Triton

The challenges of GPU programming

The architecture of modern GPUs can be roughly divided into three major components—DRAM (Dynamic Random Access Memory), SRAM (Static Random Access Memory) and ALUs (Arithmetic Logic Units)—each of which must be considered when optimizing CUDA code:

- Memory transfers from DRAM must be coalesced into large transactions to leverage the large bus width of modern memory interfaces.
- Data must be manually stashed to SRAM prior to being re-used, and managed so as to minimize shared memory bank conflicts upon retrieval.
- Computations must be partitioned and scheduled carefully, both across and within Streaming Multiprocessors (SMs), so as to promote instruction/thread-level parallelism and leverage special-purpose ALUs (e.g., tensor cores).

Overview of Triton

Triton is a powerful open-source framework for simplifying GPU programming, especially for neural networks. By abstracting low-level CUDA intricacies, Triton enables researchers to write high-performance GPU code in a Python-like syntax without needing deep CUDA expertise.

- High Efficiency: Triton kernels achieve performance comparable to or better than hand-optimized CUDA code, e.g., cuBLAS-level FP16 matrix multiplication in under 25 lines.
- Python-like Syntax: Makes GPU programming accessible with familiar and simple constructs.

Overview of Triton

Feature	Description	CUDA	TRITON
Memory Coalescing	Ensures that memory accesses by threads are grouped into fewer, more efficient transactions to optimize global memory bandwidth. Requires explicit memory alignment and access patterns in code.	Manual	Automatic
Shared Memory Management	Refers to the explicit allocation and management of on-chip shared memory to speed up frequent memory accesses by threads in a block. In	Manual	Automatic
Scheduling (Within SMs)	Handles the execution order of threads within a Streaming Multiprocessor (SM).	Manual	Automatic
Scheduling (Across SMs)	Refers to how workloads are distributed across multiple Streaming Multiprocessors (SMs) on the GPU.	Manual	Manual

Basic Syntax of Triton

```python
import triton
import triton.language as tl
@triton.jit
def add_kernel(
    x_ptr,  # *Pointer* to first input vector.
    y_ptr,  # *Pointer* to second input vector.
    output_ptr,  # *Pointer* to output vector.
    n_elements,  # Size of the vector.
    BLOCK_SIZE: tl.constexpr,  # Number of elements each program should process.
):
    # There are multiple 'programs' processing different data. We identify which program we are here:
    pid = tl.program_id(axis=0)  # We use a 1D launch grid so axis is 0.
    # This program will process inputs that are offset from the initial data. For instance, if you had a vector of
    # length 256 and block_size of 64, the programs would each access the elements
    # [0:64, 64:128, 128:192, 192:256]. Note that offsets is a list of pointers:
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    # Create a mask to guard memory operations against out-of-bounds accesses.
    mask = offsets < n_elements
    # Load x and y from DRAM, masking out any extra elements in case the input is not a multiple of the block size.
    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)
    output = x + y
    # Write x + y back to DRAM.
    tl.store(output_ptr + offsets, output, mask=mask)
```

Triton Semantics - Type Promotion

1. **Kind Hierarchy**: `{bool} < {integer} < {floating-point}`
   Example: `(int32, bfloat16) -> bfloat16`
2. **Width**: For the same kind, the higher width is chosen.
   Example: `(float32, float16) -> float32`
3. **Prefer float16**: If the same width and signedness but different types, promote to `float16`.
   Example: `(float16, bfloat16) -> float16`
4. **Prefer unsigned**: For same width but different signedness, promote to unsigned.
   Example: `(int32, uint32) -> uint32`

Triton Semantics - Broadcasting

Operations on tensors of different shapes expand them to compatible shapes without copying data:

- Shorter shapes are left-padded with ones.
  Example: `((3, 4), (5, 3, 4)) -> ((1, 3, 4), (5, 3, 4))`
- Dimensions are compatible if equal or one is 1.
  Example: `((1, 3, 4), (5, 3, 4)) -> ((5, 3, 4), (5, 3, 4))`

Triton Semantics - Rounding

- **Integer division and modulus**: Follow C semantics (rounding towards zero), unlike Python's rounding towards minus infinity.
  Example: `-7 // 3 = -2` (Triton) vs. `-7 // 3 = -3` (Python).
- **Scalar operations**: Use Python semantics for integer division and modulus when all inputs are scalars.

