# AMD's Unified CPU & GPU Processor Concept

Sven Nobis

*Institute of Computer Engineering*
*University of Heidelberg*
*Mannheim, Germany*
nobis@stud.uni-heidelberg.de

***Zusammenfassung*—Massiv parallelisierbare Berechnungen können von Grafikprozessoren (GPUs) sehr effizient durchgeführt werden. Die Entwicklung von solchen heterogenen Programmen weißt derzeit zwei Probleme auf: Zum einen sind die Kommunikationskosten zwischen CPU und GPU hoch. Zum anderen ist die Barriere hoch, da Entwickler neue, gegenüber der normalen CPU-Programmierung ungewohnte, Konzepte lernen müssen.**

**Die Lösung dieser beiden Probleme geht AMD mit seinem Unified CPU & GPU Processor Concept an. In diesem Paper werden die wesentlichen Konzepte dieser *Heterogeneous System Architecture* kurz vorgestellt und es wird auf aktuelle Entwicklung eingegangen werden.**

**So lässt sich im Groben sagen, dass das Ziel auf der Hardwareseite mit einem vereinheitlichen Speicherzugriff für beide Prozessoren umgesetzt wird. Im Gesamten soll ein System aus Compilation-, Runtime, System-(Kernel)-Komponenten und vereinheitlichten Speicher- und Programmiermodell die Ziele umsetzen. Bereits vorgestellte Softwarebibliotheken geben schon einen Eindruck, wie Entwicklung paralleler Anwendungen in Zukunft aussehen können.**

## I. Einführung

Das auch über die Informatik hinaus bekannte Mooresche Gesetz spricht von einer Verdopplung der Anzahl von Transistoren auf einem Chip alle 18 Monate. 1975 erweiterte David House, ein Kollege von Moore, die Ausgabe auf die Verdopplung der Leistungsfähigkeit von Computern. Diese lässt sich tatsächlich alle 20 Monate beobachten [1].

Die Leistungsfähigkeit eines Prozessors kann beschreiben werden, als die Zeit die es benötigt, eine gegebene Aufgabe zu lösen. Diese wird von maßgeblich zwei Faktoren beeinflusst: Die Frequenz und die Anzahl der Instruktionen die pro Taktzyklus ausgeführt werden (*Instructions executed Per Clock (IPC)*):

$$\text{Performance} = \text{IPC} \cdot \text{Frequency}$$

Bis vor ungefähr 10 Jahren konnte die Leistung maßgeblich durch die Erhöhung der Frequenz gesteigert werden. Mit der damals aktuellen NetBurst-Mikroarchitektur erreichte Intel erstmals ein Ende dieser Entwicklung. So wurde mit 3,8 GHz die maximale Frequenz auf dieser Architektur erreicht. Darüber hinaus stellte der Wärmeverlust ein zu großes Problem dar. [2]

Wenn die Frequenz nicht mehr erhöht werden kann, so muss man sich auf die Instruktionen pro Taktzyklus konzentrieren: Die Entwicklung ging von Single-Core-Prozessoren hin zu Multi-Core-Prozessoren.

Die effiziente Ausnutzung der Leistungsfähigkeit von Multi-Core-Prozessoren stellt sich aber auf der Softwareseite als ein Problem dar: Komplexer werdende Programme fordern bessere Entwicklungswerkzeuge. So werden höhere Programmiersprachen, wie C/C++ und Java, genutzt anstatt Assembler Code zu schreiben. Diese Entwicklung fand in der „Single-Core Era" statt. Mit den Beginn „Multi-Core Era" werden Werkzeuge benötigt, um die Programme effizient auf mehreren Kernen zu verteilen. Die Entwicklung führte von POSIX Threads zu OpenMP und Threading Building Blocks (TBB).

Von der „Multi-Core Era" startet derzeit ein Übergang in die „Heterogeneous Systems Era" ([3, Folie 5]). Dabei wird neben der CPU, auch der Grafikprozessor (GPU) für Berechnungen genutzt. Im Gegensatz zu CPUs sind GPUs auf ein sehr hohes Maß an Parallelisierung ausgelegt. So können sie sehr effizient auf eine große Menge an Daten die gleiche Rechenoperation ausführen. Dies ist nicht nur bei 3D-Berechnungen der Fall, sondern auch bei vielen anderen Berechnungen, z.B. im wissenschaftlichen Bereich. Bekannte Plattformen zur GPU-Programmierung sind CUDA und OpenCL.

