GPU-Parallelizing Arbitrary Python Code By Running 1 Million Python Interpreters on a GPU

Josef Dean

Abstract

We introduce *gpusnek*, a fully functional Python interpreter ported to CUDA, enabling execution of arbitrary Python code directly on the GPU by running one whole interpreter on every CUDA core/thread. This is a tremendously bad idea, but for the duration of this paper we pretend that it is not, focusing instead on the surprisingly good job it does of combining the massively parallel compute capabilities of the GPU with the expressive power and ease-of-use of Python. Our implementation of *gpusnek* exposes the full CUDA memory hierarchy to Python in addition to key GPU-specific thread synchronization primitives, which allows serious GPU-native parallel algorithms to be written and run in pure Python. It also makes available many Python-native features that are traditionally difficult to achieve directly from a GPU thread, like reading from / writing to a filesystem. There are many possible applications of the *gpusnek* library (yes, we have made it available as a library). But in this work we choose to use it to build an interactive Python REPL backed by 1 Million interpreters running on a single GPU, unlocking instant and effortless million-way parallelism for your interactive Python terminal sessions.

`https://github.com/jndean/gpusnek`

If you just want to see the fun examples, skip to Section 2.

1. The Folly of Others

There are many serious projects that try to sell you "Python but on the GPU", and these tend to be doing one (or, often, both) of the following two things. Firstly, there are those projects that run Python code on the host CPU, and that Python code is used to plug together and schedule pre-written GPU kernels on the device (e.g., cuPy, PyTorch). In this case, the code running on the GPU is most definitely not Python; it is largely hand-optimized machine code provided by NVIDIA and other vendors to perform common vectorized linear algebra operations, and woe betide any workload that cannot be expressed in terms of the available operations! The second capability offered by existing projects is to use Python to write the GPU kernel itself, either in the traditional per-thread execution model (e.g., Numba, Mojo), or by offering vector constructs within Python's syntax that express block-level logic (Triton, cuTile, Pallas). In this case, too, it is most definitely not Python that is actually running on the GPU. Code written with these tools is transliterated, traced, JIT-compiled, and otherwise transmogrified into real device-specific machine code before execution. These magical code-gen pipelines offer you the regular kernel programming experience hidden behind the familiar-looking syntax of Python, and generally restrict the programmer to a very limited subset of Python's features and data types. Thus we find them to be lacking in the real power (and pain) that a true dynamically-typed and interpreted language should achieve when executing live on the GPU.

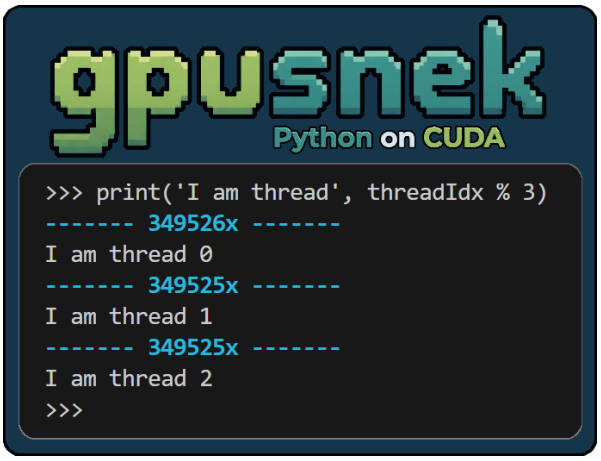

Figure 1. Running a print statement in an interactive Python REPL backed by 2^{20} Python interpreters. Output is deduplicated and printed with blue multiplicity banners, showing three unique responses each printed $\sim 350K$ times.

In contrast, *gpusnek* has the gall to put the whole Python interpreter directly on the GPU. Not just some for loops and some numerical data types, but the full language spec. Not just the bytecode-interpreting virtual machine, but also the lexer, the parser and the compiler. Not just a set of isolated single-threaded execution sandboxes, but a runtime with access to a real file system, the GPU's explicitly managed L1 cache, and inter-interpreter thread communication primitives that might let one get some real work done.

2. Examples

When you can execute arbitrary Python code on the GPU, what should you do with it? This section lists some examples, in order of increasing complexity and absurdity.

2.1. Hello World

Figure 2 shows a CUDA kernel that accepts a buffer of space in which each thread initializes a separate *gpusnek* interpreter. Then, this interpreter executes a Python source code string embedded into the kernel binary, printing "Hello World!".

```
__global__
void my_kernel(char* mem, int per_thread_mem) {
    char* my_mem = &mem[threadIdx.x * per_thread_mem];
    gpusnek_init(my_mem, per_thread_mem);

    gpusnek_do_str("print('Hello World!')\n")
}
```

Figure 2. A basic *gpusnek* kernel that prints a string directly to *stdout*.

