quadric

What’s the Difference Between an NPU and a GPNPU?

To understand the difference between an NPU (neural processing unit) and a GPNPU (general-purpose neural processing unit) let’s start with the NPU, a processing engine that accelerates machine learning (ML) workloads in System on Chip (SoC) designs.

An NPU works alongside a CPU and a GPU and/or a DSP, on a chip (see Figure 1). The other more generalized processors call on the NPU to do specific compute-intensive, matrix-math ML workloads, such as image classification acceleration or object detection. The NPU – sometimes called a Deep Learning Accelerator or Machine Learning Offload accelerator - is not a general-purpose core but instead it is narrowly optimized for the specific ML tasks, and therefore is much more efficient in both time and power consumption at those specific tasks.

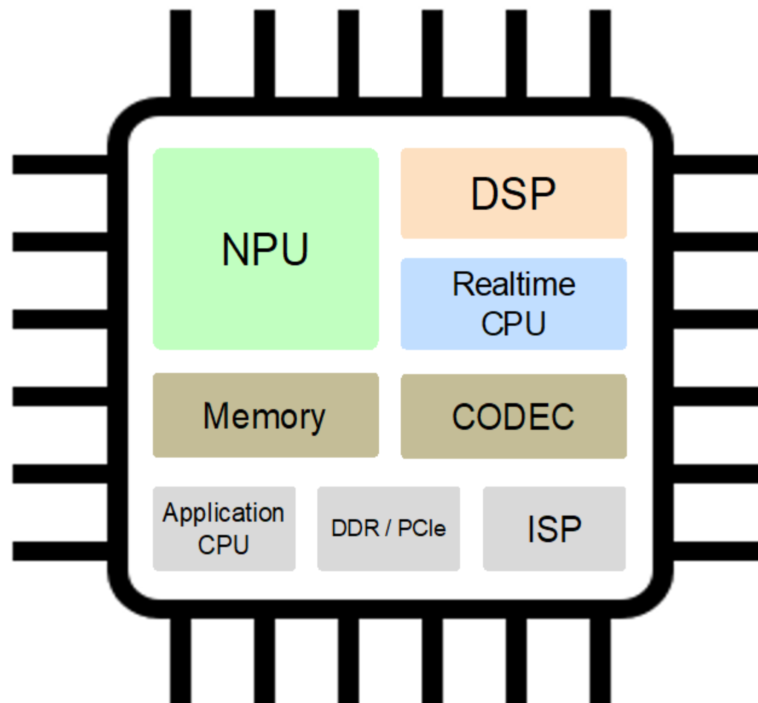

Figure 1. Conceptual block diagram of chip with NPU.

NPUs are optimized to handle matrix multiplication – the basic building block of convolutional neural networks (CNNs) that are widely used in ML applications today –

blazingly fast but usually cannot run any code other than the ML graph code they were specifically optimized to run. Most NPUs today are essentially large arrays of hardwired, fixed-point multiply-accumulate (MAC) blocks running in parallel. Most NPUs in silicon today can only support a few dozen common neural network operators (or "graph layers") – a handful each of convolution, pooling, and activation layers – among hundreds of operator types backed by the leading training frameworks such as Tensorflow and PyTorch.

The graph layers running on the NPU usually comprise 90-95% of the expected compute cycles consumed in today's most popular ML networks, and thus these ML accelerators do an admirable job of handling today's known inference workloads. Other less performance-critical operators are partitioned to run on the other processor engines in the system, delivering acceptable system performance at the cost of upfront engineering effort.

What Do the DSP and Realtime CPU Do?

DSPs are vector processor cores intended to handle a wide variety of complex math operations efficiently. They are widely used in various applications requiring signal processing – voice or image pre-processing are prime examples.

DSPs can be used to handle some matrix computations, but they are not optimized for them and tend to be inefficient compared to the matrix-optimized NPU accelerators described previously. Additionally, DSPs are typically highly utilized in the system running more conventional C code for signal pre- and post-processing, and thus have little performance headroom to handle heavy matrix computation. Therefore DSPs in most SoCs can augment the NPUs and run some but not all ML graph operators that the NPU doesn't natively handle.

The realtime CPU is responsible for controlling the overall inference functionality in the SoC. It coordinates ML inference workloads between the NPU, DSP, and the memory (used to store model weights). The realtime CPU is often the only programmable core in the inference subsystem that is exposed to the programmer. Because building and deploying multicore software development kits (SDKs) is a complex task, and because using a multicore SDK requires a complex learning cycle, most semiconductor vendors who employ CPU+DSP+NPU inference subsystems only expose the CPU to the developer for developer code, providing access to the DSP and NPU only via prebuilt application programming interfaces (APIs). If a developer needs an ML operator not supported in the APIs for the NPU or DSP, they can add a new ML operator on the CPU but generally not on the NPU or DSP.

Because CPUs are general purpose, they can functionally run any code the programmer desires, but because they lack the vector performance of a DSP or the

matrix performance of an NPU, CPUs are poor performers for new ML operators. The programmer thus must choose between high-performance ML operators prebuilt with published APIs or slow ML operators added to the CPU.

Distinctions must be made between the realtime CPU and an application-class CPU, both of which are shown in the conceptual block diagram in Figure 1. The latter is the larger CPU core running a complex operating system such as Linux, the application, and many other managerial functions. It usually has little involvement in real-time sensitive ML computations.

Challenges of an NPU + DSP + Realtime CPU Architecture

The following are just a few challenges that SoC developers are faced with and how they are presently addressed:

Future Proof Designs

Building SoCs that can handle known challenges is a good start but insufficient. The real challenge is to develop devices that are flexible enough to support some range of future requirements.