Die Entwicklung dieser heterogenen Programme weißt derzeit zwei Probleme auf:

- Die Barriere bei der Programmierbarkeit
  - Ungewohnt gegenüber CPU-Programmierung
  - Entwickler muss mit der GPU-Programmierung vertraut sein
- Hohe Kommunikationskosten zwischen CPU und GPU

AMD versucht mit seinem Unified CPU & GPU Processor Concept diese beiden Probleme zu lösen.

## II. Hintergrund

Im Folgenden sollen die Unterschiede zwischen CPUs und GPUs erläutert werden und es wird auf die beiden bestehenden Plattformen CUDA und OpenCL eingegangen werden.

### A. CPUs vs. GPUs

Im Allgemeinen ist die GPU auf das Rendern von Grafiken spezialisiert, die CPU hingegen als Hauptprozessor auf keine spezielle Aufgabe.

Betrachtet man die Prozessoren abseits des Grafikrenderns, so lässt sich die GPU optimal für Programmabläufe mit hoher Datenparallelität nutzen. Mit GPUs können große Datenmengen parallel abgearbeitet werden. GPUs besitzen dazu sehr viele Streamprozessoren (SP, auch Kerne genannt). Bei

dem aktuellen GK110-Chip von NVIDIA sind das 2880. Im Unterschied zur CPU, die nur einen bis wenige Prozessorkerne besitzt, arbeiten die Kerne der GPU nach dem Single instruction, multiple data (SIMD) Prinzip. So werden mehrere Kerne zu einer Einheit zusammengefasst. Diese führt die gleiche Operation auf allen Kernen aus. Der Unterschied besteht darin, dass jeder Kern die Operation auf anderen Daten ausführt. Da viele Daten auf einmal geladen werden, besitzen die GPUs auch eine hohe Bandbreite zum Speicher (siehe Abbildung 1). Im Gegenzug ist die Latenz hoch [4].

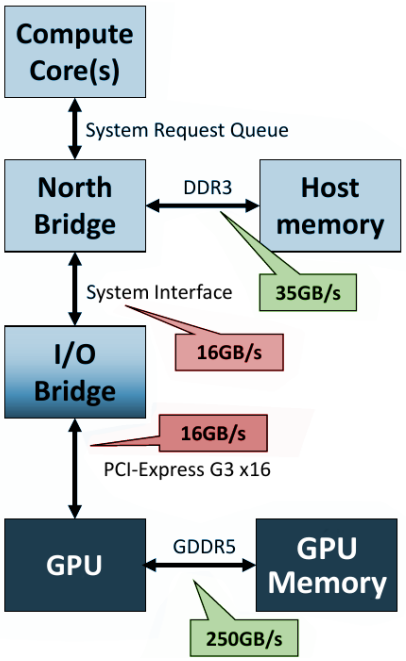



Abbildung 1. Speicheranbindung bei einem klassischen System [4, P. 4]

### B. OpenCL und CUDA

Die Open Computing Language (OpenCL) und Compute Unified Device Architecture (CUDA) sind beides etablierte Plattformen, die zur Programmierung von GPUs genutzt werden. Während CUDA eine proprietäre Entwicklung von NVIDIA ist und sich nur auf dessen Grafikkarten nutzen lässt, ist OpenCL ein offener Standard. Er ist auch nicht auf GPUs beschränkt. Ein OpenCL-System besteht stattdessen aus einem Host und Compute Devices. Auch die Kerne einer CPU können ein Computing Device sein.

Ein Compute Device besteht aus mehreren Compute Units (vgl. Abbildung 2). Die Compute Units beinhalten wiederum eine Anzahl von Processing Elements. Diese entsprechen den oben genannten Kernen.