Easy! Notice how nothing needs to be passed from the *gpusnek* initalization to subsequent *gpusnek* functions; once an interpreter has been set up by a thread, it continues to be accessible by the same thread forever after (assuming that the working memory is still alive), even across other kernel launches. For that reason, the rest of our examples omit the *gpusnek_init* call, assuming that a previous kernel launch has already set up the working memory ready for us to use. *gpusnek_do_str* utilizes the interpreter to execute a string of Python source code that is embedded into the binary. It is possible to instead embed and run pre-compiled MicroPython bytecode, but since we have also ported a GPU-native Python lexer and parser, we might as well use them to make the examples readable!

2.2. Co-operating Threads

Next we explore a few *gpusnek* features that make it possible to get real work done. Figure 3 shows a kernel for computing the mean of an array using a parallel reduction algorithm in $\log_2(N)$ time. It demonstrates the *gpusnek* API for making C++ data available inside the Python interpreter; the *gpusnek_new_int()* method just injects a new Python integer into the global namespace, while *gpusnek_bind_memory()* makes a region of memory available as a Python MemoryView object with configurable data types. Specifically in this case, we bind *data* to the data buffer in global HBM while *cache* points to a region of the L1 cache local to the thread block. This allows the Python code to perform a nice contiguous data load from HBM and then do the reduction entirely in the SRAM cache. The kernel also uses the *syncthreads()* 'function', which is one of our special additions to the MicroPython runtime (see Section 3.2.1). Rather than being a true function call, it gets mapped directly by the on-device compiler to a single bytecode instruction that invokes the *__synthreads()* CUDA intrinsic, acting as a thread barrier that syncs all CUDA threads within the same block.

```
__global__
void average_kernel(float *data) {
    int threadId = threadIdx.x;
    __shared__ float cache[NUM_THREADS];
    gpusnek_bind_memory("cache", cache, NUM_THREADS, 'f');
    gpusnek_bind_memory("data", data, NUM_THREADS, 'f');
    gpusnek_new_int("tid", threadId);
    gpusnek_new_int("N", NUM_THREADS);
    gpusnek_do_str(R"(

from math import log

cache[tid] = data[tid]
syncthreads()

steps = int(log(N, 2))
for i in range(steps):
    offset = N >> (i + 1)
    if tid < offset:
        cache[tid] += cache[tid + offset]
    syncthreads()

if tid == 0:
    data[0] = cache[0] / len(data)
)");
}
```

Figure 3. A kernel that takes the mean of an array using a parallel reduction sum, writing the answer into element 0. The kernel should be launched with one thread per element of data.

2.3. An In-Memory Virtual Filesystem

Example 2.2 showed the use of Python to implement a very GPU-shaped workload. Next, we divert to something very Python-native and entirely not GPU-shaped: reading and writing strings to files.

Figure 4 shows a simple example kernel that starts by binding a globally shared memory buffer to each interpreter. Next, it creates a block device which mimics storage hardware (such as flash storage) that accepts data read/write requests in blocks. In reality, this virtual block device is just a Python object that reads and writes the data to the shared memory buffer. Thread 0 then formats this 'device' as a *littlefs2* filesystem, and both threads mount it. From here on, they have access to a shared filesystem that lives entirely in the GPU VRAM, able to do all the usual things one expects to do with a filesystem. One thread could write Python code to a file and another could import and execute it as a module using a standard Python *import* statement! But in this example, thread 1 simply reads and prints the contents of a file written by thread 0.

```
__global__
void fs_kernel(char *fs_buf, int fs_size) {
    gpusnek_new_int("threadId", threadIdx.x);
    gpusnek_bind_memory("fs_buf", fs_buf, fs_size, 'B');
    gpusnek_do_str(R"(
import vfs

# Definition of VRAMBlockDevice omitted for brevity
bdev = VRAMBlockDevice(
    block_size=128,
    buffer=fs_buf,
)
if threadId == 0:
    vfs.VfsLfs2.mkfs(bdev)
    vfs.mount(bdev, '/vfs')
    message = 'THE FILESYSTEM LIVES ON THE GPU'
    with open('/vfs/myfile.txt', 'w') as f:
        f.write(message)

syncthreads()  # Is a barrier

if threadId == 1:
    vfs.mount(bdev, '/vfs')
    with open('/vfs/myfile.txt', 'r') as f:
        print(f.read())  # Prints the message
)");
}
```

Figure 4. Thread 0 initializes a shared VRAM buffer as a disk image for a virtual filesystem. Both threads mount this filesystem, and use it to pass a message between them.