Basic Syntax of Triton

```python
import triton
import triton.language as tl
@triton.jit
def add_kernel(
    x_ptr,  # *Pointer* to first input vector.
    y_ptr,  # *Pointer* to second input vector.
    output_ptr,  # *Pointer* to output vector.
    n_elements,  # Size of the vector.
    BLOCK_SIZE: tl.constexpr,  # Number of elements each program should process.
):
  # There are multiple 'programs' processing different data. We identify which program we are here:
  pid = tl.program_id(axis=0)  # We use a 1D launch grid so axis is 0.
  # This program will process inputs that are offset from the initial data. For instance, if you had a vector of
  # length 256 and block_size of 64, the programs would each access the elements
  # [0:64, 64:128, 128:192, 192:256]. Note that offsets is a list of pointers:
  block_start = pid * BLOCK_SIZE
  offsets = block_start + tl.arange(0, BLOCK_SIZE)
  # Create a mask to guard memory operations against out-of-bounds accesses.
  mask = offsets < n_elements
  # Load x and y from DRAM, masking out any extra elements in case the input is not a multiple of the block size.
  x = tl.load(x_ptr + offsets, mask=mask)
  y = tl.load(y_ptr + offsets, mask=mask)
  output = x + y
  # Write x + y back to DRAM.
  tl.store(output_ptr + offsets, output, mask=mask)
```

Basic Syntax of Triton - **program_id** and **num_programs**

In Triton, `pid` refers to the program ID, which represents the ID of the current program instance along a specified axis in a 3D launch grid. The `program_id` function in Triton is used to identify which program is currently executing and helps distribute work across threads or blocks in parallel computations.

axis: The dimension of the grid along which the program ID is calculated. It can be 0, 1, or 2.

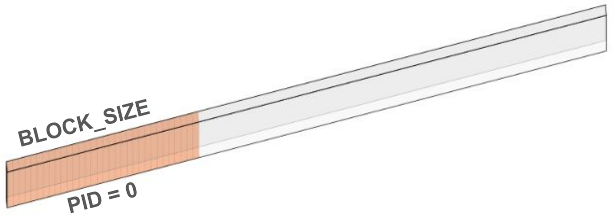
```
pid = tl.program_id(axis=0)
```

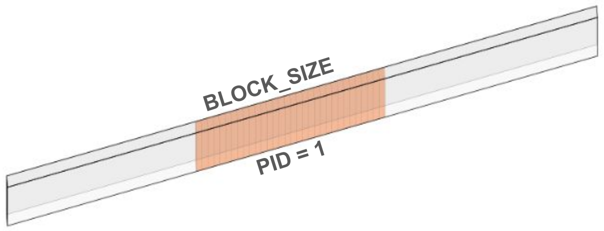
```
@triton.jit
def demo(x_ptr):
    print(tl.num_programs(0), tl.num_programs(1), tl.num_programs(2))

x = torch.ones(2, 4, 4)
triton_viz.trace(demo)[(2, 2, 1)](x)
```

```
[2] [2] [1]
[2] [2] [1]
[2] [2] [1]
[2] [2] [1]
```

Basic Syntax of Triton - BLOCK_SIZE: **tl.constexpr**

```
block_start = pid * BLOCK_SIZE
offsets = block_start + tl.arange(0, BLOCK_SIZE)
mask = offsets < n_elements
```





