

Does the Raspberry Pi 4B Have a GPGPU?

Jaidon Lybbert, University of Washington

January 24, 2024

Abstract

The Raspberry Pi 4B uses the Broadcom BCM2711 SoC, which features a quad-core ARM Cortex-A72 CPU, and a Broadcom VideoCore VI GPU. The VideoCore VI GPU is purpose-built for graphics applications, and there is very limited support for directly programming the device from the manufacturer. The focus of this article is to investigate the capabilities of the VideoCore GPU for General Purpose GPU (GPGPU) tasks, that is, compute tasks not involving graphics. Several open-source methods of programming are evaluated, and performance of the GPU is benchmarked against the CPU for the General Matrix Multiply (GEMM) algorithm extensively used in machine learning and machine vision applications.

1 Introduction

This document is organized as follows. In Section 2 I describe where the VideoCore GPU sits in the context of the computing world, and introduce the most significant open-source efforts to program it. I explain the baselines we have to compare the VideoCore to, the benchmarks performed, and walk through the process of getting something to run on the GPU in Section 3. I show the benchmark results in Section 4, and discuss the results and scaling in Section 5. I conclude in 6 by discussing the overall takeaways from my experiments and things I would like to try in the future.

2 Background

2.1 GPUs and GPGPUs

As clock rates stopped increasing, and Moore's law began to taper off, parallel processors emerged to increase compute throughput by allowing some algorithms to be broken down into pieces that could be solved simultaneously. Fueled by the \$300B gaming industry, parallel processors have rapidly taken over for video-encoding and decoding in computers and mobile phones. Nearly anything displaying to a screen now has a graphics processor. The emergence of AI training and inference has created new demand for General Purpose GPUs (GPGPUs) which are not restricted to graphics processing, but can be easily programmed for other compute tasks. NVIDIA's CUDA platform is built for this market, making programming their family of GPUs easy by providing a language, compiler, and debugging tools that integrate with existing languages and tools like C++, and Visual Studio.

Machine vision algorithms like Simultaneous Localization And Mapping (SLAM), and other applications, demand portable GPGPUs which can be programmed with arbitrary code and be integrated into a small portable device to perform parallel processing. The NVIDIA Jetson family of devices is tailored to this market, offering scaled down chips, and development boards for embedded programmers as low as \$150.

In this article, we will look at the GPU on the Raspberry Pi. A device which, as we'll see, was clearly not intended to be used as a GPGPU. However, recent developments in software drivers and graphics APIs just might enable this device to do useful compute workloads on the GPU.

2.2 VideoCore IV and VI

The Broadcom chips in the Raspberry Pis have VideoCore GPUs. The Raspberry Pi 1, 2, and 3 have the VideoCore IV GPU, and the Raspberry Pi 4 has the VideoCore VI GPU. The VideoCore IV is also found in devices such as the Roku, Amazon Fire TV Stick, and phones from Samsung, Nokia, and Apple's iPod.

We will focus on the BCM2711 SoC featured in the Raspberry Pi 4B, which has a VideoCore VI GPU.

2.3 Motivation

Two big questions motivate my research: “what is the performance of the VideoCore VI GPU?” and “how do I program it?”

This small list of large questions breaks down into a large list of smaller questions.

- What is the compute throughput of the VideoCore VI GPU?
- How do instructions map to hardware?
- How many parallel threads does the GPU support?
- What other APIs exist for GPU programming?
- Previous work done and results
- What is the memory model for the GPU + CPU?
- CPU and GPU caching?
- Supported language bindings?
- Memory latency?

2.4 Prior Works

Broadcomm has not released much documentation for the VideoCore VI GPU. Much of the information about the GPU is derived from the previous VideoCore IV reference manual [1], which Broadcomm did release, and open-source drivers written with the help of Broadcomm engineers in the Linux kernel repository [14], and the MESA graphics library [7].

Various projects over the years have attempted to make programming the GPUs easier on the Raspberry Pi.

The GitHub user “hermanhermitage” investigated the VideoCore IV GPU, and created a wiki their findings [5]. They used US patent applications filed by Broadcomm along with experimentation to characterize the architecture, memory layout, instruction set, and registers for the device. The project includes a guide on intercepting GPU execution by replacing the `bootcode.bin` in the boot partition of the SD card. Broadcomm eventually published a full

specification for the VideoCore IV, superceding this effort with official documentation.

QPULib is a C++ library by Matthew Naylor for compiling and offloading programs for the VideoCore IV GPU (Raspberry Pi versions 1, 2, 3) during runtime [10]. Efforts by Wim Rijnders extended this idea to the VideoCore VI in the Raspberry Pi 4, with the C++ library V3DLib [13]. V3DLib depends on the Mesa graphics library for disassembly of VideoCore instructions. It’s worth noting that neither of these projects have had active development in the last 3 years, and support for new OS versions is unknown.

A totally separate project, `py-videocore6` [6], is a Python library enabling the programming of the VideoCore VI through the Direct Rendering Manager (DRM) graphics driver in the Linux kernel.

The Raspberry Pi Foundation recently announced that the Raspberry Pi 4 is now Vulkan 1.2 compliant through the Mesa graphics driver. This was a joint effort by Raspberry Pi, Mesa, and Broadcom developers. Vulkan is a graphics library, which in contrast to older graphics libraries like OpenGL ES, supports an efficient compute-only pipeline, without any involvement of the framebuffer. Vulkan has been getting used more recently in machine learning applications for its cross-platform support.

3 Methods

3.1 Setting the Baseline

To evaluate the performance of the VideoCore VI GPU, there are two useful comparisons:

- How does it perform compared to the CPU on-chip?
- How does it perform compared to an NVIDIA Jetson?

The first question is arguably more relevant to someone who already has a Raspberry Pi. After all, why go through the trouble if the CPU is way faster anyway?

3.2 CPU GMEM Benchmarks

As a benchmark, I chose to use a naive matrix multiply algorithm, as it’s easy to implement and analyze. The algorithmic complexity is $O(n^3)$. For

the basic CPU version, I use Bing AI to generate the algorithm in Rust which iterates over many matrix sizes, and records their execution times to a `.csv` file. I then use Bing AI to generate a Python function to plot the results from the `.csv` file. See the Appendix for Bing chat logs. From there, I debugged the boilerplate code with the help of the *rust-analyzer* and *pylsp* in my code editor, and modified it to my liking.

3.3 Cortex-A72 Characterization

The ARM Cortex-A72 CPU is thoroughly documented by ARM, with a public reference manual [8]. Details specific to the Raspberry Pi 4B are published in the Raspberry Pi documentation [11]. The relevant high-level details are summarized here.