2.4. One Million Python Interpreters (the Mega-REPL)

Next we demonstrate the scale of the parallelism available on a GPU. MicroPython provides an excellent premade REPL (Read-Eval-Print Loop) that lets a user write and execute Python code interactively in a terminal. Once the interpreter is initialized, the provided *pyexec_event_repl_process_char* function can be called to ingest one character at a time and, after each call, potentially print output from an execution (execution is likely only triggered when a return character is ingested). Having ported this MicroPython routine to CUDA as a part of *gpusnek*, we can have the host capture one key press at a time and send them via individual GPU kernel launches to be ingested by one (or many) GPU-side Python REPLs. The output from all interpreters is sent back to the host and de-duplicated, then printed to the terminal with multiplicity (see the turquoise banners in Figure 5). In this way, we create an interactive REPL where code is executed by as many interpreters as we can fit on the GPU, and all of the outputs are presented back to the user. By allocating just 12KB to each heap, 5KB to each stack, and 1KB to each *stdout* buffer, we can comfortably fit 2^{20} interpreters on an NVIDIA RTX 4070 Ti SUPER with 16GB of onboard VRAM, and thus unlock instant interactive million-way parallelism in Python on a consumer desktop platform. Figures 5 and 6 attempt to convey the experience of using it.

```
Starting gpusnek REPL with 1,048,576 snek...
>>> numThreads
1048576
>>> threadIdx % 3
------- 349526x -------
0
------- 349525x -------
1
------- 349525x -------
2
>>>
```

(a) Starting up an interactive terminal session with the Mega-REPL and printing some runtime-provided values. Outputs are deduplicated and printed with turquoise multiplicity banners (unless they are all identical, in which case the banner is omitted).

```
>>> if int(threadIdx ** 0.125) ** 8 == threadIdx:
...     print('8th power:', threadIdx)
...
------- 1x -------
8th power: 0
------- 1x -------
8th power: 1
------- 1x -------
8th power: 256
------- 1x -------
8th power: 390625
------- 1x -------
8th power: 65536
------- 1x -------
8th power: 6561
------- 1048570x -------
>>>
```

(b) Using the effortless parallelism of the Mega-REPL to parallelize the search for 8th powers. Each thread only checks one value (its own thread index). $1,048,570$ threads find and print nothing.

```
>>> with open('shakespeare.txt', 'r') as f:
...     f.seek(threadIdx * 6)
...     context = f.read(40)
...
>>> 'million' in context[:6 + len('million') - 1]
------- 1048556x -------
False
------- 20x -------
True
>>> if 'Snake' in context[:6 + len('Snake') - 1]:
...     print(context)
...
------- 1x -------
  Snakes, in my heart-blood warm'd, that
------- 1048575x -------
>>>
```

(c) The interpreters have a shared a filesystem loaded into VRAM, and we have preloaded the complete works of William Shakespeare as a ($\sim$ 5.2MB) text file. Each thread reads just a small chunk of the file (the full contents wouldn't fit in any one thread's heap!). Interactive parallelized searches over the full text are easily achieved by querying just the local chunk: we find 20 (case-sensitive) instances of the word 'million', and print the context around the sole instance of 'Snake'.

Figure 5. Example use cases for a REPL with 2^{20} threads.

3. How It's Made

gpusnek is a port of MicroPython [1], which is a wonderful open source implementation of Python 3 designed to run on microcontrollers in embedded applications.

In some ways, running code on CUDA is a lot like running code on a lot of tiny microcontrollers, packed side-by-side onto a single chip. Thus, many of the design goals driving MicroPython make it ideally suited to our use case:

- Small and lightweight - MicroPython claims to need only 256K ROM and 16K RAM to operate, which is going to be useful if we want 1 Million independent copies living in the HBM of one GPU, running on relatively low-powered "cores".
- Configurable - MicroPython defines a wealth of compile-time flags that allow us to disable problematic subsystems or toggle between implementations that suit our hardware better. These days you can call *printf* on a GPU, but you still cannot call *malloc*!
- Self-contained - not relying on many external libraries improves compatibility with exotic / adversarially-chosen hardware and tool chains.

3.1. Porting Challenges

MicroPython has a well defined process for porting to a new hardware platform. The core systems are defined centrally, and they interact with the hardware by calling out to an API that must be implemented by the port-writer to, e.g., read a character from the keyboard or print to the terminal. However, in spite of MicroPython's extensive configurability, our port to the GPU is sufficiently far from the beaten path that we must also make several invasive changes to the core Python implementation to get it working.