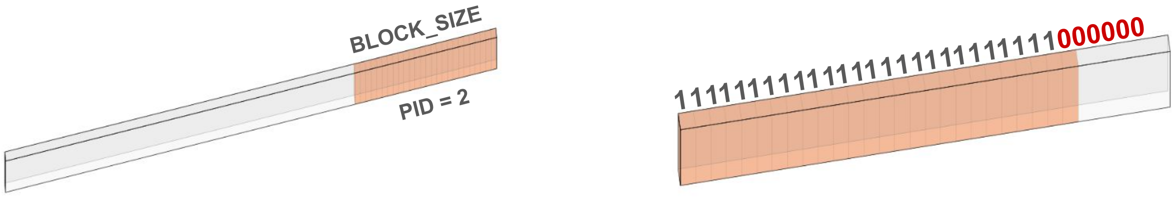
Basic Syntax of Triton - mask

The mask is used to safeguard memory operations from out-of-bounds access.

In this case:

- **mask = offsets < n_elements** determines which indices are within the array size n_elements.
- During data loading and storing (**tl.load**, **tl.store**), the mask prevents access to non-existent elements, avoiding errors.

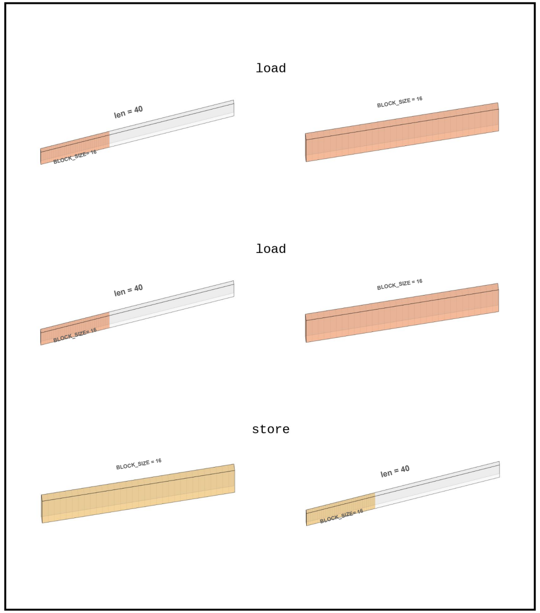
`load`



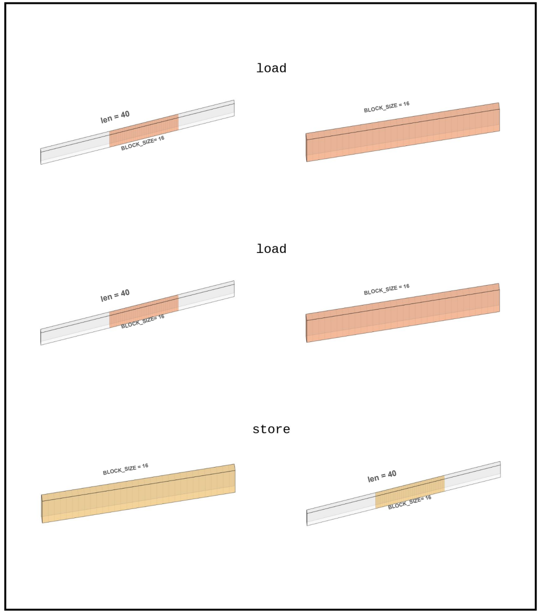
Basic Syntax of Triton - **tl.load, tl.store**

```
x = tl.load(x_ptr + offsets, mask=mask)
y = tl.load(y_ptr + offsets, mask=mask)

output = x + y

tl.store(output_ptr + offsets, output, mask=mask)
```