3.3.1 CPU Architecture

The Cortex-A72 CPU in the Raspberry Pi 4B has 4 cores, each with 32kB data cache and 48kB instruction cache. The cores share a 1MB L2 cache. The organization is shown in Fig 1.

The CPU supports SIMD operations through the Advanced SIMD and FP feature, has a Translation Lookaside Buffer (TLB), and has a module for prefetching memory, as shown in Fig 2. Using the 128-bit SIMD pipeline, for 4 cores it is theoretically possible to do 24 GFLOPS at 1.5GHz with single-precision.

3.3.2 Memory

- 32kB data + 48kB instruction L1 cache per core
- 1MB L2 cache

3.4 VideoCore VI Characterization

The VideoCore VI on the Raspberry Pi 4B is a 32-bit vector processor that operates on 16-element 32-bit integer or floating-point registers. These registers are 16-elements deep, meaning each register addresses 16 32-bit values. The processor is split into 8 Quad Processing Units (QPU), organized into two “slices” of 4 QPUs, which each have 4 physical cores Fig 3. Each physical core contains an add ALU and a multiply-add ALU Fig 4 [13].

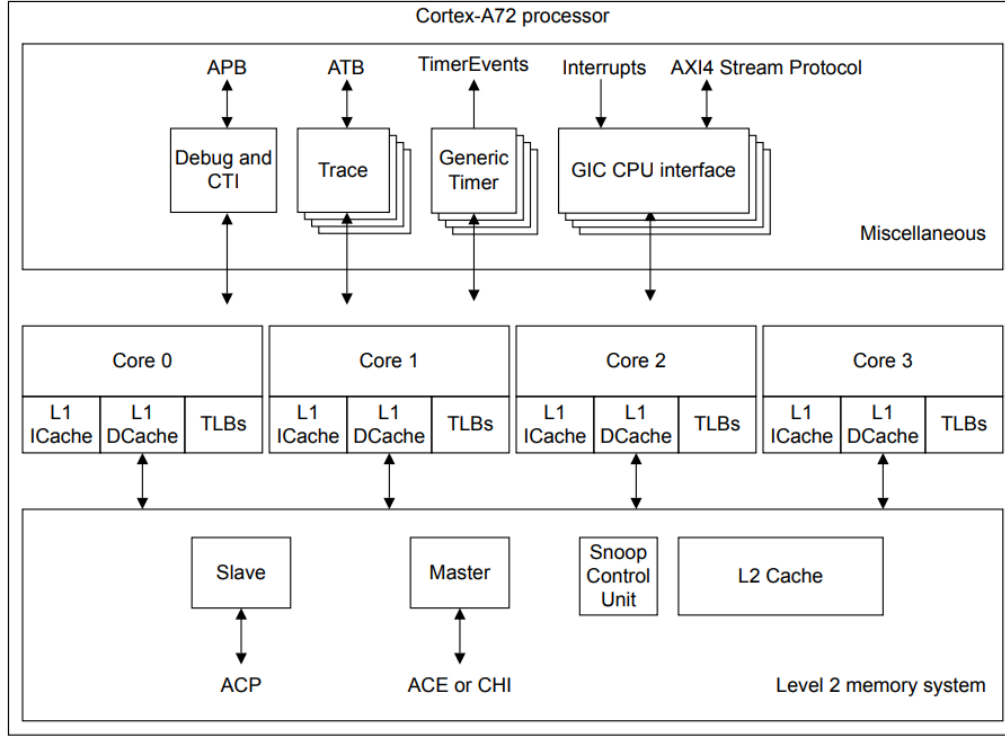


Figure 1: Block diagram of the quad-core ARM Cortex-A72 CPU.

This puts the theoretical throughput of the device at 32 GFLOPS [6].

3.5 Programming the VideoCore VI

I found three open-source projects which could allow programming the VideoCore VI with compute workloads. I attempt to install each of them here.

3.5.1 V3DLib

V3DLib is a language and compiler for the VideoCore VI GPU created by Wim Rjinder's in C++. The project has a large number of dependencies, and ultimately uses the Mesa graphics library to translate code to run on the device. Because of the large number of dependencies and stale development (hasn't been updated in 3 years), it may be prohibitively difficult to use this

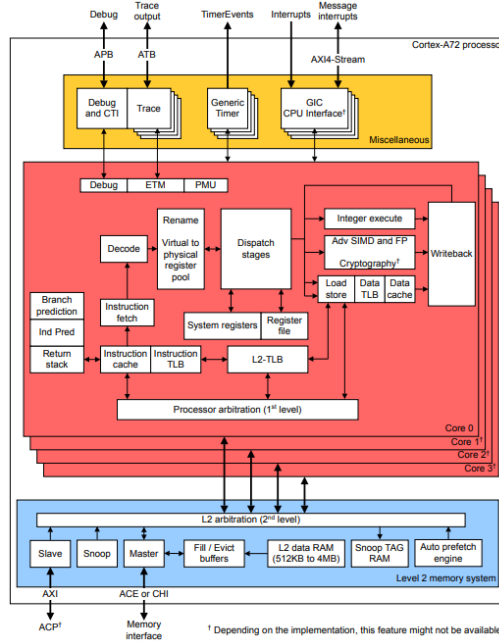


Figure 2: Functional block diagram of the quad-core ARM Cortex-A72 CPU.

method of programming.

I cloned the repository and ran the build script.

```
> sudo apt-get install git
> sudo apt install libexpat1-dev
> git clone --depth 1 https://github.com/wimrijnders
/V3DLib.git
> cd V3DLib
> make QPU=1 DEBUG=1 all
```

The build failed due to missing headers, which I found were included in the raspberry pi development headers. I installed those, and the build failed again due to some XML parsing library missing. Since the build was taking a while, and it seemed like the project had long since been abandoned, I stopped here, to try the other options.