Die Abbildung 3 zeigt das Execution Model von OpenCL. Eine Aufgabe umfasst einen Kernel, der eine bestimmte Funktion über einen bestimmten Bereich von Daten ausführt. Die kleinste Einheit sind dabei Work-Items. Diese werden zu einer ein- bis drei-dimensional Work-Group zusammengefasst, in der jedes Work-Item einen lokale Adresse hat. Eine Work-Group teilt sich gemeinsamen Speicher und ist Teil einer n-

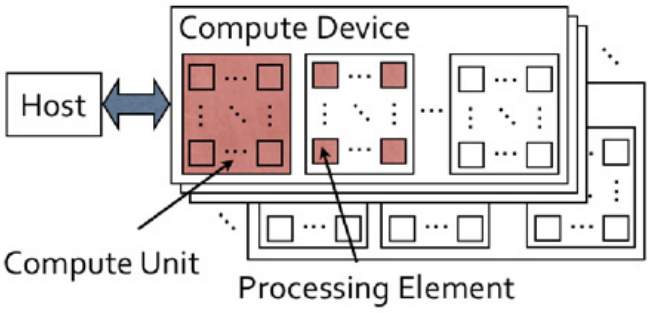



Abbildung 2. OpenCL Platform Model [5]

dimensionalen Range (NDRange). In dieser hat jede Work-Group einen globale Adresse. Eine Kommunikation unter den Work-Items kann über den Speicher erfolgen.

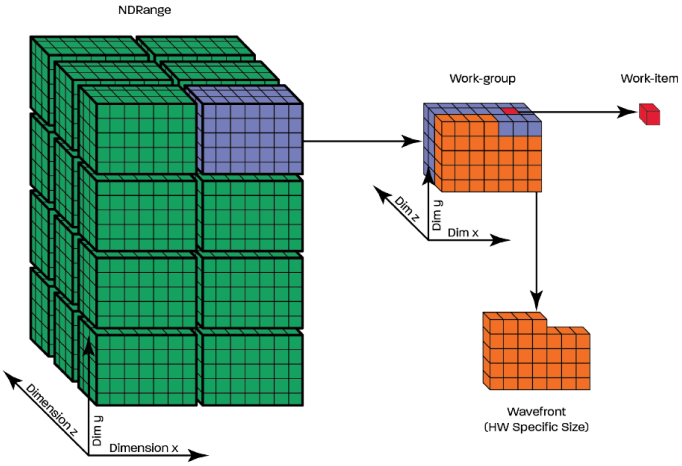



Abbildung 3. OpenCL Execution Model [6, P. 11]

## III. Abgrenzung zu verwandten Arbeiten

Auch in CUDA wurden Konzepte umgesetzt, die der HSA ähneln [7]. So führte NVIDIA mit CUDA 4 *Unified Virtual Addressing (UVA)* ein. Es bietet einen einheitlichen virtuellen Speicheradressraum. Dabei kann die GPU auf Speicher des Host Systems zugreifen, ohne diesen vorher kopieren zu müssen. Jedoch bleibt hier PCI-Express-Schnittstelle der Flaschenhals (vgl. Abbildung 1). OpenCL hat eine ähnliche Unterstützung mit Shared Virtual Memory eingeführt.

Mit CUDA 6 führt Unified Memory ein. Das Konzept beschränkt sich hier jedoch auf die Entwicklersicht. Es muss kein explizites Kopieren der Daten in den Speicher der GPU zurück erfolgen. So existieren weiterhin zwei physikalische Speicher, den der GPU und den der CPU.

## IV. Der Weg zur Heterogeneous System Architecture

Mit dem Konzept der Heterogeneous System Architecture will AMD die CPU und GPU zu einem System vereinen. Durch die nahtlose Integration beider Architekturen soll

für den Entwickler die Barriere zur GPU-Programmierung verschwinden. Zusätzlich sollen Kommunikationskosten zwischen den beiden Architekturen reduziert werden.

Die Umsetzung bedarf neben eines neuen Systems auf Softwareseite, zuerst einmal zwei grundlegende Änderungen auf der Hardwareseite: Im ersten Schritt müssen die CPU und die GPU auf einem Die vereint werden. Dieser Typ von Mikroprozessoren wird von AMD Accelerated Processing Unit genannt.

Den Anfang bei den „Mainstream“-Prozessoren macht dabei die Llano-Serie. Diese wurden im Juni 2011 veröffentlicht und vereint erstmals die beiden Processing Units auf einem Chip. Die zweite Generation der APUs basiert auf der Piledriver-Architektur und wurde mit der Trinity-Serie im Oktober 2012 veröffentlicht. Anfang 2013 folgte die Richland-Serie.

