuführen.

So etwas ist mit den Lösungen, die Intel und AMD mit den AVX-Befehlen für ihre x86-Mikroprozessoren verfolgt haben, nicht möglich.

Technisch gesehen sind sowohl Arms Neon- als auch SVE-Befehle eine Form von SIMD-Befehlen. Das ist eine Abkürzung, die für Single Instruction Multiple Data steht. Die Idee von SIMD ist also, dass die CPU eine einzige Anweisung abarbeitet und diese dann dieselbe Operation auf mehreren Werten ausführt, und zwar alle gleichzeitig. Es gibt

jedoch einen ganz entscheidenen Vorteil von SVE, nämlich der, für den das „S“ steht: skalierbar.

Ein typischer Neon-Befehl, das mit 32 jeweils 128 Bit breiten Vektor-Registern V0 bis V31 arbeitet, sieht wie folgt aus:

ADD v2.16B, v0.16B, v1.16B
oder

ADD v2.2D, v0.2D, v1.2D

Dort gibt es ein auf den ersten Blick merkwürdiges Suffix wie .16B oder 2D hinter jedem Registernamen. Der Grund dafür ist einfach zu verstehen:

128 Bit sind die Obergrenze dafür, wie viele Zahlen parallel bearbeitet werden können und aus wie vielen Bits jede dieser Zahlen bestehen kann. Wenn man z. B. 64-Bit-Zahlen addieren möchte, ist man auf 2 Zahlen limitiert, bei 8-Bit-Werten können gleichzeitig 16 Zahlen verarbeitet werden. Während das Suffix .16B bedeutet, dass die CPU den 128-Bit-Registerinhalt als 16 Zahlen à 8 Bit betrachten soll, steht 2D für ein Doppelwort, d. h. die CPU nimmt zwei 64-Bit-Zahlen in dem Vektorregister an.

Beim Laden und Speichern eines Vektorregisters braucht man die Suffixe natürlich nicht, weil ja jedesmal die vollständigen 128 Bit geladen bzw. gespeichert werden und es dabei egal ist, aus wie vielen Elementen diese 128 Bit zusammengesetzt sind.

Bild 1: Vektoraddition mit 4 Lanes, d. h. es werden vier Berechnungen parallel ausgeführt.
(Bild: SEGGER)

Die Anzahl der Elemente, in die ein Register während einer Berechnung aufgeteilt ist, entscheidet, wie viele sogenannte „Lanes“ für Berechnungen eingerichtet werden. Jede Lane benötigt folgerichtig auf dem Chip eine eigene ALU (Arithmetisch-Logische-Einheit) der CPU. Bild 1 zeigt ein Beispiel für die Berechnung mit 4 Lanes, d. h. vier 32-Bit-Werte aus den Register V8 und V9 werden im Rahmen einer Instruktion (mit dem Suffix „.4S“ für „Single Word“ = 32 Bit) addiert und die Ergebnisse in das Register V10 geschrieben.

Das große Problem mit SIMD-Anweisungen

Wie oben gesehen, können unterschiedliche SIMD-Befehle wie eigentlich derselbe Befehl jedoch mit unterschiedlichen Argumenten aussehen. Diese werden jedoch im Befehlssatz als separate Anweisungen kodiert. Intel begann mit MMX, dann kamen SSE, SSE2, AVX, AVX2 und schließlich AVX-512. MMX hatte z. B. 64-Bit-Vektorregister, so dass man z. B. zwei 32-Bit-Werte oder acht 8-Bit-Werte parallel verarbeiten konnte. Mit der Zeit, als mehr Transistoren zur Verfügung standen, entschied man sich, neue und größere Vektorregister zur Verfügung zu stellen. SSE2 hat z. B. 128-Bit-Vektorregister. Irgendwann war das nicht mehr genug, und so kam AVX und AVX2 brachte 256-Bit-Vektorregister. Und jetzt ist gerade AVX-512 mit den 512 Bit breiten Vektorregistern Stand der Technik.

Jedes Mal, wenn Intel größere Register zur Verfügung stellte, musste der Befehlssatz um zig neue Instruktionen ergänzt werden, weil die Länge des Vektorregisters in der SIMD-Anweisung kodiert ist. Man braucht also z. B. eine ADD-Anweisung nicht nur für jede Vektorregisterbreite 64, 128, 256 oder 512 Bit, sondern jede davon braucht auch noch eine eigene Variante für die Anzahl der verwendeten Lanes.