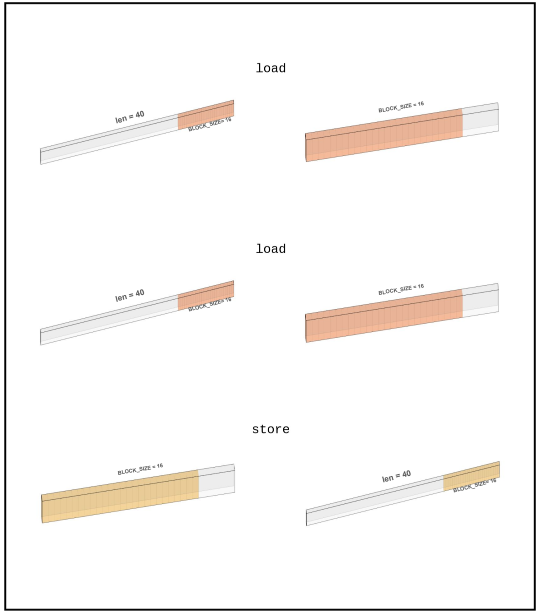
Basic Syntax of Triton - **tl.load, tl.store**



PID = 0



PID = 1



PID = 2

Basic Syntax of Triton

```python
import triton
import triton.language as tl
@triton.jit
def add_kernel(
    x_ptr,
    y_ptr,
    output_ptr,
    n_elements,
    BLOCK_SIZE: tl.constexpr,
):
    pid = tl.program_id(axis=0)

    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n_elements

    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)

    output = x + y

    tl.store(output_ptr + offsets, output, mask=mask)
```

```python
print(x_ptr.dtype)              # pointer<fp32>
print(y_ptr.dtype)              # pointer<fp32>
print(output_ptr.dtype)         # pointer<fp32>
print(type(n_elements))         # <class 'int'>
print(type(BLOCK_SIZE))         # <class 'int'>

print(pid.dtype)                # int32

print(block_start.dtype)        # int32
print(offsets.dtype)            # int32
print(mask.dtype)               # int1

print(x.dtype)                  #fp32
print(output.dtype)             # fp32
print((z_ptr + offsets).dtype)  # pointer<fp32>
```

Memory allocation and pointer arithmetic in C - number.

```
#include <stdio.h>
int main(void)
{
    int n = 10;
    int *ptr = &n;
    printf("address=%p \t value=%d \n", (void*)ptr, *ptr); //address=0060FEA8   value=10
    ptr++;
    printf("address=%p \t value=%d \n", (void*)ptr, *ptr); //address=0060FEAC   value=6356652
    ptr--;
    printf("address=%p \t value=%d \n", (void*)ptr, *ptr); //address=0060FEA8   value=10
    return 0;
}
```

A number in C is represented as a pointer of a specific size.
In the case of int32, it is 4 bytes.