Die GPU und die CPU haben noch getrennten physikalischen Speicher, einen sogenannten Non-uniform memory access (NUMA). Die Abbildung 5 soll dies verdeutlichen. So binden bei einer klassischen CPU alle Kerne einen gemeinsamen Speicher an (erste Ziele). Bei einer APU wird der Speicher partitioniert: In einen Teil für die CPU und einen für die GPU (zweite Zeile). Beide haben einen separaten virtuellen Adressraum (Abbildung 6(a)).

Um seinen eigenen Speicher anzusprechen, hat die GPU eine inkohärente Speicherschnittstelle (Garlic, siehe Abbildung 4). Darüber können große Datenmengen (z.B. Texturen) auf einmal geladen werden. Des Weiteren gibt es eine langsamere kohärente Schnittstelle (Onion), um mit der CPU einfach Daten austauschen zu können. Dabei wird vom Betriebssystem vorgegeben, welche Speicherbereiche gemeinsam genutzt werden. Diese werden Pinned Memory genannt.

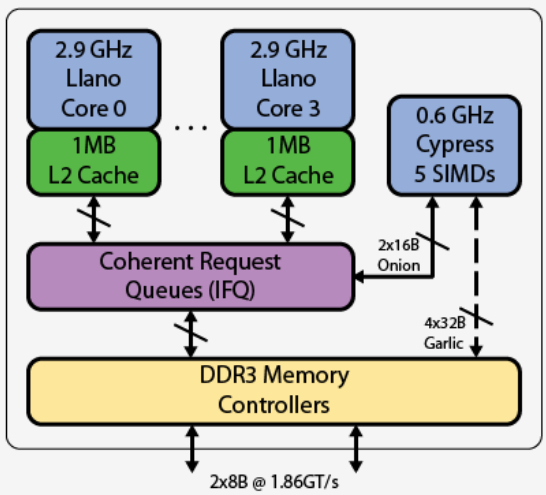



Abbildung 4. Physikalische Speicheranbindung der Llano-Serie [8]

### A. *Heterogeneous Unified Memory Access (hUMA)*

Mit der Steamroller-Architektur (Kaveri) soll die sogenannte HSA-specific Memory Managment Unit (HMMU) eingeführt werden. Diese hebt die Partitionierung des physikalischen Adressraums auf (dritte Zeile in Abbildung 5). So

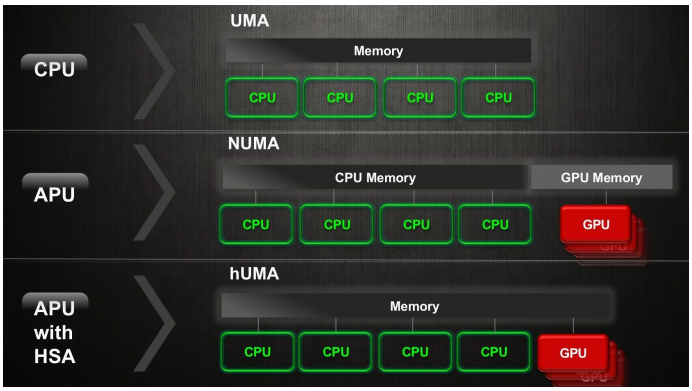
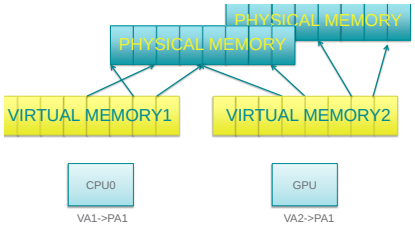
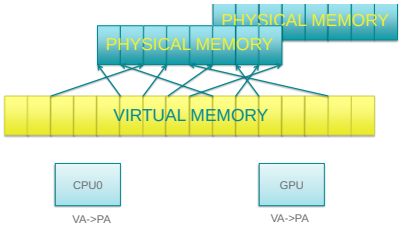



Abbildung 5. Vergleich der physikalischen Speicherbereiche [9]




(a) NUMA




(b) hUMA

Abbildung 6. Virtueller Adressraum [10, P. 7-8]

haben CPU und GPU nur noch einen gemeinsamen Speicher. Auch der virtuelle Adressraum wird vereint. So hat eine Speicherzelle in beiden Umgebungen die gleiche Adresse (siehe Abbildung 6(b)).