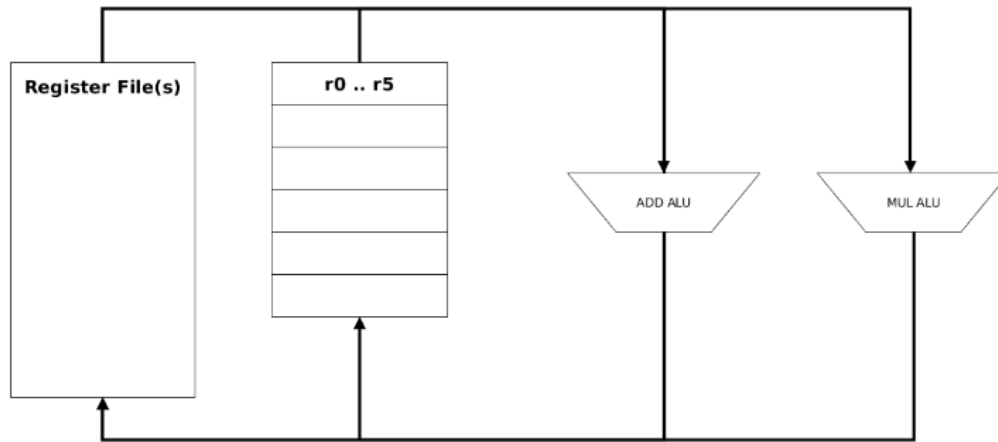


Figure 3: Block diagram showing one physical core of the VideoCore processor.

3.5.2 pyvideocore6

I followed the instructions in the GitHub repository to install the Python package.

```
sudo usermod --append --groups video clusterpi
mkdir PyVideoCore
cd PyVideoCore
python3 -m venv venv
source venv/bin/activate
pip3 install setuptools wheel
git clone https://github.com/Idein/py-videocore6.git
cd py-videocore6/
python3 -m pip install --target sandbox/ --upgrade .
nose
```

```
python3 setup.py build_ext --inplace
PYTHONPATH=sandbox/ python3 -m nose -v -s
```

The tests failed to run with the error message

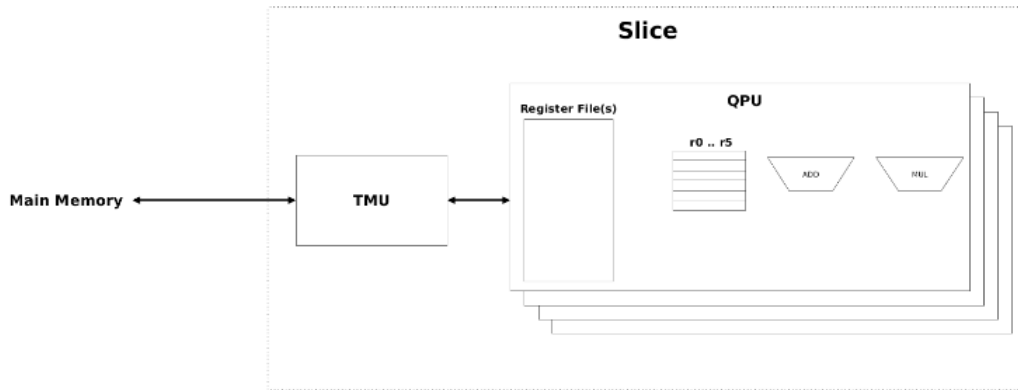


Figure 4: Block diagram showing the architecture of the VideoCore processor.

```
File "/home/clusterpi/PyVideoCore/py-videocore6/
    sandbox/nose/suite.py", line 106, in _set_tests
    if isinstance(tests, collections.Callable) and
        not is_suite:
```

```
~~~~~
AttributeError: module 'collections' has no
    attribute 'Callable'
```

I quickly found that 'Callable' had been moved to 'collections.abc.Callable' in newer versions of Python. To fix this I simply added this line to `suite.py` under the import statements.

```
collections.Callable = collections.abc.Callable
```

All tests then ran without issue. I then ran the SGEMM example. The results are shown in the next section.

```
PYTHONPATH=sandbox/ python3 examples/sgemm.py
```

With more investigation, I discovered that the project developers had created a Basic Linear Algebra Subroutine (BLAS) library using this assembler. This project has more types of SGEMM implementations, among other algorithms. I attempted to install this BLAS library to test it out.

First I cloned and built the dependency.

```
git clone https://github.com/Idein/libdrm_v3d
```

```
cd libdrm_v3d/  
cmake .  
make  
ctest -V
```

The tests passed.

I then installed the package.

```
make package  
sudo dpkg -i libdrm_v3d_0.0.0_arm64.deb
```

I then compiled the BLAS library from the source repository, and ran the tests. The SGEMM results are shown in the next section.

```
git clone https://github.com/Idein/qmkl6  
cd qmkl6/  
cmake .  
sudo PYTHONPATH=~/.PyVideoCore/py-videocore6/sandbox/  
make  
ctest -V
```

3.5.3 Vulkan 1.2

For my final test of programming the GPU, I will use the Vulkan 1.2 compliant Mesa graphics library driver. The Mesa driver for the VideoCore VI itself uses the Linux kernel V3D device driver to communicate with the device, just like `py-videocore6` [3]. In contrast to `py-videocore6`, programming the GPU using the Vulkan 1.2 method uses the well-documented cross-platform Vulkan API instead of the one-off custom assembler that `py-videocore6` uses. Since the Vulkan API is standard across all platforms, a wealth of knowledge and open-source exists for programming this way. To benchmark the device, I will use Google’s `uVkCompute` project. Which is described as “a micro Vulkan compute pipeline and a collection of compute shaders for benchmarking/profiling purposes[4].”

I started by downloading, compiling, and installing the latest Mesa driver (24.0.0) following the Mesa 3D documentation [3]. I used GCC 12.2.0 and Python 3.11.2. I ran into many dependency issues, and found an article by Qengineering on building Mesa specifically for the Raspberry Pi 4. I used this guide as a reference [12]. Some of the dependencies listed in the guide

are no longer available on the default apt repository, and I simply skipped them.

At the install step of the guide, my process was slightly different.

```
mkdir mesa
cd mesa
python -m venv venv
source venv/bin/activate
pip install meson mako
git clone -b 24.0 https://gitlab.freedesktop.org/
    mesa/mesa.git mesa_vulkan
cd mesa_vulkan

CFLAGS="-mcpu=cortex-a72" \
CXXFLAGS="-mcpu=cortex-a72" \
meson --prefix /usr \
-D platforms=x11 \
-D vulkan-drivers=broadcom \
-D gallium-drivers=kmsro,v3d,vc4 \
-D buildtype=release build

ninja -C build -j4
```

The build took about 45 minutes, and finished without error. I installed the binaries.

```
sudo ninja -C build install
```

I originally intended to install the uVkCompute project from Google, but the installation process for the Vulkan SDK on the Raspberry Pi deterred me. The developers of Vulkan SDK provide an installer for x86 Linux, but don't have information on building for the `aarch64` architecture. It is certainly possible to do, but the path is unclear. I instead opted to compile and test a sample project showcasing headless Vulkan from a popular set of Vulkan samples (137 contributors, 9.5k stars on GitHub) from Sascha Willems [15]. This project contains everything needed to build the sample projects with CMAKE and a compiler, including the Vulkan libraries. This is also used by the Q-engineering install guide [12] so I knew it would work.