ML technology is evolving rapidly. New models, libraries, and operators are introduced at a rapid pace. This makes it essential to develop devices optimized for ML inference that can be programmed to support new operators and algorithms when they become available.

The existing heterogeneous SoC architecture approach described above is often not flexible enough to support new operators with the performance required. This is due to the inflexibility of hardwired NPUs that cannot be reconfigured. Developers tackle this challenge by adding code to the DSP or the realtime CPU to compensate for the NPU's shortcomings.

This approach is suboptimal in performance and creates a new set of problems. For example, splitting matrix operations between two disparate cores (NPU and CPU) penalizes inference latency and power dissipation since large data blocks have to traverse the chip going from one core to the other.

Multiple Toolchains

Dealing with multiple IP cores from multiple IP vendors invariably leads to reliance on multiple toolsets, creating many challenges. It is exceedingly difficult to debug a system using more than one debugger. As an example, it is virtually impossible to find quick answers to common debugging questions such as:

- Where is the system bottleneck?
- Why can't I get the throughput that I expected?

- Why does inference latency vary so drastically
- Is this problem a software bug or hardware issue?

Unfortunately, presently there are no easy ways to address this problem. Diversity in toolsets invariably leads to longer development times.

What's a GPNPU?

A general-purpose neural processing unit (GPNPU) uses a unified processor architecture that can handle matrix and vector operations and scalar (control) code in one execution pipeline. These workloads are traditionally handled separately by the NPU, DSP, and realtime CPU. The entire architecture is a single software-controlled core, allowing for the simple expression of complex parallel workloads.

You can consider a GPNPU to be a hybrid combination of the NPU, DSP, and realtime CPU (see Figure 2). A GPNPU is entirely driven by code – both traditional DSP C++ code and ML graph code - empowering developers to continuously optimize the performance of their models and algorithms throughout the device's lifecycle.

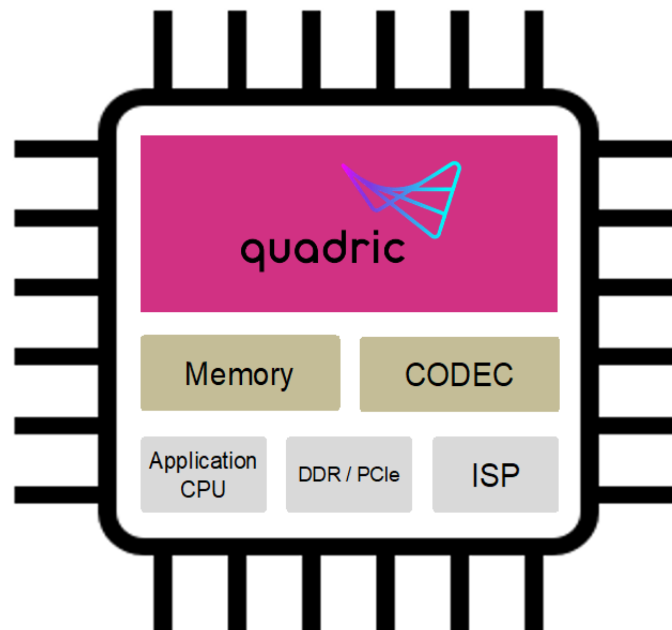

Figure 2. Conceptual block diagram of an SoC with a Quadric GPNPU

Why Use a GPNPU instead of a NPU + DSP + Realtime CPU?

There are several benefits of using a GPNPU:

System Simplicity

A GPNPU enables hardware developers to instantiate a single core that can handle an entire ML workload plus the typical DSP pre-processing and post-processing, signal conditioning workloads often intermixed with ML inference functions. Dealing with a single core drastically simplifies hardware integration and eases performance optimization. System design tasks such as profiling memory usage to ensure sufficient off-chip bandwidth are greatly simplified.

Programming Simplicity

With the right software, a GPNPU dramatically simplifies software development since matrix, vector, and control code can all be handled in a single code stream. ML graph code from the common training toolsets (Tensorflow, Pytorch, ONNX formats) is compiled by the software development toolkit (SDK) and can be merged with signal processing code written in C++, all compiled into a single code stream running on a single processor core.

The GPNPU SDK can meet the demands of both hardware and software developers, who no longer need to master multiple toolsets from multiple vendors. The entire subsystem can be debugged in a single debug console. This can dramatically reduce code development time and ease performance optimization.

This new programming paradigm also benefits the end users of the SoCs since they will have access to program all the GPNPU core resources.

Future Proof Flexibility – With High Performance

A Quadric GPNPU can run any algorithm written in C++ using a Compute Library (CCL) API. The CCL API enables programmers to rapidly express an algorithm in C++ that fully utilizes the high-performance matrix and vector capabilities of the GPNPU. Unlike other IP solutions that tout “future proof” capability that runs new user code on a slow DSP or CPU, a GPNPU runs user-written ML operators or customer C++ kernels at the same high-speed, highly parallel performance levels as the “native” operators.

This delivers future-proof flexibility *with* high-performance. This is incredibly powerful for the SoC design team since SoC developers can quickly write code to implement new

neural network operators and libraries long after the SoC has been taped out. This eliminates fear of the unknown and dramatically increases a chip's useful life.

Again, this flexibility is extended to the end users of the SoCs. They can continuously add new features to the end products, giving them a competitive edge.

All specifications are subject to revision

For more information, please contact Quadric or visit www.quadric.io. Quadric is a registered trademark and Chimera is a trademark of Quadric Inc. All content copyright © 2022 Quadric Inc.