Dies bietet einen großen Vorteil gegenüber einer klassischen GPU: Ein der Daten entfällt. Wie erwähnt ist die PCI-Express Schnittstelle der Flaschenhals des Systems (Abbildung 1 auf der vorherigen Seite). Bindet die GPU ihren Speicher mit bis zu 250 GB/s an, so können die Daten jedoch nur mit 16 GB/s über die PCI-Express zur GPU hin übertragen werden. Mit hUMA entfällt dieser Schritt.

Ein notwendiges Feature dafür ist die Unterstützung für das Page Faulting. Mit Page Faulting können Speicherbereiche von Arbeitsspeicher auf die Festplatte ausgelagert werden. Bei den bisherigen APUs unterstützt die GPU kein Page Faulting. Ein Verschieben von Pinned Memory vom RAM auf die Festplatte (Page out) wird durch einen Betriebssystemaufruf verhindert. Dieser wird durch den Memory Controller ausgelöst und kann den Page out abbrechen. Mit hUMA werden nun auch Page Faults der GPU genauso gehandhabt wie bei der CPU.

Die Sicherstellung der Cache-Kohärenz ist bei heute übli-

chen Mehrprozessorsystemen Standard. hUMA schließt nun zusätzlich zu den CPUs auch die GPUs mit ein, da diese nun ebenfalls den gleichen Speicher nutzen.

## V. Heterogeneous System Architecture

Im Folgenden soll zuerst auf die Konzepte der Heterogeneous System Architecture eingegangen werden. Auf die Umsetzung von der Hardwareseite wurde schon im vorherigen Abschnitt eingegangen und wird mit der Erläuterung der Komponenten des Systems fortgeführt. Zum Schluss wird, mit dem Abschnitt Softwareentwicklung, HSA aus der Sicht des Entwicklers gezeigt.

### A. Konzepte

In diesem Abschnitt wird auf die grundlegenden Konzepte der Heterogeneous System Architecture eingegangen.

*Zur Begrifflichkeit:* CPU und GPU werden bei HSA generalisiert: So entspricht die CPU einer latency compute unit (LCU). Die GPU entspricht einer throughput compute unit (TCU), d.h. einer Einheit, die sehr effizient in paralleler Ausführung ist.

*1) Unified Adress Space:* Auf die Vereinheitlichung des Adressraums wurde bereits in Abschnitt IV eingegangen. So zeigt ein Pointer (virtuelle Adresse) sowohl auf der LCU als auch auf der TCU auf den gleichen Speicher. Das ermöglicht auch die gemeinsame Nutzung Pointer-basierter Datenstrukturen, wie verkettete Listen. Durch die virtuelle Speicherverwaltung und des Paging kann außerdem ein viel größer Speicherraum angesprochen werden.

*2) Unified Programming Model:* Alle bisherigen GPU-Programmiermodelle, wie CUDA oder OpenCL, haben gemeinsam, dass die Daten explizit zwischen den Adressräumen kopiert werden müssen und die GPU als entfernter Prozessor gesehen wird. Für Multi-Core-CPUs sind in den letzten Jahren Aufgaben-basierte APIs (Task parallelism; z.B. Thread Building Blocks) in den Vordergrund gerückt. Anstatt das sich der Entwickler manuell um die Erstellung, Synchronisierung und Terminierung von Threads kümmert, wird hier der API eine Aufgabe (z.B. Sortieren) zur Abarbeitung übergeben. Die API kümmert sich dann um die dynamische Verteilung der Aufgabe auf die Cores. HSA erweitert dies: So soll der Entwickler in seiner gewohnten Programmierumgebung bleiben und das Programm trotzdem sowohl TCU als auch LCU nutzen. Dabei sollen existierende Sprachen und APIs zur parallelen Datenverarbeitung (Data parallelism) um die Fähigkeit erweitert werden, HSA zu nutzen.

*3) Queuing:* Die Kommunikation zwischen den Compute Units erfolgt über Warteschlangen. Dies ist in Abbildung 7 dargestellt. Wie bei OpenCL kann die LCU der TCU so Aufgaben übergeben. Jedoch kann bei HSA auch die TCU sich selber neue Aufgaben zuweisen, ohne einen Roundtrip über die CPU machen zu müssen. Um System Operationen, z.B. zum Allokieren von Speicher, zu ermöglichen, kann die TCU auch der LCU Aufgaben übergeben.