```
git clone --recursive https://github.com/
    SaschaWillems/Vulkan.git
```

```

cd Vulkan
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make -j4

```

The project took about 10 minutes to build with no errors. I then ran the headless compute example. This example computes a fibonacci sequence in the Vulkan compute shader on the GPU and returns the results to the shell output.

```

cd bin
./computeheadless

```

The program ran successfully with the output in Fig 5, validating Vulkan as a method of programming the Raspberry Pi 4B GPU.

```

Running headless compute example
GPU: V3D 4.2.14
Compute input:
0      1      2      3      4      5      6      7      8      9
      10     11     12     13     14     15     16     17     1
8      19     20     21     22     23     24     25     26     2
7      28     29     30     31
Compute output:
0      1      1      2      3      5      8      13     21     3
4      55     89     144    233    377    610    987    1597    2
584    4181    6765    10946   17711   28657   46368   75025   121393
      196418  317811  514229  832040  1346269
Finished. Press enter to terminate...

```

Figure 5: Output of the headlesscompute Vulkan sample from Sascha Willems, running on the Raspberry Pi 4B through the Mesa graphics library driver.

4 Results

4.1 CPU GEMM Performance

I implemented a naive GEMM algorithm in Rust, and compared my implementation to the Numpy implementation, which uses all cores and the SIMD

engine. The difference was outstanding. See Fig 6, 7. My implementation was orders of magnitude slower. In the time my Rust implementation multiplied two 200x200 matrices, Numpy could do two 2000x2000 matrices. This is particularly impressive considering the $O(N^3)$ complexity of the algorithm. The Numpy algorithm equates to 8 GFLOPS, while my Rust algorithm performs at just 8 MFLOPS.

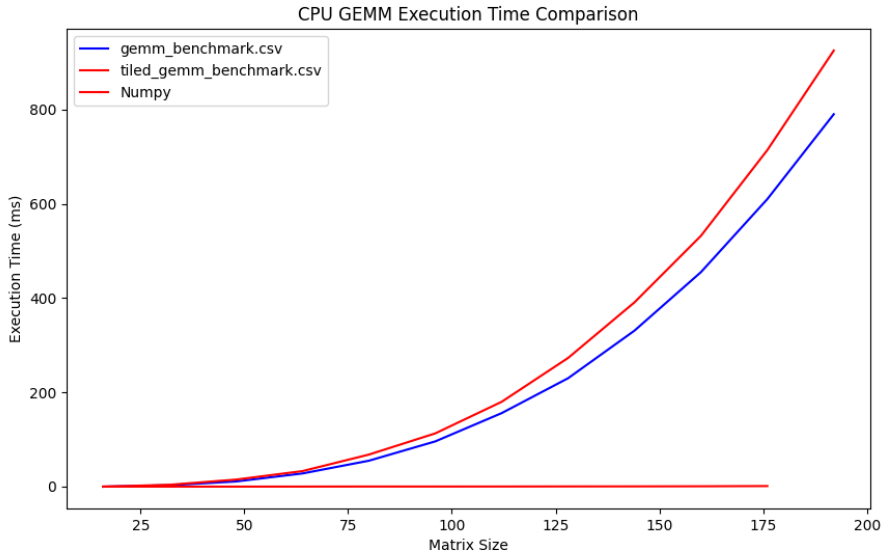


Figure 6: Comparison of 3 CPU GEMM implementations, the slow ones are mine implemented in Rust. The Numpy implementation is the “@” operator in the Python library Numpy.

4.2 GPU Theoretical Comparison to the CPU

The CPU and GPU share memory on chip, the performance differences will come down to how well they cache and manage memory and their throughput. I will ignore the memory considerations, and simply compare their theoretical compute throughput to get a ballpark estimate of how they compare. Although the VideoCore does have cache, it is unknown how much, since that information is not published.

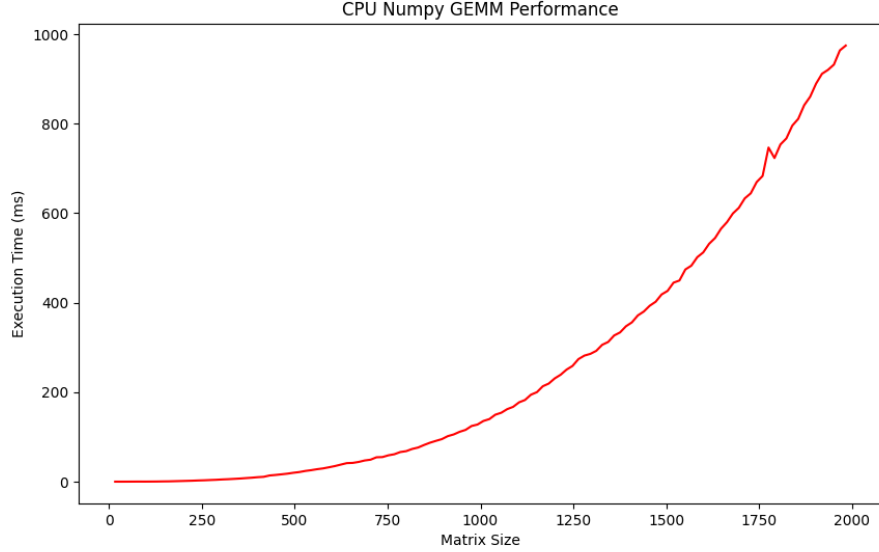


Figure 7: Numpy CPU implementation of GEMM using the all cores and the SIMD engine of the A72.

Property	VideoCore VI	ARM A72
FLOPS	32 GFLOPS	24 GFLOPS

4.3 GPU Benchmark Comparison to CPU

I used the py-videocore6 SGEMM test to compare CPU and GPU performance.

```
==== sgemm example (1024x1024 times 1024x1024) ====
numpy: 0.0961 sec, 22.38 Gflop/s
QPU:   0.5735 sec, 3.75 Gflop/s
Minimum absolute error: 0.0
Maximum absolute error: 0.0003814697265625
Minimum relative error: 0.0
Maximum relative error: 0.13134673237800598
```

These GPU results are consistent with the test results reported by the project developers, but in their case, Numpy had similar performance to the

GPU. It would seem that the Numpy library has been updated since then, using the SIMD hardware of the Cortex-A72 to speed up the matrix multiply, and achieve very near the theoretical peak performance.

I also tried the qmlk6 BLAS library implementation of SGEMM.

The peak throughput of the GPU during the qmlk6 SGEMM tests reached 5.8 GFLOPs, Fig 8.