3.1.1. NVCC AND C++

To compile CUDA code to GPU machine code we must use NVIDIA's C++ compiler *nvcc*. The first problem is that MicroPython is written in C99, not C++. The C language is *very nearly* just a subset of C++; however, when one shoves 60K lines of C code through a C++ compiler, one soon finds examples of the edge cases where that is not true. For example, unlike C, C++ does not like the following patterns that are common throughout MicroPython: *goto* statements jumping over variable declarations; struct initialization using named fields; assigning void pointers to non-void pointers; taking a reference to a temporary inline variable; and so on and so on. Fortunately, the compiler is very good at classifying and pointing out these problems, so the process of converting all of the core MicroPython systems to valid C++ code was quite laborious, but not (to our knowledge) perilous.

```
>>> img_size = int(numThreads ** 0.5)
>>> X = threadIdx % img_size
>>> Y = threadIdx // img_size
>>> x0 = (X / img_size) * 0.13 - 0.62
>>> y0 = (Y / img_size) * 0.13 - 0.65
>>> x, y, i, max_iter = 0, 0, 0, 30
>>> while ((x*x + y*y) <= 4) & (i < max_iter):
...     x, y = x*x - y*y + x0, 2*x*y + y0
...     i += 1
>>> colour = i / max_iter
>>> outbuf[threadIdx * 3: threadIdx * 3 + 3] = [
...     256 * (1 - colour),   # B
...     256 * (colour ** 2),  # G
...     256 * colour,         # R
... ]
>>>
```

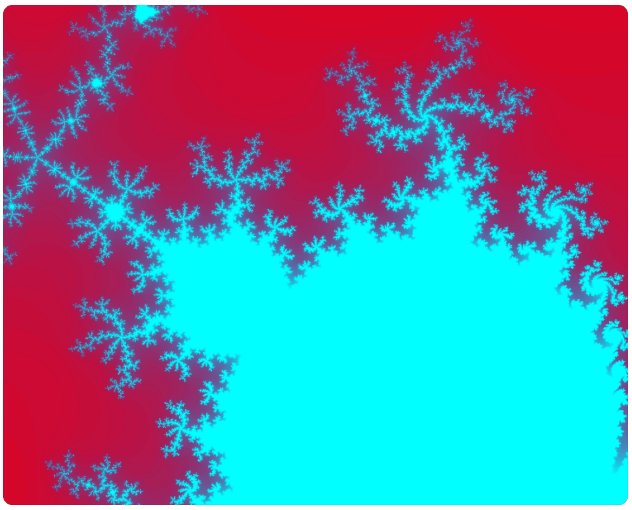

Figure 6. **(Top)** Each thread computes the colour of 1 pixel in the Mandelbrot Beetle fractal, and writes it into a buffer shared between all threads. **(Bottom)** Rendering that shared buffer as an RGB image texture. Using 2^{20} interpreters, each computing 1 pixel, quickly produces a 1024×1024 image (cropped here).

3.1.2. MUTABLE GLOBAL STATE

MicroPython is designed to be the only runtime on a single-threaded microcontroller, so it is able to store all of the interpreter state in static global mutable variables. This becomes a problem when running multiple copies of the interpreter on different threads on the same GPU - different threads executing different Python code will trample each other's global state. The solution is clear: modify MicroPython to instead have a global *array* of interpreter states (dynamically initialized), and any accesses to these globals go through a macro that uses the current thread's ID to index into the state array. This is why (as mentioned in Example 2.1) calls to the *gpusnek* API don't need to be given a reference to the thread-local interpreter - each thread always knows how to look up its local interpreter using just its own thread ID.

3.1.3. SETJMP & LONGJMP

MicroPython makes use of the C functions *setjmp* and *longjmp*. The former allows you to checkpoint the current state of the CPU's registers into memory, while the latter allows you to reinstate a previously made checkpoint. Since the CPU registers tend to include the Program Counter (PC) and Stack Pointer (SP) (which track the current position in the program and the stack), calling *longjmp* acts like a *goto* statement that can jump out of functions, straight back to the location in the code where *setjmp* was called. MicroPython uses this functionality to implement exception handling by setting checkpoints at locations that can catch Python exceptions. When an exception is raised in a function that may be deep in the call stack, the runtime can *longjmp* to the latest checkpoint to unwind large numbers of stack frames, without having to define a way to return from all the intermediate frames in an error state. This is all very well, but unfortunately CUDA does not provide an implementation of *setjmp* and *longjmp*. To some degree, this might be because conceptually it does not map very well to GPU hardware.

For example, in modern GPUs thread divergence is tracked explicitly by the program, which records bit masks at branch points in the code indicating which threads are still running in lock-step and could be about to diverge. These masks are moved around as data, and may end up being stored on the stack if functions are called. When execution reaches an opportunity to reconverge threads, e.g., the end of an *if* statement block, the bit mask is read and used to determine which threads should be brought back in sync. If one of our Python threads participates in a thread divergence event but then raises an exception when its warp neighbours do not, that thread may *longjump* over the thread reconvergence point in an intermediate call frame without participating, causing a deadlock when the other threads reach that point and wait for the masked sync.