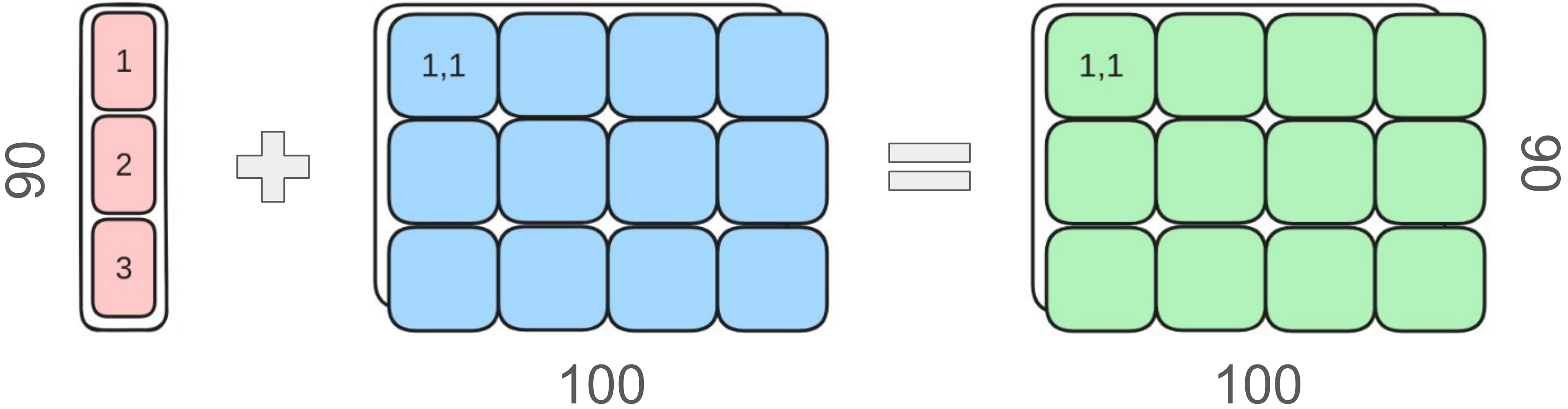
Memory allocation and pointer arithmetic in C - array.

```
#include <stdio.h>
int main(void)
{
    int numbers[] = {11, 12, 13, 14};
    for(size_t i=0; i< sizeof(numbers) / sizeof(numbers[0]); ++i){
        printf("numbers[%d]: %p; value: %d\n", i, numbers+i, *(numbers+i));
    }
    return 0;
}
```

```
numbers[0]: 0x7fffb4d10140; value: 11
numbers[1]: 0x7fffb4d10144; value: 12
numbers[2]: 0x7fffb4d10148; value: 13
numbers[3]: 0x7fffb4d1014c; value: 14
```

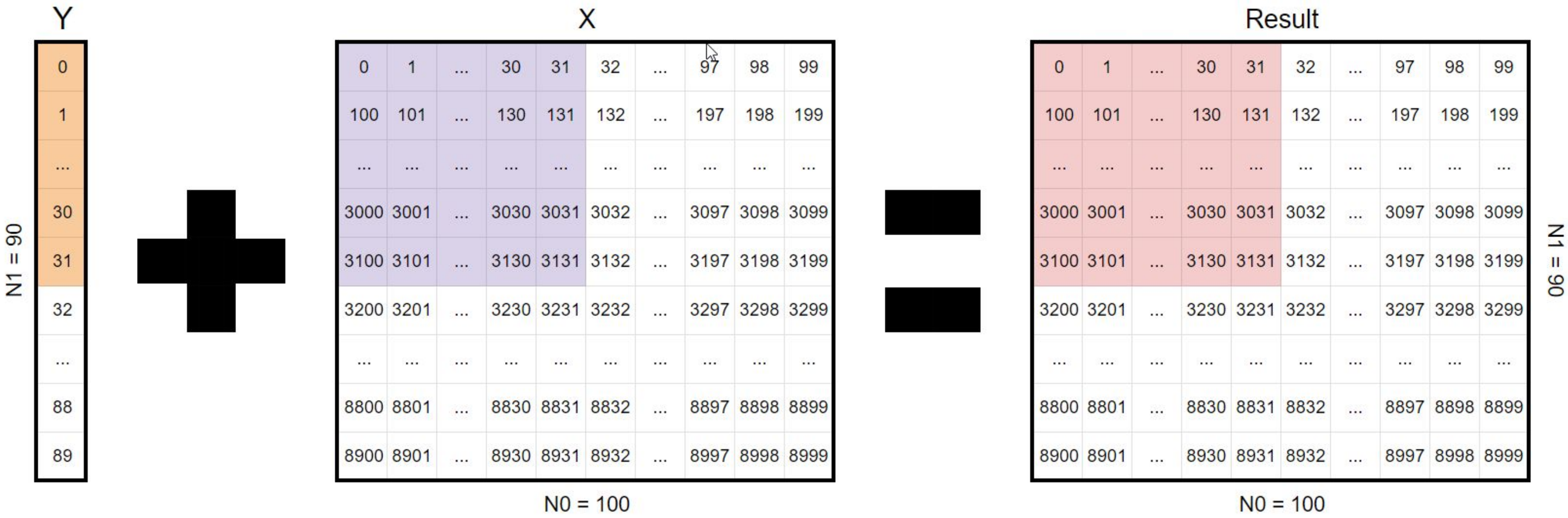
An array in C represents a contiguous block of memory where its elements are stored. The size of the array is determined by **n*sizeof(data_type)**. Pointer arithmetic applies to this structure.
In other words, an **array in C is a pointer to the first element of the array with pointer arithmetic**.

