To measure the energy usage of a single operation within different GPUs we wanted to create sets of well-defined repeatable workloads that are flexible enough to run on different GPUs. To do so, we created two simple kernels or each operation. The first, called the control, had a known amount of the operations we were attempting to measure. The second, called the test, was identical to the control except for an increase in the number of those operations. While running the kernels we gather data on their energy usage, and can then compare that data to find the amount of energy a single operation uses.

It was important to keep the kernels simple for several reasons. Within our measurements, it was necessary to know exactly what was being run on the GPU. If our kernel was being optimized or executed differently than expected, our results may not be valid. A simpler kernel helps reduce the amount of factors to account for, so we attempted to reduce complexity as much as possible. Reducing the complexity also made it easier to generalize the kernels so that the same operations could be measured for different data types, reducing the number of necessary kernels. Fewer kernels allowed us to spend more time on each of them.

Code Sample?

Because the goal of each kernel is to execute a specific number of operations, we recognized that after a brief setup, we could iterate over the same operation a variable number of times. This allowed for the kernel to be adjusted to a length that gives solid measurements and stay simple. Each kernel followed this format of setup, iterate, and save result. It was often necessary to have dummy calculations within the control kernel so that adding more operations into the test did not accidentally add unwanted work. It is important to point out that the control and test kernels always had the same number of loop iterations, and the extra work came from slightly different workloads inside of the loops.

Only one kernel is run on the GPU at a time, and from run to run we need consistency. Because the kernels were being run on GPUs, it seems necessary to run the GPU at maximum workload. Ensuring that the kernel works as hard as possible for both the control and test helps account for non-kernel work. This non-kernel work may be from setting up and organizing the kernels. We want to try to keep it constant between runs, and one way to do so is to run the GPU at max workload. We commonly refer to this extra work as base power, as it is the power the GPU uses to perform all maximum capacity workloads. Keep in mind that this base power does not include any of the power from running any of the kernel’s instructions.

It does become important to find the value of this base power when the control and test kernels have different runtimes. The varying runtimes result from the necessity to have identical iteration counts in the kernel, but different amounts of operations. These differences may originate from issues such as memory lookup latency, where one kernel spends more time waiting for data than another. Because the GPU is always doing work in the background, we want to be able to remove this base power from the measurements before calculating the results.

To find base power we…

It is also important to ensure that the kernel is constantly running at max workload. Since we calculated base power at max workload, if there is a section of the run where the workload drops, then the base power value may be inaccurate. The easy way to give ensure the GPU is working at max is to give it far too many threads to run at once. SHOULD I EXPLAIN PARALLELISM AND THREADS? This can lead to problems, however, without being careful to make sure that all of the work ends at the same time. If the last threads slowly complete at the end of the run, then a graph of the kernels power usage throughout the run would show a downward slope as it has less work to do. To fix this, we implemented a few changes. Its important that threads don’t run too long, this allows the last threads to finish at around the same time. The second fix was to understand that threads are grouped into blocks, and blocks are executed independently of one-another. The GPU is split up into a handful of units, called SMs, which can each run one block at a time. We ensured that the number of blocks can be evenly divided by the number of SMs. This means that there will be no single block running threads at the end of the run with other empty SMs doing no work.

After making these changes, a graph of the kernel’s power usage during the run starts with a jump up to a height where it remains flat until the end.

Show graph?

To perform all of the necessary measurements for each kernel, we decided to use a built in library that queries instruments in the GPU. It is possible to use an external oscilloscope to measure the energy being taken from the PSU, as has been done BY OTHERS. Using built in libraries makes it much easier to run tests faster, on many GPUs, and without the need of extra equipment or expertise. The library also measures the power and temperature within the GPU die, which helps exclude other energy uses on a GPU board such as fans or DRAM. There is no need to change the measurement system between runs. Once we had a method to query power, temperature, and measure runtime, we did not need to vary it.

------------------------------------------------------------------------------------------------------------

MICROBENCHMARKING SUBSECTIONS:

------------------------------------------------------------------------------------------------------------

To prevent the complier from optimizing the kernel entirely, we pass a value into the kernel, and save the loops result to an array.

**Arithmetic operations approach**:

Arithmetic operations are the most straightforward operations to create kernels for. They only require a difference in the number of operations between the test and control. This is achieved by adding more operation lines to the tests loop. The general format of both kernels was to initialize one or two variables, and then use them within the math of the loop. LOAD AND STORE TO MEMORY TO PREVENT OPTIMIZATION

**Base Power approach**:

GPUs, an all processors, dissipate some energy regardless of the operations being performed, going towards maintaining state in memory cells, making scheduling decisions each cycle regardless of the operations being scheduled, and other necessary functions. However, unlike operations, which consume energy per unit work accomplished, these sources of energy drain are incurred per unit time the device is active. To account for energy spent per unit time not directly applied to the operations being performed, we must find a way to vary workload and execution time independently, to detect the impact of energy loads that are time-dependent rather than operation-dependent. We attempted two approaches, one artificially varying runtime for a known, fixed workload, and another where the runtime was kept constant while varying the workload performed within that time.