This problem is challenging to foresee and engineer around, due to the somewhat arcane and also quite proprietary workings of the GPU architecture and its corresponding compiler. NVIDIA is not forthcoming with details about these systems, and it is only thanks to reverse-engineering efforts such as [3] that we can know enough to foresee the above issue. It is more than likely that other less foreseeable issues exist, either in software or in hardware, that would make resuming execution difficult to implement in "user-space", and a hack job implementation in assembly will just violate some opaque internal invariants and crash the GPU. Therefore, we leave the implementation of *setjmp* and *longjmp* on the GPU as possible future work.

Sections 3.1.4 and 3.1.5 touch more on the ways we work around their absence.

3.1.4. EXCEPTION HANDLING

Without *setjmp* and *longjmp*, MicroPython's exception handling mechanism cannot function. Currently in *gpusnek* we simply do not support exception handling: when an exception is raised, the exception message is printed directly to *stderr*, and the thread unceremoniously exits. As a workaround, we recommend writing Python code that does not raise any exceptions.

A partial solution would be possible if, as discussed above in Section 3.1.3, someone put the effort in to build an approximate *setjmp* implementation. There would still exist the problem that diverged threads should not be allowed to jump in case they jump over reconvergence points, but the implementation could use CUDA intrinsic *__activemask()* to check for divergence of the warp at run time and succeed in cases which are probably safe.

We speculate that a full solution would be to undertake a major rewrite of MicroPython that makes exception handling a first-class member of the function-calling API, i.e., have it be explicit in every function. This sounds like a huge amount of work that would add run-time memory and cycle overhead, plus it could introduce copious boilerplate to the code. It's not even clear if it would work. As such, we do not consider this a direction for future work.

3.1.5. DYNAMIC MEMORY ALLOCATION AND GARBAGE COLLECTION

CUDA offers no dynamic memory allocation capability on the device, i.e., you cannot call *malloc* from a GPU thread. Instead, memory allocation is handled by the host CPU code, and those buffers are treated as static when passed into GPU kernels. That's not going to work well for us if one of our GPU kernels contain Python interpreters that expect to be able to dynamically create large data structures on-the-fly, without ever talking to the host. Thankfully, MicroPython includes an implementation of a heap allocator, allowing us to pass a large static buffer in at interpreter initialization which will then be dynamically managed by the MicroPython runtime. As long as we give each thread its own pre-allocated heap space, this works well, with one small downside!

The MicroPython heap allocator uses a stack-sweeping garbage collection algorithm to reclaim unused memory. The garbage collector searches any locations that may contain root pointers to Python objects, and this includes the C call stack which may be holding references to local variables in a function call frame. In normal operation on a CPU, the exception handling mechanism (see Section 3.1.4) is also taking snapshots of the CPU's register state using C's *setjmp* function and saving them to the stack regularly, which means even object pointers not spilled from registers by the com-

piler will end up on the stack by garbage-collection time, ready to be swept. However, as we have already discussed, *gpusnek* is unable to take these register snapshots, and as such the callee-saved registers in each thread have a (rather slim) chance of being missed by the garbage collector. This is exacerbated by the tenacity with which the *nvcc* compiler attempts to avoid spilling registers to memory. We employ the following desperate mitigations to increase the chances that all live Python variables are referenced somewhere the garbage collector can find them:

- A separate compilation unit for the top-level garbage collection routine, with the intention of preventing the function call from being inlined by *nvcc*.
- Enabling MicroPython's builtin 'pystack' implementation, which stores the Python call frames in a separate explicitly managed stack buffer rather than directly in the C call stack. The garbage collector can search this stack buffer.

To date, this seems to have been sufficient to prevent disaster.

3.2. Adding New Functionality

Having overcome the porting challenges described above in Section 3.1, *gpusnek* could have been considered a technically functional but minimally capable toy. The next step, then, was to add new capabilities that allow the user to better exploit the unique offerings of Python running natively on a GPU.

3.2.1. A GPU THREAD BARRIER

CUDA offers the programmer the *__syncthreads()* intrinsic: a hardware-enforced barrier where each CUDA thread will wait until all threads in the same thread block have arrived and synced. We make this functionality directly available in *gpusnek* via the *syncthreads()* call, which appears to be a globally-available function but is in fact a custom lexer and parser rule that emits a new dedicated instruction in the compiled bytecode. The example in Section 2.2 demonstrates using this functionality to enable threads to work together on shared data.

3.2.2. STDIN AND STDOUT

gpusnek offers multiple IO modes that can either cyclically buffer input/output in dedicated per-thread memory regions, or use CUDA's included *printf* functionality to print directly from each kernel to the host's terminal. This makes it more flexible and useful as a component to be incorporated into other projects.