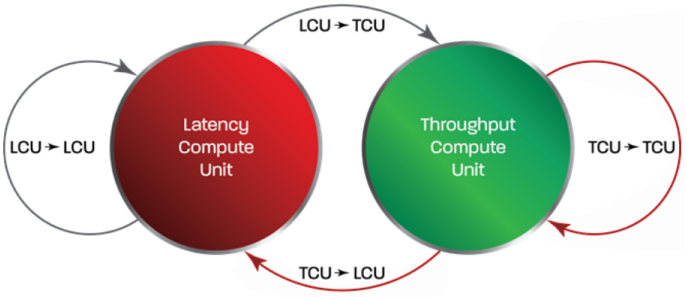



Abbildung 7. Queuing [6, P. 9]

*4) Premption & Context Switching:* Aktuelle GPUs unterstützen kein präemptives Multitasking. Das hat zur Folge, das ein Prozess der die TCU für einen längeren Zeitraum in Beschlag nimmt, andere Prozesse beim Voranschreiten hindert. Ein besonderes Problem gibt es bei Prozessen, die in einen Fehlerzustand laufen. So ist die TCU belegt, bis der Zustand beseitigt ist und der Prozess abschlossen. HSA soll durch Fehlerbehandlungsmechanismen, flexible Jobsteuerung und die Möglichkeit Jobs zu stoppen und später fortzufahren (job preemption) dieses Problem lösen.

*5) HSA Intermediate Language:* Quellcode der auf der TCU ausgeführt werden soll, wird in einen Zwischencode kompiliert, der HSA Intermediate Language (HSAIL). HSAIL wird jedoch auch von LCU unterstützt, sodass das Programm auch auf nicht HSA-kompatibler Hardware ausgeführt werden kann. Dieser Zwischencode wird erst zur Laufzeit in das Hardware Instruction Set der jeweils gerade verbauten TCU kompiliert. Dadurch soll Kompatibilität und Portabilität sichergestellt werden. So kann der Entwickler davon ausgehen, dass sein Programm auf der kommenden Hardwaregeneration performant laufen wird.

Das Execution Model von HSAIL gleicht dem bereits beschriebenen OpenCL Execution Model.

### B. System Komponenten

Zum System gehört zum einen die Heterogeneous Hardware: eine APU. Dazu kommen drei Softwarekomponenten, die in den folgenden Abschnitten vorgestellt werden:

*1) Compilation Stack:* Der HSA Compilation Stack hat die Aufgabe eine höhere Programmiersprache in HSAIL zu kompilieren. Der Stack basiert auf der LLVM Compiler Infrastructure, einer modularen Compiler-Architektur.

Die Abbildung 8 auf der nächsten Seite zeigt den Aufbau des Stacks: Mit dem Compiler Front End der höheren Programmiersprache wird der Quellcode in die LLVM Intermediate Representation (LLVM IR), einer Zwischensprache, kompiliert. Dabei ist Compiler Front End in diesem Schritt dafür verantwortlich die parallelen Programmteile heraus extrahieren. Das sind die Teile, die auf der TCU ausgeführt werden sollen. Das OpenCL Front End als Beispiel übernimmt diese Aufgabe bereits. In der LLVM IR werden geeignete Optimierungen vorgenommen. Danach wird der parallele Teil in

HSAIL konvertiert. Daraus wird, zusammen mit dem nativen CPU-Code für die gesamte Anwendung, die ausführbare Datei generiert.

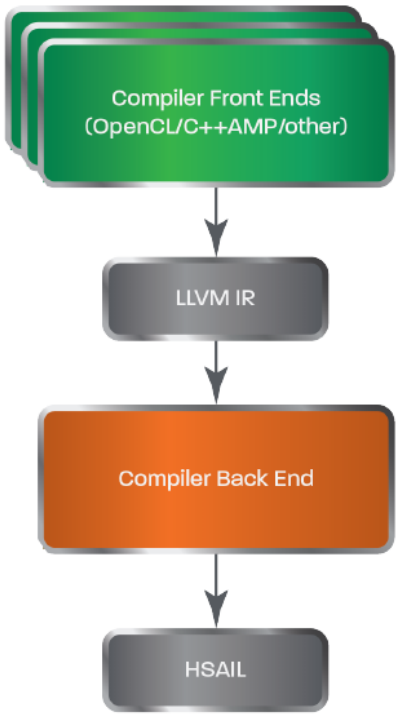



Abbildung 8. HSA Compilation Stack [6, P. 15]