Triton pointer arithmetic - 2D arrays



Triton pointer arithmetic - 2D arrays

B0 = B1 = 32
N0 = 100 N1 = 90



Triton pointer arithmetic - 2D arrays

```python
@triton.jit
def mul_relu_block_back_kernel(x_ptr, y_ptr, dz_ptr, dx_ptr, N0, N1, B0: tl.constexpr, B1: tl.constexpr):
    pid_x = tl.program_id(0) # program ID along the X-axis (axis 0)
    pid_y = tl.program_id(1) # program ID along the y-axis (axis 1)

    offset_x = tl.arange(0, B0) + pid_x * B0 # offset_x = [0, 1, 2, ..., 31]
    offset_y = tl.arange(0, B1) + pid_y * B1 # offset_y = [0, 1, 2, ..., 31]

    # 2D array [[0, 1, 2, ..., 31], [100, 101, 102, ..., 131], ..., [3100, 3101, ..., 3131]]
    offset = offset_x[None, :] + offset_y[:, None] * N0

    mask_x = offset_x < N0 # N0 = 100; array of 32 bool values
    mask_y = offset_y < N1 # N1 = 90; array of 32 bool values

    mask = mask_x[None, :] & mask_y[:, None] # array of 32x32 bool values

    y = tl.load(y_ptr + offset_y, mask_y, 0) # 1-d pointer array with length = 32
    x = tl.load(x_ptr + offset, mask, 0) # 1-d pointer array with length = 32x32

    res = x + y[:, None] # 1-d pointer array with length = 32x32

    tl.store(dz_ptr + offset, res, mask)
```

Calling a Triton Kernel from Python

```python
import triton
import triton.language as tl
@triton.jit
def add_kernel(
    x_ptr,
    y_ptr,
    output_ptr,
    n_elements,
    BLOCK_SIZE: tl.constexpr,
):
    pid = tl.program_id(axis=0)

    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n_elements

    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)

    output = x + y

    tl.store(output_ptr + offsets, output, mask=mask)
```

```python
def add(x: torch.Tensor, y: torch.Tensor):
    assert x.is_cuda and y.is_cuda
    output = torch.empty_like(x)
    n_elements = x.numel()
    grid = lambda meta: (triton.cdiv(n_elements, meta['BLOCK_SIZE']),)
    add_kernel[grid](x, y, output, n_elements, BLOCK_SIZE=1024)
    return output
```

```python
if __name__ == "__main__":
    x = torch.tensor([1, 2, 3, 4], dtype=torch.float32, device='cuda')
    y = torch.tensor([10, 20, 30, 40], dtype=torch.float32, device='cuda')
    output = add(x, y)
    print(output)  # tensor([11., 22., 33., 44.], device='cuda:0')
```

!

1) How many values can a pid take?
2) What does the mask array look like?

Calling a Triton Kernel from Python

```python
def add(x: torch.Tensor, y: torch.Tensor):
    # We need to preallocate the output.
    output = torch.empty_like(x)
    assert x.device == DEVICE and y.device == DEVICE and output.device == DEVICE
    n_elements = output.numel()
    # The SPMD launch grid denotes the number of kernel instances that run in parallel.
    # It is analogous to CUDA launch grids. It can be either Tuple[int], or Callable(metaparameters) -> Tuple[int].
    # In this case, we use a 1D grid where the size is the number of blocks:
    grid = lambda meta: (triton.cdiv(n_elements, meta['BLOCK_SIZE']), )
    #  - Each torch.tensor object is implicitly converted into a pointer to its first element.
    #  - `triton.jit`'ed functions can be indexed with a launch grid to obtain a callable GPU kernel.
    #  - Don't forget to pass meta-parameters as keywords arguments.
    add_kernel[grid](x, y, output, n_elements, BLOCK_SIZE=1024)
    # We return a handle to z but, since `torch.cuda.synchronize()` hasn't been called, the kernel is still
    # running asynchronously at this point.
    return output
```