3.2.3. AN IN-MEMORY VIRTUAL FILESYSTEM

We have included in *gpusnek* a port of MicroPython's virtual filesystem implementation, as well as a port of the specific filesystem *littlefs2* [2]. *littlefs2* is designed to run on microcontrollers, and therefore boasts the following qualities that make it perfect for running on a low-resource GPU thread:

- Strictly bounded RAM usage as the content of the filesystem grows.
- No dynamic memory allocations (can be provided with statically allocated buffers for working memory).
- No unbounded recursion, making maximum stack sizes computable.

With these implementations successfully ported to the GPU, we are able to format an in-VRAM buffer as a *littlefs2* disk image and mount it, all from within Python. If multiple threads mount the same buffer, they can access a shared filesystem from separate interpreters concurrently, even supporting simultaneous reads. Simultaneous writes are not supported, though we implement no mechanism to prevent them; race conditions in user code may silently corrupt not only the contents of the written file, but also the entire disk image :). Since the filesystem lives in global VRAM it can persist between kernel launches for the duration of the program. To persist it longer, we can also set up a mechanism to copy the disk image back to the host at shutdown, save it to the host disk, then reload it on subsequent launches. The host could even choose to mount the disk image itself, allowing for a filesystem shared between both the host CPU and the GPU (assuming an application-appropriate synchronization mechanism is put in place to replicate the disk image between host RAM and GPU VRAM).

The examples in Sections 2.3 and 2.4 demonstrate the filesystem in action.

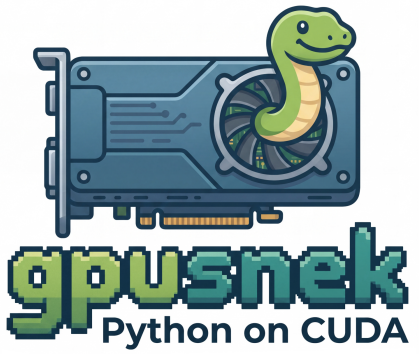

Figure 7. Putting the *gpusnek* logo here fills an inconvenient gap in the document, pushing the heading for the next section onto the next page. It also adds visual interest, when we have otherwise had 2 pages of solid text.

Device	Runtime	Logical Cores	Duration (milliseconds)	Duration (relative)
GPU (V100)	CUDA C	5120	0.318	1×
	GPUSNEK	5120	2,701	8,489×
	GPUSNEK (PRECOMPILED)	5120	2,474	7,776×
CPU (5800X3D)	CPython	1	3,440	10,814×
	CPython (NO GIL)	16	423	1,330×
	C	1	96.8	304×
	C (OpenMP)	16	9.44	30×

Table 1. Performance of various hardware / runtime configurations on a fixed-size embarrassingly parallel summation task. To create a sensible CPU Python baseline, we add a for loop over the thread dimension to serialize the work that is parallelized on the GPU. We do not use libraries like numpy, because we are trying to compare the overheads of using native data types and for loops. On all hardware platforms, we endeavor to measure performance on *cold* caches.

4. Performance

In general, we find that *gpusnek* offers great **qualitative** value, and prefer not to dwell on its **quantitative** value. Therefore, while at first glance the results in this section may appear to dissuade the reader from ever using *gpusnek* for any real-world task, we suggest that it is in fact more specifically dissuading the user from ever **profiling** their usage of *gpusnek* in such tasks. With this in mind, Figure 8 shows how we define kernels in pure CUDA C versus *gpusnek* to sum the 256 rows of a 256×2^{18} matrix in FP32.

4.1. Headline Kernel Performance

Table 1 shows that during profiling, the pure CUDA C kernel is overall 8489 times faster than the *gpusnek* kernel, with the former completing the task in a few hundred microseconds and the latter being better measured in full seconds. Compared to just completing the same task with a single-threaded Python script running on the CPU, we find that porting to *gpusnek* on a V100 achieves a modest 40% speedup, and also demonstrate that it would be dramatically more effective to instead parallelize over multiple CPU cores with sensible multi-threading.

We would certainly expect an interpreted language to be slower than C, but would we expect Python on CUDA to be this much slower than the CUDA C kernel? For reference, we also include in Table 1 the same task executed on the CPU in both C and CPython, observing just a 35 to $45\times$ slowdown between the two. So why is doing the same on a GPU two orders of magnitude worse? To investigate, we first break down where each kernel spends its time.