Die SIMD-Befehle im x86-Universum haben also zu einer Explosion der Anzahl der Instruktionen im Befehlssatz geführt, und natürlich unterstützt nicht jeder x86-Prozessor alle diese Befehle: Nur die neueren CPUs unterstützen z. B. AVX-512. Auf einem Core-i-Prozessor vor der Skylake-Generation wird entsprechender Code nicht funktionieren.

Diese Strategie ergibt für Arm keinen Sinn, da man ein sehr breites Spektrum an Anwendungen abdecken muss, von winzigen Embedded-Geräten bis hin zu Super-Computern wie dem Fugaku. Eine CPU mit 512 Bit breiten Vektoren (oder noch breiteren) ergibt für Supercomputer Sinn, aber nicht für MPUs, die in Steuergeräten im Auto zum Einsatz kommen. Alleine der Energiebedarf der Transistoren, die für die Implementierung notwendig wären, und die Kosten der Silizumfläche würden die Kunden verzweifeln lassen. Theoretisch könnte Arm unterschiedliche Befehlssätze zur Verfügung stellen. Dies würde aber Arms Geschäftsmodell, dass dieselbe Software auf einer Vielzahl der Chips funktioniert, zuwiderlaufen.

Variable Vektorlänge ist die Lösung für Arm

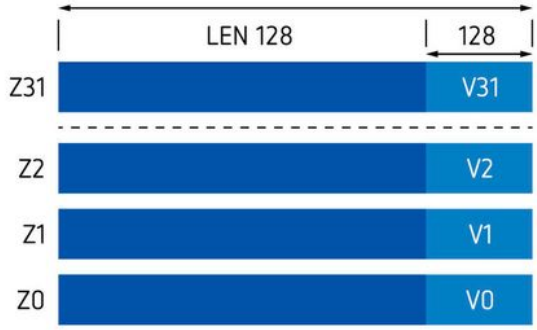

Bild 2: Die 32 SVE-Vektorregister können flexibel vom Chip-Designer zwischen 128 und 2048 bit Länge definiert werden.
(Bild: Arm)

Was SVE und SVE2 ermöglichen, ist die unterschiedliche physikalische Länge von Vektorregistern für jeden Chiptyp, für den CPU-IP verkauft wird. Mit SVE/SVE2 muss jedes der 32 Vektorregister Z0 bis Z31 eine minimale Länge von 128 Bit und eine maximale Länge von 2048 Bit haben (Bild 2), in Schritten von 128 Bit ist jede Breite denkbar (also 256, 384, 512,..., 2048 Bit). Für Low-Power-MPUs können Arms Lizenznehmer Designs mit 128-Bit-Vektorlänge designen, für Super-Computer könnten Designs mit 2048 Bit breiten Vektoren erstellt werden.

Und selbst bei „nur“ 128-Bit-Implementierungen gibt es bei SVE gegenüber Neon einen großen Vorteil: Man erhält kleineren C- und damit auch Assembler-Code, später wird hierzu ein konkretes Beispiel folgen. Das bedeutet weniger Anweisungen, die in den Cache passen müssen, und weniger Anweisungen, die im Frontend der CPU dekodiert werden müssen: Am Ende des Tages spart man auf diese Weise Transistoren und Energie.

Um flexible Operationen auf ausgewählten Elementen zu ermöglichen, führen SVE und SVE2 16 sogenannte „Prädikat-Register“ P0-P15 ein, um die gültige Operation auf aktiven Lanes der Vektoren anzuzeigen. In diesem Beispiel

ADD Z0.D, P0/M, Z1.D, Z2.D

werden die aktiven Elemente Z1 und Z2 addiert und das Ergebnis in Z0 abgelegt. P0 gibt an, welche Elemente der Operanden aktiv und inaktiv sind. 'M' nach P0 zeigt an, dass das inaktive Element zusammengeführt wird, d. h. das inaktive Element Z0 behält seinen ursprünglichen Wert vor der ADD-Operation. Ein 'Z' nach P0 würde bedeuten, dass das inaktive Element im Zielvektorregister auf Null gesetzt wird.

Die Prädikat-Register weisen nur 1/8 der Größe der SVE-Register auf, da die kleinste Einheit eines Vektor-Elementes ja 8 Bits bzw. 1 Byte beträgt und somit für die Anzeige einer aktiven/inaktiven Lane 1 Bit pro Byte ausreicht (Bild 3).