It is not possible to create a kernel without doing some work. Because of this, it is not possible to create a single kernel that measures base power, as it would be impossible to distinguish energy usage between base power and workload. It is, however, possible to run two kernels with different GPU utilization or workloads to allow the differences in workload energy to be known. Because GPUs are built for high levels of thread parallelism, they rely on the instruction-level parallelism between simultaneously running threads to fully utilize their execution units. If executing too few threads, the execution units will have more stalls than they would when fully occupied, reducing effective instruction throughput. Therefore, running the same kernel with different thread occupancy will cause the runs to have different runtimes for equivalent work. In general, thread occupancy is a mechanism allowing us to manipulate runtime independent of workload. By examining the measured differences between two runs with known relative workloads, it is possible to calculate the base power.

In a similar way, kernels run with different workloads of known relative size, including some variation in thread occupancy, provide enough information to decouple base power from operation energy. This must be done in a careful way, however, as not any change to the workload will cause the workload to change in a predictable way. The key to this approach is the known workload in both kernels. It’s important that the difference comes not from the kernel’s instructions, but from the amount of threads created.

Calculating base power using what we call “method 1” differs from the measurement of operations and memory lookup because it does not directly concern operation numbers. It does still contain two kernels, a control and a test, but the difference between them is how they are run on the GPU.

As mentioned, GPUs have a set of SMs, and each SM can run multiple blocks of threads at a time. WHAT IS UNDERUTILIZATION AND WHY DOES IT MATTER? Each SM also has its own memory, called shared memory, which all the threads within that SM can share data with. It is possible to request a specific amount of shared memory for each block. If a block requests all of the shared memory in the SM, then no other block can be run on the SM until it is finished. We take advantage of this functionality by limiting the number of blocks that can be run concurrently on each SM. The control may only be allowed to run one block per SM at a time, yet the test may be able to run three at a time.

Effectively, this changes the amount of parallelization that is allowed to happen, yet keeps the kernels workload identical. Because there is less parallelization in the control kernel, we expect it to be longer in runtime. The only difference between the two kernels is the runtime, and that difference in runtime contains some value of base energy. It then becomes simple to remove the known workload from the resulting measurements, and calculate the base power for those runs. As long as the control and test have identical workloads, the kernel’s specific calculations do not matter. We recycle an arithmetic kernel because they all have already been created and analyzed.

The second approach to base power measurement continues to use an identical kernel for the control and test runs. But unlike the first approach that manipulates the runtime, the second approach adds more threads to the test kernel’s workload. The kernel is first called with a handful of blocks such that the GPU is underutilized, and the average power is lower than it would be at full utilization. When the kernel is run again with more blocks, then the GPU at a higher utilization will have higher power consumption. We know the workloads of the two runs, so we can again calculate the GPUs base power by comparing the two runs.

ADD IDEAS: APPROACH 2 IS THE OPPOSITE OF AN OPERATION MEASUREMENT BC WE ASSUME WE KNOW THE OPERATION ENERGY AND WANT TO FIND THE BASE POWER ENERGY (OPERATION MEASUREMENT ASSUMES WE KNOW BP AND WANT TO FIND OPERATION ENERGY). ALSO TALK ABOUT WHAT WE WANT THE GRAPHS TO LOOK LIKE.

DISCUSS HOW WE HAVE 1 BP IDEA: KNOWN VARIATION BT TWO RUNS SO THAT WE CAN FIGURE OUT THE BP. TWO IDEAS FROM THAT ONE: KEEP WORKLOAD SAME AND VARY TIME BY UNDERUTILIZING, CHANGE WORKLOAD BT RUNS BUT KEEP UNDERUTILIZED. SPEND MOST TIME ON THE GENERAL IDEA OF FINDING BP FROM TWO DIFFERENT RUNS (VARYING UTILIZATION/INSUFFICIENT INSTRUCTION-LEVEL PARALELLISM). AT END HAVE SMALL PARAGRAPH EXPLAINING HOW WE SPECIFICALLY USED THE CORE IDEA BY RESTRICTING RESOURCES INCREASING NUM OF BLOCKS.

**Memory Test Approach:**

The goal of the memory lookups was to understand how much energy is consumed while loading from different memories on the GPU. Creating tests for these lookups is slightly more complicated than arithmetic tests due to the complexity of caches, and the difficulty of hitting the intended one. Intentionally hitting a specific cache can be difficult, because there may be another cache in between the registers and intended cache. If not careful, the kernel can hit the middle cache, and skew the results. Cache thrashing is a method to avoid this problem, and is explained SOMEWHERE ELSE. Global memory became the most difficult memory to hit because of this issue.

Because it is possible to specifically create variables inside of shared memory, it trivial to create kernels for shared memory. The L1 constant cache, however, required an intrinsic to force the compiler to load a given address from L1. This allowed

DON’T DISCUSS MID-LEVEL CACHES B/C THEY’RE NOT IN GPUS. MENTION HOW DATA NEEDS TO BE USED IN ARITHMETIC/WRITTEN TO MEMORY SO THE COMPILER DOESN’T OPTIMIZE OUT WORK/LOADING. EXPLAIN THE USE OF DUMMY MATH IN CONTROL.