Table 2 shows that the basic *gpusnek* kernel spends a total of 236ms (across all 2^{18} thread launches) initializing the on-device Python interpreters and compiling the Python source code string into executable bytecode. It does not make much sense that all 2^{18} interpreters parse and compile 2^{18} copies of the same Python source code into identical bytecode, so the 'precompiled' version of the *gpusnek* kernel shown in Figure 8 can instead use a preinitialized interpreter to ex-

```
#define NUM_ROWS (256)
#define NUM_COLS (262144)
#define SIZE (NUM_ROWS * NUM_COLS)

__global__ void cuda_sum(float *data) {
    int tid = threadIdx.x;
    float total = 0.0f;
    for (int row = 0; row < NUM_ROWS; row++)
        total += data[row * NUM_COLS + tid];
    data[tid] = total;
}

__global__ void gpusnek_sum(float *data, ...) {
    gpusnek_init(...);
    gpusnek_bind_memory("data", data, SIZE, 'f');
    gpusnek_new_int("tid", threadIdx.x);
    gpusnek_new_int("NUM_ROWS", NUM_ROWS);
    gpusnek_new_int("NUM_COLS", NUM_COLS);
    mp_obj_t compiled_code = gpusnek_compile(R"(
total = 0.0
for row in range(NUM_ROWS):
    total += data[row * NUM_COLS + tid]
data[tid] = total
)");

    gpusnek_do_bytecode(compiled_code);
}

__global__ void gpusnek_precompiled_sum(
    float *data, mp_obj_t precompiled_code
) {
    gpusnek_bind_memory("data", data, SIZE, 'f');
    gpusnek_do_bytecode(precompiled_code);
}
```

Figure 8. Pure CUDA C versus *gpusnek* kernels for summing rows of a matrix. Each thread sums the elements of one column, so we launch 2^{18} threads. The result is written to row 0. A precompiled version of the *gpusnek* kernel avoids initializing the Python interpreter or parsing the Python source code on every kernel launch.

TASK	DURATION (μs)	
	CUDA C	GPUSNEK
KERNEL OVERHEADS	2.71	4.65
INITIALIZE INTERPRETERS	-	11,541
PARSE & COMPILE SRC	-	225,886
EXECUTE LOAD-SUM LOOP	313	2,455,729
WRITE RESULT	2.55	7,799
TOTAL	318	2,700,960

Table 2. Breakdown of time spent in each GPU kernel.

ecute bytecode that was compiled ahead of time and thus completely remove these costs. However, these superfluous Python setup costs represent less than 10% of the *gpusnek* kernel run time, so removing them does not solve our performance gap.

4.2. Confusing Performance Metrics

Next, we explore the performance metrics that are available through NVIDIA's frankly phenomenal suite of profiling tools. Firstly, we present Table 3, containing completely true but potentially confusing metrics that raise more questions than they answer. After some discussion, we subsequently present Table 4, which contains a much more telling selection of metrics that provide the answers to these questions.

Metric	CUDA C	gpusnek
Memory Bandwidth Utilization	94.5%	50.3%
Compute Utilization	6.35%	2.28%
Warp cycles per instruction issue	237	108
Avg. participating threads per warp	99.2%	93.1%

Table 3. Kernel profiling metrics that absolutely do not explain why *gpusnek* takes 8000 times longer than CUDA C to perform the same task.

We see from Table 3 that the CUDA kernel is almost maxing out memory bandwidth and under-utilizing compute. Being bandwidth-bound makes sense for this workload; the sum operation is mainly just loading row after row of data, and performing a single cheap addition operation each time. In contrast, the *gpusnek* kernel is far from bandwidth-bound, using only 50% of the memory bandwidth to VRAM. However, confusingly, it also uses even less compute than the CUDA C kernel. What is it doing in the nearly 50% of the time when it is waiting on neither memory nor compute resource? Next, looking at the average number of cycles that elapse between each instruction being issued for each warp, we see that despite the low compute utilization, each *gpusnek* kernel is in fact completing the execution of issued instructions more than twice as fast as the C kernel. Finally, in case we were suspecting divergence of threads within a warp as a cause for performance degradation, we observe that the thread participation per warp is high for both kernels. This is as expected; every thread is executing an identical sequence of operations, so even with the much more complex task of running the Python interpreter there is no reason for threads within a warp to diverge. In conclusion, nothing in Table 3 explains why the Python slowdown factor is over $8000\times$ on a GPU compared to $40\times$ on a CPU.

4.3. Enlightening Performance Metrics

Metric	CUDA C	gpusnek
Registers per thread	32	255
Occupancy (warps per SM)	53.5	7.97
VRAM Sector Loads	8,388,608	26,942,463,116
Sector Load Efficiency	100%	14.9%
L1 Cache Hit Rate	0%	51.6%

Table 4. Kernel profiling metrics that actually shed some light on why *gpusnek* is so much slower than should be expected when moving to an interpreted language.