*2) Runtime System:* Das Runtime System ist in eingebettet in die Umgebung in der es ausgeführt wird. Da bei der Kompilierung auch der parallele Teil noch als CPU-Code vorliegt, läuft die Anwendung auch auf Systemen die keine Unterstützung für HSA haben. Unterstützt das System jedoch HSA, so kommt der Runtime Stack zum Einsatz. Dieser in Abbildung 9 dargestellt. Dazu ist zu sagen, dass der CPU-Teil nicht notwendigerweise Vorhandensein muss. Die HSA Runtime wird in der Regel von der Runtime der eingesetzten Sprache getriggert (z.B. OpenCL). Die HSA Runtime übergibt die Arbeit dann an das HSA GPU Device. Dort wandelt der Finalizer den HSAIL-Code in das Hardware Instruction Set des Device. Der Kernel Treiber kümmert sich danach um die Allokierung der Ressourcen und die Ausführung auf der GPU.

*3) Kernel-Space System Komponenten:* Das Kernel-Space System besteht wiederum aus vier Komponenten:

- dem Gerätetreiber, der die HSA Hardware Ressourcen verwaltet
- dem hMMU-Gerätetreiber
- einem Scheduler, der die Ausführung der Jobs verwaltet
- einem angepasster OS Memory Manager, mit HSA-Support, um die Regionen zu Verwalten, die von der TCU genutzt werden

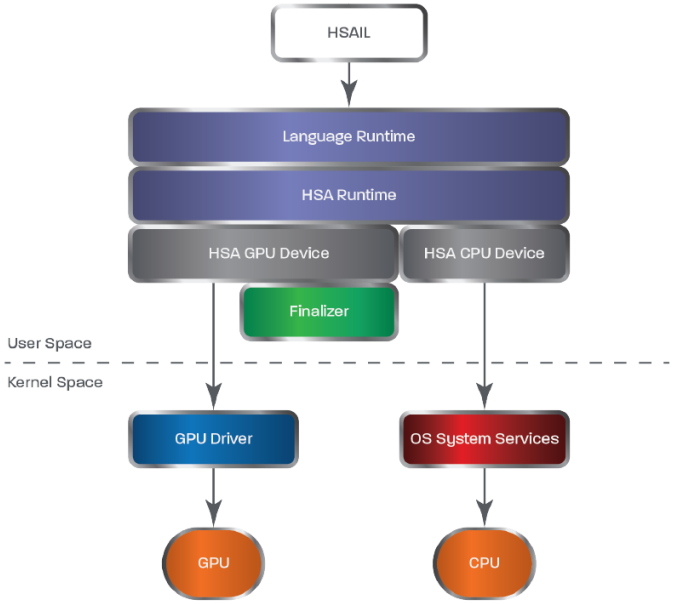



Abbildung 9. HSA Runtime Stack [6, P. 16]

### C. Softwareentwicklung mit HSA

HSA hat sich nicht zum Ziel gesetzt, eine neue Sprache zur parallelen Entwicklung zu etablieren. Stattdessen soll es sich in bestehende Programmiersprachen und -modelle einpflegen. So kann der Entwickler in seiner vertrauten Umgebung bleiben und soll schon durch geringe Änderungen seines Quellcodes von den Vorteilen der HSA profitieren. Einige bereits bestehende Ansätze werden im folgenden kurz vorgestellt:

*1) OpenCL:* HSA soll keine alternative zu OpenCL darstellen, sondern durch die vorgestellten Konzepte optimiert werden. Durch diese sollen OpenCL-Programme mit geringer Latenz auf der GPU ausgeführt werden können [3].

*2) C++ Accelerated Massive Parallelism:* C++ AMP ist eine Bibliothek und offene Spezifikation, die die Implementierung von Data Parallelism direkt in C++ erlaubt. Ursprünglich wurde es von Microsoft auf DirectX 11 entwickelt. Durch eine Portierung auf LLVM und OpenCL soll in Zukunft HSAIL Code generiert werden [11]. Die Ausführung auf OpenCL funktioniert bereits.

*3) BOLT Library:* Diese C++ Template Library bietet fertige Routinen für bekannte Probleme, wie `scan`, `sort`, `reduce` und `transform`. Die Routinen sind dabei auf Heterogeneous Computing optimiert und werden auf der GPU ausgeführt. Die API lehnt sich an die C++ Standard Template Library an. Der Entwickler kann die Routinen einfach Einsetzen und braucht sich nicht groß mit den Besonderheiten der GPU-Programmierung auseinandersetzen.