```
10: CNT: m = 512, n = 512, k = 1024, alpha = 2.000000,  
lda = 512, ldb = 512, beta = 3.000000, ldc = 512: err_  
abs = [0.000000, 0.001038], err_rel = [0.000000, 0.000002],  
0.092633 sec, 5.804143 Gflop/s
```

Figure 8: Best compute throughput results from running the qmlk6 tests. The meaning of the various parameters are common usages in BLAS libraries. In this case we are multiplying a 512x1024 matrix A by a 1024x512 matrix B to produce a 512x512 result C. The parameters lda, ldb, ldc refer to the “leading dimension of [a, b, c].”

4.4 Theoretical Comparison to NVIDIA Jetson Nano

In order to make meaningful observations about the capability of the Broadcomm VideoCore VI GPU on the Raspberry Pi 4B, I will compare the programming experience and capabilities of the device to the widely used NVIDIA CUDA platform. Obviously, with the NVIDIA devices being more expensive, they have many more cores and features, but on a core-to-core basis they are similar SIMD processors with similar function. For comparisons sake, I will show how the \$35 2GB Raspberry Pi 4B **theoretically** stacks against the \$150 NVIDIA Jetson Nano device.

4.4.1 Hardware

Property	Raspberry Pi 4B	Jetson Nano
GPU	Broadcomm VideoCore VI	128-core NVIDIA Maxwell
CPU	Quad-core ARM A72	Quad-core ARM A57
Memory	4GB LPDDR4	4GB LPDDR4
Memory Bandwidth	12.8GB/s	25.6GB/s
GPU GFLOPS	32	472
Power Usage (W)	3.4 — 7.6	5 — 10
Cost	\$35	\$130

Note *Values are taken from Raspi documentation [11], published benchmarks [16], and the NVIDIA product page [2]*

4.4.2 Software

NVIDIA NVIDIA provides proprietary drivers, the CUDA language, the NVCC compiler, the NSIGHT debugger, and profiling tools with NSIGHT Systems. Their drivers are compliant with OpenCL, Vulkan, and OpenGL standards. There is a large amount of open-source example code provided by NVIDIA for using these APIs, and interoperability. There is CUDA integration in the most well-known ML libraries like TensorFlow and PyTorch.

VideoCore Broadcomm doesn't provide any device driver for the VideoCore GPU. They officially published the specification for the VideoCore IV, but never the VI or VII in the Pi 4 and 5. All software libraries I have found ultimately communicate with the VideoCore through the Linux kernel V3D DRM device driver, which supports OpenGL ES. The open-source Mesa device driver is a wrapper around the DRM driver for the VideoCore VI and recently became Vulkan 1.2 compliant. Software development for programming the GPU outside of graphics is sparse and outdated. The options are to either program the GPU with a custom open-source assembler maintained by 1 person, or install the Mesa driver and program it using Vulkan compute shaders. No working language, debugger, or profiling tool exists as far as I can tell.

Verdict NVIDIA wins the battle in the world of software for GPGPU development, and it's not even close. The VideoCore GPU, true to its name, was designed for video and graphics, and Broadcomm never intended for it to be used for GPGPU tasks.

5 Discussion

5.1 Cortex-A72 vs VideoCore VI

While the theoretical 32 GFLOPs compute throughput of the VideoCore VI is higher than the 24GFLOPs of the A72, in practice, directly programming the VideoCore VI through py-videocore6 or qmkl6 resulted in SGEMM performance that was not even 1/6 the theoretical performance, while the Numpy CPU implementation very nearly achieved the theoretical peak of 24GFLOPS. Ultimately, the GPU was at best capable of 1/4 the compute throughput of the CPU.

5.2 Raspberry Pi 4B vs Jetson Nano

5.2.1 Performance

The Raspberry Pi 4B and Jetson Nano both feature a Quad-core ARM CPU, and 4GB LDDR4 for the models and price point chosen. The Jetson Nano claims double the memory bandwidth compared to the benchmarked Raspberry Pi 4B results, but I couldn't find any true benchmarks for the Nano. The power usage is slightly higher for the Nano.

The benchmarks show that the practical compute throughput of the VideoCore VI is about 5.8GFLOPS at best, and usually much lower. This pales in comparison to the theoretical 472 GFLOPS the Jetson Nano is capable of. If we assume that both devices are running compute-bound workloads like the SGEMM (a poor assumption), so that memory access time isn't a bottleneck, the Nano outperforms the Pi 4B by a factor of 81.4, while being 4.2x the cost.

The Real World of Memory-bound Workloads In reality, workloads are not compute-bound in general. One metric to analyze how much an algorithm is compute or memory-bound is the **arithmetic intensity** of the algorithm defined as the ratio of arithmetic operations to memory loads and

stores. Algorithms are designed to be ideally compute-bound by maximizing the number of arithmetic operations and minimizing the number of memory operations. Despite incredible efforts optimizing memory access patterns, it is actually quite rare to achieve the compute-bound case.

The general matrix-multiply (GEMM) algorithm is incredibly useful, and is great for GPUs because it can be made to have good arithmetic intensity, and be compute-bound with fast memory and good caching. In the ideal case with infinite cache, given an $N \times N$ matrix, for $2N^2$ memory reads, you perform N^3 arithmetic operations for an arithmetic intensity $N/2$, so the best case is very large matrices. In practice, this value must be much much greater than ratio of memory latency to the time it takes to do one arithmetic operation in order for the workload to be compute-bound. And of course, cache is not infinite.

All this is to say that the compute throughput of the NVIDIA Nano only really means anything if you're doing specific algorithms like AI training or machine vision, and little to nothing if you're talking about serving web requests.

At Scale If we look at a compute cluster made up of 400 Raspberry Pi 4Bs and compare to another made of 100 Nvidia Jetson Nanos, each slaving away training a neural net by doing a bunch of very large SGEMMs, the comparison would look something like the table below.

Property	Pi Cluster	Nano Cluster
Nodes	400	100
GPU	Broadcomm VideoCore VI	128-core NVIDIA Maxwell
CPU	Quad-core ARM A72	Quad-core ARM A57
Memory	1.6TB LPDDR4	400GB LPDDR4
Memory Bandwidth	5.12TB/s	2.56TB/s
GPU TFLOPS	2.32	47.2
CPU TFLOPS	9.6	2.4
Total TFLOPS	11.2	49.6
Power Usage (kW)	1.3 — 3.0	0.5 — 1.0
Cost	\$14k	\$15k

At scale, the Pi cluster begins to look like less of a machine-learning cluster, and more like a data or web server. The strengths of the Pi is the

cheap access to a large number of processors and RAM, while the Nano would perform massively better for AI workloads.

5.2.2 Programming

The `qmk16` BLAS library performed well compared to the `py-videocore6` example. The BLAS library abstracts away the assembly instructions needed to use the `py-videocore6` assembler to program the device, and can be called directly from C++ code. `qmk16` is also easier to install and more lightweight than Vulkan. It would be my preferred method of programming using the GPU if I were to do single-platform compute-only tasks consisting only of BLAS routines. There are some minor issues installing the project due to no maintainence.

Mesa was a significant effort to install and requires a large amount of dependencies and disk space to get going. The benefit of Mesa is enabling Vulkan as a standard cross-platform API, which can also be used for graphics. Vulkan would generally be my preferred choice for programming the VideoCore, for the cross-platform code, and interoperability with graphics. By enabling Vulkan, the programming experience vastly improves in reliability and wealth of knowledge and open-source implementations of algorithms online.

6 Conclusion

So, does the Raspberry Pi 4B have a GPGPU? Clearly, the answer is no. There is no support from the manufacturer for programming the GPU directly. What does exist is a few stagnant open-source projects made by a very few curious people who reverse-engineered the device from patent filings, disassembling OpenGL ES shader code, and inferring details from benchmarking and overdue specifications from previous generation chips. The GPU is slower than 1/4 the speed of the A72 using SIMD at the GEMM algorithm. Everything points to leaving the GPU for graphics tasks, which it has specialized hardware for.

However, there may still be value in using this VideoCore chip as if it were a GPGPU for select applications. After all, you have an entirely separate vector processor on the device capable of GFLOPs, why not use it if it's easy enough? Maybe there is some long running process that you can offload to

your GPU to leave the CPU free for other tasks. Something that comes to mind is SLAM bundle adjustment. Bundle adjustment is an optimization step that executes on a large matrix of collected keypoints. It isn't time-critical like instantaneous pose-estimation, so it is okay to take seconds or minutes to perform. Offloading this algorithm to the GPU would free the CPU to service control interrupts and rapidly compute pose-estimation.

Note *All my code and project files for this and future reports, can be found on my GitHub repository [9].*

References

- [1] Broadcom Corporation. Videocore® iv 3d architecture reference guide. Technical Report VideoCoreIV-AG100-R, Broadcom Corporation, 2013.
- [2] NVIDIA Corporation. Jetson store. <https://developer.nvidia.com/embedded/jetson-developer-kits>, 2024.
- [3] Mesa Developers. Mesa 3d documentation. <https://docs.mesa3d.org/drivers/v3d.html>. Accessed: 2024-02-07.
- [4] Google. `pvkcompute`: A micro vulkan compute pipeline and a collection of benchmarking compute shaders. <https://github.com/google/uVkCompute>, 2024. Accessed: 2024-02-07.
- [5] hermanhermitage. videocoreiv: Tools and information for the broadcom videocore iv (raspberrypi). <https://github.com/hermanhermitage/videocoreiv>, 2024. Accessed: 2024-02-05.
- [6] Idein. py-videocore6: A python library for gpgpu programming on raspberry pi 4. <https://github.com/Idein/py-videocore6>, 2021. Accessed: 2024-01-30.
- [7] intel. external-mesa/src/gallium/drivers/v3d at master · intel/external-mesa. <https://github.com/intel/external-mesa/tree/master/src/gallium/drivers/v3d>, 2024. Accessed: 2024-01-30.
- [8] ARM Limited. Arm cortex-a72 mpcore processor technical reference manual. Technical report, ARM Limited, 2021.

- [9] Jaidon Lybbert. Classwork for embedded and real-time systems, university of washington, winter 2024. <https://github.com/jaidonlybbert/EEP522A-W24>, 2024. Accessed: 2024-01-23.
- [10] Matthew Naylor. Qpulib: A programming language and compiler for the raspberry pi’s quad processing units. <https://github.com/mn416/QPULib>, 2021. Accessed: 2024-01-30.
- [11] Raspberry Pi. Processors. <https://www.raspberrypi.org/documentation/hardware/raspberrypi/processors/README.md>, 2024. Accessed: 2024-02-07.
- [12] Q-engineering. Install vulkan on a raspberry pi 4. <https://qengineering.eu/install-vulkan-on-raspberry-pi-4.html>, 2023. Accessed: 2024-02-07.
- [13] Wim Rijnders. V3dlib: A c++ library for creating programs to run on the videocore gpu’s of all versions of the raspberry pi. <https://github.com/wimrijnders/V3DLib>, 2021. Accessed: 2024-01-30.
- [14] Linus Torvalds. linux/drivers/gpu/drm/v3d. <https://github.com/torvalds/linux/tree/master/drivers/gpu/drm/v3d>, 2024. Accessed: 2024-01-30.
- [15] Sascha Willems. Vulkan examples and demos. <https://github.com/SaschaWillems/Vulkan>, 2024. Accessed: 2024-02-07.
- [16] Rob Zwetsloot. Raspberry pi 4 specs and benchmarks. *The MagPi*, 2024.

7 Appendix: Bing Chat Log

User

write a general matrix multiply (GEMM) algorithm in Rust

Bing

Sure, here's a simple implementation of a General Matrix Multiply (GEMM) algorithm in Rust. This code multiplies two matrices *a* and *b* and stores the result in *c*.