Table 4 provides the answers to the questions raised in this section. We see that the C kernel uses 32 registers per thread, whilst *gpusnek* uses 255 to run its much more complex code. Physical registers are a finite resource, with each Streaming Multiprocessor (SM) on a V100 having 65,536 physical registers available. Dividing the registers available on the hardware by the per-kernel requirement tells us that each SM can only have 256 *gpusnek* threads in residence at once, arranged as 8 warps of 32 threads. And indeed, looking at the occupancy metrics we see *gpusnek* manages just 8 warps per SM, far lower than the 53.5 warps averaged by the C kernel. The cost of this low occupancy is significant. A GPU relies on having many thread warps in residence on the same SM to cover high memory load latency; while one warp is stalled waiting for data to be fetched, another resident warp can be using the SM compute resources to execute instructions (e.g., initiating another data fetch). Therefore, even though Table 3 showed us the C kernel took 237 cycles per instruction per warp, by keeping 53.5 warps in residence the SM averaged just $237/53.5 = 4.43$ cycles per instruction. Meanwhile, *gpusnek* is waiting $108/7.97 = 13.6$ cycles per issued instruction. This increased idle time partially explains why both bandwidth and compute utilization are low; when all 8 warps are waiting for data fetches to return, the SM goes completely idle.

Next we look at the number of 'VRAM Sector Loads' performed by each kernel. A sector is a line of 32 bytes that gets fetched from VRAM in a single request, and is the smallest granularity operable by the global memory subsystem. Since our test data is a $2^8 \times 2^{18}$ FP32 matrix, we would expect to be able to load it with $8,388,608$ sector loads, and this is precisely the number reported by the C kernel. Meanwhile, the *gpusnek* kernel issues $3212\times$ as many loads to perform the same task! This is likely the main cause of our diminished Python performance, and is attributable to a combination of the following issues:

- Firstly, there is all the state maintained by the internals of the Python interpreter. Constantly updating these variables as the VM executes the Python bytecode will incur some number of extra reads and writes for every operation performed. However, as we have previously observed, CPU implementations only incur around a $40\times$ duration penalty for this overhead, so this must be compounding with other GPU-specific effects.

- Secondly, CUDA sets the maximum number of registers available to a single thread at 255. Since the *gpusnek* kernel has hit this cap, it is very likely spilling values that would prefer to be registers into local memory, and thus further inflating the number of VRAM load/store operations required to execute each bytecode instruction.

- Thirdly, we consider memory locality. The C kernel has been written so that each warp of 32 adjacent threads is loading a contiguous run of adjacent values in memory in the same cycle, maximizing the efficiency of loading data in 32-byte sectors. This is confirmed by the 'Sector Load Efficiency' being 100%. The *gpusnek* kernel performs the same efficient loads when fetching the data to be summed, but the memory traffic is dominated by loading internal interpreter state. Most of these loads probably look like reading one 4-byte value out of some MicroPython struct. Adjacent threads all perform the same operation on the same cycle, so they all read the same offset into the same type of struct, but each struct lives in the private heap/stack of a different interpreter and these are located far apart in memory. As such, these loads are perfectly non-contiguous, fetching an entire 32-byte sector only to use 4 bytes of it. This would give a $4/32 = 12.5\%$ 'Sector Load Efficiency', which we can see is about what we measure in practice. This applies another factor of 8 increase to the number of extra load operations required to perform the same work.

The final metric in Table 4 suggests one amusing upside to the terrible non-contiguity of the *gpusnek* load operations. Since the C kernel is perfectly efficient at loading data there is no need to ever load from the same sector twice, and it therefore has zero L1 cache hits. Meanwhile, the internal workings of the Python interpreter involve reading and writing to the same state variables multiple times, plus the state within each interpreter is closely collocated. Thus, we find that 51.6% of *gpusnek* load operations successfully hit the hardware-managed L1 cache rather than going to VRAM. This is the reason for the lower 'Warp cycle per instruction issue' value in Table 3; *gpusnek* threads spends less than half as much time waiting between instructions because half the time they can load from the L1 cache!

5. Conclusion

In this work we introduced *gpusnek*, a functional port of the MicroPython lexer, parser, compiler and interpreter to CUDA, enabling GPU-native execution of familiar Python code at GPU scales of parallelism. We implement not only those components that appear to be a good fit for both Python and CUDA but also those that are normally the domain of just one of the other, such as thread-synchronization primitives and filesystem accesses. With these tools, we bring all the familiar upsides and downsides of Python to the GPU, and even discover some new ones! In particular, we demonstrate that it is possible to interactively drive 1 Million Python interpreters at the command line on consumer hardware, and use them to parallelize non-trivial tasks. Though the performance section makes it quite clear that no serious practitioner should employ *gpusnek* for any reason, the full source code is made available in the below repository.

https://github.com/jndean/gpusnek

References

[1] Micropython. www.micropython.org. Accessed: 31-03-2026.

[2] littlefs2. https://github.com/littlefs-project/littlefs. Accessed: 31-03-2026.

[3] Shoushtary, M. A., Murgadas, J. T., and Gonzalez, A. Control flow management in modern gpus, 2024. URL https://arxiv.org/abs/2407.02944.