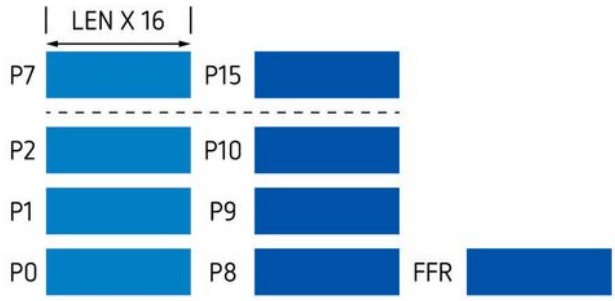

Bild 3: Die 16 Prädikat-Register definieren, welche Operanden in einem Vektorregister bei einer Rechenoperation aktiv bzw. inaktiv sind.
(Bild: SEGGER)

Bild 4 zeigt ein Beispiel, wie sich die Maskierung mittels Prädikat-Register auswirkt. In den (128 Bit breiten) Vektorregistern Z0 und Z1 befinden sich jeweils vier 32-Bit-Operanden, die addiert werden sollen. Mit P0(/M) = [1, 0, 1, 0] würden in dem Beispiel nur zwei 32-Bit-Werte für eine Addition herangezogen werden, weil die anderen beiden durch die "0" ausmaskiert wurden.

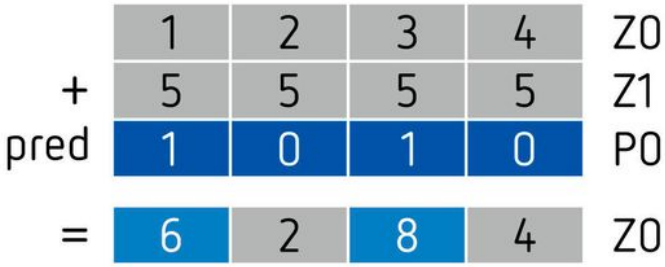

Bild 4: Vektoraddition mit Hilfe der Prädikat-Register. In diesem Beispiel werden zwei von vier Elementen in dem Vektorregister ausgeblendet.
(Bild: SEGGER)

Die prädiktionsgesteuerte Schleifensteuerung ist eine effiziente Schleifensteuerungsfunktion. Diese ermöglicht es, den durch die Verarbeitung von Teilvektoren verursachten Overheads von Schleifenköpfen und -schwänzen zu entfernen, indem der Index der aktiven und inaktiven Elemente in den Prädikat-Registern registriert wird. Dies bedeutet, dass in der nächsten Schleife nur die aktiven Elemente die erwarteten Operationen ausführen.

Dies hilft, den Code drastisch zu vereinfachen und zu vermeiden, dass man die genaue Länge des Vektors kennen muss. Angenommen, man hätte sechs 32-Bit-Werte zu verarbeiten. Mit Neon würde man Anweisungen wie diese haben:

ADD v2.4S, v0.4S, v1.4S

Man kann das ein Mal machen, aber dann hat man noch zwei Elemente übrig. Wenn man es zwei Mal macht, addiert man acht Elemente, was zu viel ist. Daher muss normaler Vektor-Code so viel wie möglich machen. Und doch hat man am Ende noch etwas, was man „Drain Loop" nennt. Das ist die Stelle, an der man einfache skalare Operationen verwenden muss, um den Rest zu berechnen.

SVE vereinfacht diese Aufgabe drastisch. In dem Beispiel

WHILELO P0.S, X8, X9

wird in P0 ein Prädikat erzeugt, das ab dem niedrigsten nummerierten Element wahr ist, solange der inkrementierende Wert des ersten, vorzeichenlosen skalaren Operanden X8 kleiner als der zweite skalare Operand X9 ist, und danach falsch bis zum höchstnummerierten Element.

Im Listing ist ein Codebeispiel in C ersichtlich, welches eine skalare Schleife einmal in Neon und einmal in SVE implementiert. Mit SVE entfällt die „Drain Loop".

Das Prädikat-Register wird z. B. auch beim Laden und Speichern von Daten verwendet. Dazu gibt es zwei Funktionalitäten, die sich Gather-Load und Scatter-Store nennen. Dieser flexible Adressmodus in SVE erlaubt eine Vektor-Basisadresse oder einen Vektor-Offset. Das ermöglicht das Laden in ein einzelnes Vektorregister von nicht zusammenhängenden Speicheradressen. Zum Beispiel werden mit

LD1SB Z0.S, P0/Z, [Z1.S, #4]

vorzeichenbehafteten Bytes in die aktiven 32-Bit-Elemente (durch P0 festgelegt) von Z0 Speicheradressen, die durch die 32-Bit-Vektorbasis Z1 plus den Index #4 erzeugt werden, geladen.

Die Anweisung

LD1SB Z0.D, P0/Z, [X0, Z1.D]

lädt vorzeichenbehaftete Bytes in aktive Elemente von Z0 von Speicheradressen, die durch eine 64-Bit-Skalar-Basis X0 plus einen Vektorindex in Z1.D erzeugt werden.