```
fn gemm(a: &Vec<Vec<f64>>, b: &Vec<Vec<f64>>, c: &mut Vec<Vec<f64>>) {
    let n = a.len();
    let m = b[0].len();
    let p = a[0].len();

    for i in 0..n {
        for j in 0..m {
            let mut sum = 0.0;
            for k in 0..p {
                sum += a[i][k] * b[k][j];
            }
            c[i][j] = sum;
        }
    }
}
```

Please note that this is a basic implementation and does not include any optimizations. For large matrices, you might want to consider using a library like *ndarray* or *nalgebra* which are optimized for numerical computations. Also, this code assumes that the input matrices are well-formed and have compatible dimensions for multiplication. In a production setting, you would want to add checks for these conditions.

Let me know if you need further assistance! 😊

User

create a rust function to initialize a randomized m-by-n matrix to satisfy the parameters of this function header
`fn gemm(a: &Vec<Vec>, b: &Vec<Vec>, c: &mut Vec<Vec>) { let n = a.len(); let m = b[0].len(); let p = a[0].len();`

```
for i in 0..n {
    for j in 0..m {
        let mut sum = 0.0;
        for k in 0..p {
            sum += a[i][k] * b[k][j];
        }
        c[i][j] = sum;
    }
}

}
```

Bing

Sure, here's a function that creates a randomized m-by-n matrix in Rust. This function uses the *rand* crate, so you'll need to add `rand = "0.8"` to your *Cargo.toml* file.