*4) Aparapi:* Diese API ermöglicht es Java Entwicklern die Rechenleistung der GPU zu nutzen. Dabei wird der Java Bytecode, der parallel ausgeführt werden soll, nach OpenCL kompiliert und auf der GPU ausgeführt. Mit der Fokussierung auf HSA, soll in Zukunft soll HSAIL generiert werden.

## VI. Fazit und Ausblick

HSA ist ein interessanter Lösungsansatz, die heutigen Probleme im Bereich des GPU-Computing zu lösen. Dass es sich hierbei um eine offene Plattform handelt, lässt auf eine Etablierung hoffen. Bis dahin ist es noch ein weiter Weg. Zuerst einmal muss die Hardware auf dem Markt sein, die HSA vollständig unterstützt: Mit der Veröffentlichung der Kaveri-APU im Januar zeigt AMD, dass sie hier größten Schritt auf dem Weg zu HSA geschafft haben. Die Kaveri-APU unterstützt sowohl hUMA, als auch das Queuing [12].

Auch auf Softwareseite sind schon interessante Ansätze zu sehen. Fertig ist jedoch noch keiner davon. So nutzen die bisher gezeigten Bibliotheken nur OpenCL. Sie bieten jedoch auch schon so Nutzungsmöglichkeiten. Es stellt sich die Frage, ob man mit Bolt schon jetzt, ohne großen Aufwand, bestehende Programme beschleunigen kann.

Ob HSA das Halten kann, was es verspricht, zeigt sich erst, wenn sowohl Hardware als auch Software HSA vollständig unterstützen. Ist das jedoch der Fall, so vermute ich, dass sich dadurch das Potenzial der GPU in einigen Anwendungen ohne hohen Aufwand nutzen lässt. Das und der offene Ansatz, der eine Implementierung auch auf ähnlichen Plattformen möglich macht, spricht für die Zukunft dieses Konzepts.

Wenn man sich mit dem Thema GPU-Computing beschäftigt, sollten man die Entwicklung von HSA im Auge behalten.

## Literatur


[1] M. Kanellos, "Moore's law to roll on for another decade," CNET News, Feb. 2003. [Online]. Available: http://news.cnet.com/2100-1001-984051.html

[2] P. Gepner and M. Kowalik, "Multi-core processors: New way to achieve high system performance," in *Parallel Computing in Electrical Engineering, 2006. PAR ELEC 2006. International Symposium on*, 2006, pp. 9–13.

[3] P. Rogers, "Heterogeneous system architecture overview," HOT CHIPS 2013, Aug. 2013. [Online]. Available: http://de.slideshare.net/hsafoundation/hsa-intro-hot-chips2013-final

[4] H. Fröning, "Lecture 03 – Basic Architecture," Lecture: GPU Computing, 2013.

[5] A. Staff, "Opencl™ and the amd app sdk v2.4," Apr. 2011. [Online]. Available: http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-and-the-amd-app-sdk-v2-4/

[6] G. Kyriazis, "A heterogeneous system architecture: Technical review," HSA Foundation, Tech. Rep., Aug. 2012.

[7] M. Harris, "Unified memory in cuda 6," Nov. 2013. [Online]. Available: http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/

[8] D. Kanter, "Amd fusion architecture and llano," real world technologies, Jun. 2012. [Online]. Available: http://www.realworldtech.com/fusion-llano

[9] M. Fischer, "AMDs Unified Memory: Mehr Performance und Energieeffizienz für Kaveri und Co," Heise Online, Apr. 2013. [Online]. Available: http://heise.de/-1850975

[10] I. Bratt, "HSA Queueing," HOT CHIPS 2013, Aug. 2013. [Online]. Available: http://www.slideshare.net/hsafoundation/hsa-queuing-hot-chips-2013

[11] G. Stoner, "Bringing c++amp beyond windows via clang and llvm," Nov. 2013. [Online]. Available: http://hsafoundation.com/bringing-camp-beyond-windows-via-clang-llvm/

[12] B. Benz, "AMD fordert mit Kaveri Intels Core i5 heraus," Heise Online, Jan. 2014. [Online]. Available: http://heise.de/-2085447