Die Auswirkungen von SVE auf zukünftige Anwendungen

Im Gegensatz zur x86-Architektur muss Arm nicht alle paar Jahre zahlreiche neue SIMD-Anweisungen zum Befehlssatz hinzufügen. SVE/SVE2 bietet eine Grundlage mit viel Stabilität und Wachstumsspielraum. Für Entwickler wird das Schreiben und Optimieren von Arm-Code einfacher, insbesondere auch für maschinelles Lernen,

Spracherkennung u.s.w. Für die Industrie bedeutet es, dass es eine viel größere Vielfalt an Geräten geben wird, die maschinelles Lernen nutzen. Das Endgerät kann dann vieles von dem übernehmen, was früher die Cloud erledigen musste („KI at the Edge“).

Ein typischer Anwendungsfall und eine der häufigsten verwendeten Operationen bei maschinellem Lernen ist die Matrix-Multiplikation. Es handelt sich um eine binäre Operation, die als Eingabe zwei Matrizen (A der Dimensionen (M,K) und B der Dimensionen (K,N)) erhält und als Ausgabe eine Matrix C der Dimensionen (M,N) zurückgibt, in der jedes Element Cij das Skalarprodukt der Zeile i der Matrix A mit der Zeile j der Matrix B ist (0 <= i < M, 0 <= j < N).

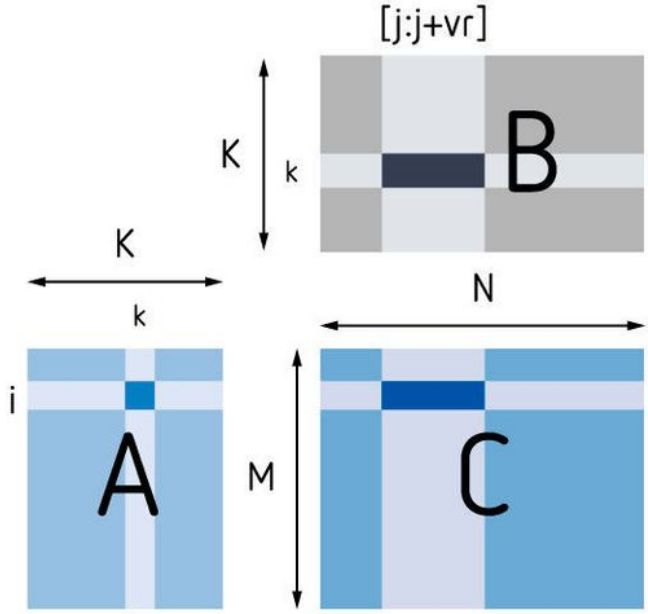

Bild 5: Matrix-Multiplikation unter Nutzung der SVE-Vektorinstruktionen. Eine Zeile von A wird spaltenweise mit jeweils mehreren Spalten in der hellgrauen Schablone von B multipliziert und ergibt den dunkelblauen Vektor in C.
(Bild: SEGGER)

Die Matrix-Multiplikation hat ein großes Potenzial für die Parallelisierung aufgrund der sich wiederholenden Operationen und der Anordnung der Daten. Zum Beispiel ist jedes Element der Zeile i der resultierenden Matrix C das Skalarprodukt der Zeile i der Matrix A mit der entsprechenden Spalte der Matrix B. Wenn vr die Anzahl der Elemente eines Vektorregisters darstellt, kann man vr Elemente der Zeile i der Matrix C auf einmal berechnen, indem man die Skalarprodukte der Zeile i der Matrix A mit vr Spalten der Matrix B berechnet (Bild 5):

$$C_{i,[j:j+\mathrm{vr}]} = \sum_{k=1}^{K} A_{i,k} * B_{k,[j:j+\mathrm{vr}]}$$

(Bild: Segger)

In einem Whitepaper [1] hat Arm mehrere Codebeispiele für Implementierungen von Matrizenmultiplikationen, u. a. auch die soeben geschilderte, zusammengefasst. Sie nutzen allesamt die Arm C-Spracherweiterungen (ACLE) [2] sowohl für SVE als auch SVE2.

Ein Beispiel hierfür bietet SEGGER's Entwicklungsumgebung Embedded Studio, die mit zwei Toolchains (GCC und LLVM) ausgeliefert wird [3], welche beide ACLE unterstützen (GCC ab Version 10, LLVM ab Version 11) und qualitativ hochwertigen Code produzieren. Welche von beiden den kleinsten oder schnellsten Code erzeugt, hängt von der Zielanwendung ab. Deshalb enthält Embedded Studio beide Toolchains, um Entwicklern die besten Tools für ihr jeweiliges Projekt zur Verfügung zu stellen. Im Vergleich zu Wettbewerbsprodukten erzielt Embedded Studio z.B. bei dem anerkannten Open-Source-Benchmark emBench-IoT (https://github.com/embench/embench-iot) über 10 % kompakteren Code.