```
use rand::Rng;
```

```
fn create_random_matrix(m: usize, n: usize) -> Vec<Vec<f64>> {
    let mut rng = rand::thread_rng();
    let mut matrix = vec![vec![0.0; n]; m];
    for i in 0..m {
        for j in 0..n {
```

```

        matrix[i][j] = rng.gen::();
    }
}
matrix
}

```

This function generates a matrix of size m by n filled with random floating-point numbers between 0 and 1. You can use this function to initialize the matrices a , b , and c for the `gemm` function. Please note that the dimensions of these matrices should be compatible for the multiplication operation in `gemm`. For instance, if a is an m by p matrix and b is a p by n matrix, then c should be an m by n matrix.

Let me know if you need further assistance! 😊

User

write a Rust function which creates two m -by- n matrices using the function `create_random_matrix(m: usize, n: usize) -> Vec<Vec>`, then passes them into the function `gemm(a: &Vec<Vec>, b: &Vec<Vec>, c: &mut Vec<Vec>)` and returns the execution time in milliseconds

Bing

Sure, here's a function that does that. This function uses the `Instant` struct from the `std::time` module to measure the execution time.

```

use std::time::Instant;

fn measure_gemm(m: usize, n: usize, p: usize) -> u128 {
    let mut a = create_random_matrix(m, p);
    let mut b = create_random_matrix(p, n);
    let mut c = vec![vec![0.0; n]; m];

    let start = Instant::now();
    gemm(&a, &b, &mut c);
    let duration = start.elapsed();

    duration.as_millis()
}

```

This function first creates two random matrices a and b of dimensions m by p and p by n respectively using the `create_random_matrix` function. It also initializes a zero matrix c of dimensions m by n . It then measures the time taken to execute the `gemm` function on these matrices and returns this time in milliseconds.

Please note that the actual execution time can vary depending on the system's load and the specific hardware used. Also, this function assumes that the `gemm` and `create_random_matrix` functions are defined in the same scope.

Let me know if you need further assistance! 😊

User

write a function that iterates over 100 matrix sizes from 10×10 to 1000×1000 and calls this function `fn measure_gemm(m: usize, n: usize, p: usize) -> u128` which returns the execution time in ms, using the values returned by `measure_gemm`, plot the results with execution time on the y-axis and matrix size on the x-axis

Bing

Sure, here's a function that does that. However, please note that Rust doesn't have built-in support for plotting. You would need to use a library like `plotters` or export the data to a CSV file and use a separate tool like Python's `matplotlib` or Excel to create the plot.

```

fn benchmark_gemm() {
    // Prepare a file to write the results
    let mut file = std::fs::File::create("gemm_benchmark.csv").unwrap();
    writeln!(file, "Matrix Size,Execution Time").unwrap();
}

```

```
// Iterate over matrix sizes from 10x10 to 1000x1000
for size in (10..=1000).step_by(10) {
  let time = measure_gemm(size, size, size);
  println!("Size: {}x{}, Time: {} ms", size, size, time);
  writeln!(file, "{},{},", size, time).unwrap();
}

println!("Benchmark results written to gemm_benchmark.csv");
}
```

This function writes the matrix size and corresponding execution time to a CSV file named `gemm_benchmark.csv`. You can then use this file to create a plot with any tool you prefer. The x-axis should represent the matrix size and the y-axis should represent the execution time.

Let me know if you need further assistance! 😊

User

write a python script which reads a CSV with values formatted as “matrix size, execution time” and plots the results with matrix size on the x-axis and execution time on the y-axis

Bing

Sure, here’s a Python script that does that. This script uses the pandas library to read the CSV file and matplotlib to create the plot.

```
import pandas as pd
import matplotlib.pyplot as plt

# Read the CSV file
data = pd.read_csv('gemm_benchmark.csv')

# Create the plot
plt.figure(figsize=(10, 6))
plt.plot(data['Matrix Size'], data['Execution Time'], marker='o')
plt.title('GEMM Execution Time')
plt.xlabel('Matrix Size')
plt.ylabel('Execution Time (ms)')
plt.grid(True)
plt.show()
```

This script assumes that the CSV file is named `gemm_benchmark.csv` and is located in the same directory as the script. The CSV file should have two columns named ‘Matrix Size’ and ‘Execution Time’. The script reads the CSV file into a pandas DataFrame, then creates a line plot with ‘Matrix Size’ on the x-axis and ‘Execution Time’ on the y-axis.

Please make sure to install the required libraries by running `pip install pandas matplotlib` in your terminal before running the script.

Let me know if you need further assistance! 😊

User

what row delimiter is used for the `pandas.read_csv()` function?

Bing

The `pandas.read_csv()` function in Python uses a newline (`\n`) as the row delimiter by default. However, you can specify a custom row delimiter using the `lineterminator` parameter. Please note that the `lineterminator` parameter does not allow a string longer than 1 character.

Here is an example of how to use the `lineterminator` parameter:


```
import pandas as pd

# Read a CSV file with a custom row delimiter
data = pd.read_csv('filename.csv', lineterminator=':')
```

In this example, the colon (:) is used as the row delimiter. Please replace 'filename.csv' with the path to your CSV file.

Let me know if you need further assistance! 😊

User

I don't see any output from the plotting function which reads the csv. Here is the format of the csv file Matrix Size,Execution Time 10,0, 20,0, 30,1, 40,4, 50,7, 60,13, 70,22, 80,33, 90,46, 100,63, 110,81, 120,92, 130,124, 140,149, 150,191, 160,229, 170,293, 180,351, 190,401, 200,556, 210,548, 220,706, 230,714, 240,819, 250,921, 260,1091, 270,1182, 280,1320, 290,1451, 300,1557, 310,1953, 320,2277, 330,3491, 340,2674, 350,3033, 360,3042, 370,3240, 380,3533, 390,3719, 400,4265,

Bing

The CSV file format you provided seems correct. If you're not seeing any output from the plotting function, it could be due to a few reasons:

1. **Library Import Issues:** Make sure you have the necessary libraries (pandas and matplotlib) installed and imported correctly in your Python environment.
2. **File Path:** Ensure that the CSV file is located in the correct path that your Python script is referring to. If the file is in the same directory as your Python script, you can just use the filename. Otherwise, you need to provide the full path to the file.
3. **Plot Display:** In some environments, plots created by matplotlib don't show up automatically. You might need to call `plt.show()` explicitly to display the plot.
4. **Data Reading:** Make sure that pandas is reading the CSV file correctly. You can check this by printing out the DataFrame after reading the CSV file. If the data is not being read correctly, you might need to adjust the parameters of the `pandas.read_csv()` function.

Here's the adjusted Python script with explicit plot display and data checking:

```
import pandas as pd
import matplotlib.pyplot as plt

# Read the CSV file
data = pd.read_csv('gemm_benchmark.csv')

# Check the data
print(data)

# Create the plot
plt.figure(figsize=(10, 6))
plt.plot(data['Matrix Size'], data['Execution Time'], marker='o')
plt.title('GEMM Execution Time')
plt.xlabel('Matrix Size')
plt.ylabel('Execution Time (ms)')
plt.grid(True)
plt.show()
```

If you're still having issues, could you please provide more details about the error or the issue you're facing? 😊

User

pandas seems to be interpreting the first value as a row label, here is the printed output Matrix Size Execution Time 10 0 NaN 20 0 NaN 30 1 NaN 40 4 NaN 50 7 NaN 60 13 NaN 70 22 NaN 80 33 NaN 90 46 NaN 100 63

NaN 110 81 NaN 120 92 NaN 130 124 NaN 140 149 NaN 150 191 NaN 160 229 NaN 170 293 NaN 180 351 NaN 190 401 NaN 200 556 NaN 210 548 NaN 220 706 NaN 230 714 NaN 240 819 NaN 250 921 NaN 260 1091 NaN 270 1182 NaN 280 1320 NaN 290 1451 NaN 300 1557 NaN 310 1953 NaN 320 2277 NaN 330 3491 NaN 340 2674 NaN 350 3033 NaN 360 3042 NaN 370 3240 NaN 380 3533 NaN 390 3719 NaN 400 4265 NaN

Bing

The issue you're experiencing is due to the default behavior of `pandas.read_csv()`. By default, it uses the first row of the CSV file as the header. If your CSV file doesn't have a header row, the first row of data will be used as the header, which seems to be the case here.

To fix this, you can use the `header=None` argument when calling `pandas.read_csv()`. This tells pandas that your CSV file does not have a header row, and the column names will be assigned automatically as integers.

Here's how you can modify your code:

```
import pandas as pd
import matplotlib.pyplot as plt

# Read the CSV file
data = pd.read_csv('gemm_benchmark.csv', header=None, names=['Matrix Size', 'Execution Time'])

# Check the data
print(data)

# Create the plot
plt.figure(figsize=(10, 6))
plt.plot(data['Matrix Size'], data['Execution Time'], marker='o')
plt.title('GEMM Execution Time')
plt.xlabel('Matrix Size')
plt.ylabel('Execution Time (ms)')
plt.grid(True)
plt.show()
```

In this modified script, I've added `header=None` to the `read_csv()` function and also provided a `names` parameter to manually set the column names. Now, pandas should correctly interpret your CSV file.

Let me know if you need further assistance! 😊

User

how do I profile in microseconds using Rust?

Bing

In Rust, you can use the `std::time::Instant` struct to measure the execution time of a code block. Here's an example:

```
use std::time::Instant;

fn main() {
    let start = Instant::now();

    // Code block to profile
    // ...

    let elapsed = start.elapsed();
    let elapsed_micros = elapsed.as_micros();

    println!("Elapsed: {} microseconds", elapsed_micros);
}
```

In this example, `start` is the time at which the measurement begins. After the code block to profile, `elapsed` is the `Duration` since `start`. The `as_micros` method converts this `Duration` into microseconds.

Please note that the actual precision depends on your system's hardware and operating system.

Let me know if you need further assistance! 😊

User

create a function in Rust to add two m-by-n matrices

Bing

Sure, here's a simple function in Rust that adds two m-by-n matrices. This function assumes that the two input matrices have the same dimensions.

```
fn add_matrices(a: &Vec<Vec<f64>>, b: &Vec<Vec<f64>>) -> Vec<Vec<f64>> {
    let m = a.len();
    let n = a[0].len();
    let mut result = vec![vec![0.0; n]; m];

    for i in 0..m {
        for j in 0..n {
            result[i][j] = a[i][j] + b[i][j];
        }
    }

    result
}
```

This function iterates over each element in the matrices a and b, adds the corresponding elements together, and stores the result in a new matrix. The new matrix is then returned as the result.

Please note that this code does not include any error checking. In a production setting, you would want to add checks to ensure that the input matrices are well-formed and have the same dimensions.

Let me know if you need further assistance! 😊

User

Create a Python function that reads in two .csvs and plots them on the same figure as scatter plots. The two .csv files are named “tiled_gemm_benchmark.csv” and “gemm_benchmark.csv”. They each contain two columns: “Matrix Size” and “Execution Time”

Bing

Sure, here's a Python function that does that. This function uses the pandas library to read the CSV files and matplotlib to create the scatter plots.

```
import pandas as pd
import matplotlib.pyplot as plt

def plot_csvs(file1, file2):
    # Read the CSV files
    data1 = pd.read_csv(file1)
    data2 = pd.read_csv(file2)

    # Create the scatter plots
    plt.figure(figsize=(10, 6))
    plt.scatter(data1['Matrix Size'], data1['Execution Time'], color='blue', label=file1)
    plt.scatter(data2['Matrix Size'], data2['Execution Time'], color='red', label=file2)
    plt.title('Execution Time Comparison')
    plt.xlabel('Matrix Size')
    plt.ylabel('Execution Time (ms)')
    plt.legend()
    plt.grid(True)
    plt.show()
```

```
# Call the function with your file names
plot_csvs('tiled_gemm_benchmark.csv', 'gemm_benchmark.csv')
```

This function reads the two CSV files into pandas DataFrames, then creates a scatter plot for each DataFrame on the same figure. The `Matrix Size` column is plotted on the x-axis and the `Execution Time` column is plotted on the y-axis. The `label` parameter in the scatter function is used to create a legend for the plot.

Please make sure to install the required libraries by running `pip install pandas matplotlib` in your terminal before running the script.

Let me know if you need further assistance! 😊