SVE2 vs. SVE

SVE2 ermöglicht die Vektorisierung eines breiteren Anwendungsspektrums als SVE. Beispiele sind Anwendungsfälle in Edge- und Client-Server-Computing, Codecs/Filter, Computer Vision, AR/VR, Networking, Baseband sowie Kryptografie. Konkret gibt es SVE2-Befehle für Festkomma-DSP-Arithmetik (z. B. Arithmetik komplexer Zahlen für LTE), erweiterte Permutations-Instruktionen (z. B. CV, FIR, FFT, LTE, ML, Genomik, Kryptoanalyse), Beschleunigung für String-Verarbeitung (Parser) und Kryptografie (AES, SM4, SHA-Standards).

Implementierung von SIMD auf Mikrocontrollern

Mit der Armv8.1.-M-Architektur wurden die „M-Profile Vector Extensions" (MVE) für die Arm Cortex-M-Prozessorfamilie vorgestellt. Der Cortex-M55 ist der erste Arm-Prozessor, der diese Technologie unterstützt.

Helium – so analog zu Neon die Bezeichnung – ist ein von Grund auf neu entwickeltes Design, das effiziente Leistung in kleinen Prozessoren ermöglicht. Es bietet viele neue architektonische Eigenschaften, die in Neon nicht verfügbar sind, wie die hier zuvor im SVE-Kontext beschriebenen Funktionen Schleifenprädikation, Lane-Prädikation und Scatter-Gather-Speicherzugriffe. Allerdings ist die Vektorlänge wie bei Neon auf 128 Bit festgelegt.

Mit emVDSP bietet SEGGER eine Signalverarbeitungs- und Vektor-Bibliothek an, die auf mehrere Architekturen incl. Armv8.1.-M ausgerichtet ist. emVDSP bietet eine reguläre API über alle Datentypen für alle Targets und unterstützt sowohl „normale" als auch Sättigungs-Festkomma-Arithmetik. Wo Algorithmen beschleunigt werden können, nutzt emVDSP die zugrundeliegenden Hardware-Features und natürlich Vektorbefehle,

um mehrere Operationen parallel laufen zu lassen. Wie ein Performance-Vergleich zeigt [4], schneidet emVDSP in der Standard-Distribution ohne Tuning durchweg besser ab als CMSIS-DSP, wobei in emVDSP auch noch jede Funktion einzeln getunt werden kann.

Fazit

Die nunmehr in der Armv9-Architektur verankerten skalierbaren Vektorerweiterungen können wesentlich dazu beitragen, Anwendungen wie maschinelles Lernen auf kleinen und ressourcenschwachen Endgeräten zu ermöglichen – ohne Einsatz der Cloud und die damit verbundenen Ressourcen und Latenzzeiten. Sie reduzieren die Codegröße, machen Programme schneller und sparen Energie.

Um ressourcenschonende Anwendungen zu entwickeln, benötigt es jedoch noch mehr: eine Entwicklungsumgebung, die kompakten und performanten Code generiert und Bibliotheken z. B. für die Erstellung von GUIs, Kryptografie, Security und Konnektivität, die einen maximal effizienten Code beinhalten, um Speicherbedarf und Energieverbrauch der Endanwendung zu minimieren.

SEGGER bietet handcodierte und über viele Jahre optimierte Bibliotheken für all diese Einsatzzwecke [5] und mit Embedded Studio eine Entwicklungsumgebung an, die speziell für minimalistischen und schnellen Code in Embedded Devices entwickelt wurde. Sie sind eine optimale Grundlage für Anwendungen wie "KI at the Edge".

Literatur

[1] Whitepaper mit Implementierungen von Matrizen-Multiplikationen unter Nutzung von SVE

[2] Dokumentation der Arm-C-Spracherweiterungen für SVE/SVE2

[3] Entwicklungsumgebung Embedded Studio von SEGGER

[4] Performance-Vergleich von SEGGER's Bibliothek emVDSP mit CMSIS-DSP

[5] SEGGER's optimierte Embedded Software

* Frank Riemenschneider ist Senior-Marketing- und PR-Manager bei der SEGGER Microcontroller GmbH in Monheim am Rhein und Arm Accredited Engineer.

(ID:47458891)