Basic Syntax of Triton

- **Jit** Decorator foit Decorator for JIT-compiling a function using the Triton compiler.
- **autotune** Decorator for auto-tuning a triton.jit'd function.
- **heuristics** Decorator for specifying how the values of certain meta-parameters may be computed.
- **Config** An object that represents a possible kernel configuration for the auto-tuner to try.

```
@triton.autotune(
    configs=[
        triton.Config({"BLOCK_SIZE": 1024}, num_warps=4),
        triton.Config({"BLOCK_SIZE": 2048}, num_stages=1),
    ],
    key=["n_elements"],
)
@triton.jit
def _quantize_global(
    x_ptr,
    absmax_inv_ptr,
    output_ptr,
    n_elements,
   BLOCK_SIZE: tl.constexpr,
):
    pid = tl.program_id(axis=0)
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n_elements
    x = tl.load(x_ptr + offsets, mask=mask)
    absmax_inv = tl.load(absmax_inv_ptr)
    output = tl.libdevice.llrint(127.0 * (x * absmax_inv))
    tl.store(output_ptr + offsets, output, mask=mask)

def quantize_global(x: torch.Tensor):
    absmax = x.abs().max().unsqueeze(0)
    absmax_inv = 1.0 / absmax
    output = torch.empty(*x.shape, device="cuda", dtype=torch.int8)
    assert x.is_cuda and output.is_cuda
    n_elements = output.numel()
    grid = lambda meta: (triton.cdiv(n_elements, meta["BLOCK_SIZE"]),)
    _quantize_global[grid](x, absmax_inv, output, n_elements)
    return output, absmax
```

Basic Syntax of Triton - Config

- **kwargs** – a dictionary of meta-parameters to pass to the kernel as keyword arguments.
- **num_warps** – the number of warps to use for the kernel when compiled for GPUs. For example, if num_warps=8, then each kernel instance will be automatically parallelized to cooperatively execute using 8 * 32 = 256 threads.
- **num_stages** – the number of stages that the compiler should use when software-pipelining loops.

```
configs=[
    triton.Config({"BLOCK_SIZE": 1024}, num_warps=4),
    triton.Config({"BLOCK_SIZE": 2048}, num_stages=1),
]
```

Basic Syntax of Triton - heuristics

Decorator for specifying how the values of certain meta-parameters may be computed. This is useful for cases where auto-tuning is prohibitevely expensive, or just not applicable.

```
@triton.heuristics(values={'BLOCK_SIZE': lambda args: triton.next_power_of_2(args['x_size'])})
```

values (dict[str, Callable[[list[Any]], Any]]) – a dictionary of meta-parameter names and functions that compute the value of the meta-parameter. each such function takes a list of positional arguments as input.

Basic Syntax of Triton - autotune

Decorator for auto-tuning a triton.jit function.

1. **configs** (list[triton.Config]):
   - A list of `triton.Config` objects defining configurations.
2. **key (list[str])**:
   - Names of argument(s) that trigger re-evaluation of configurations.
3. **prune_configs_by**:
   - A dictionary of pruning functions:
     - **'perf_model'**: Predicts running time of different configurations.
     - **'top_k'**: Selects top-k configurations based on performance.
     - **'early_config_prune' (optional)**: Prunes configurations based on initial criteria.
4. **reset_to_zero (list[str])**:
   - A list of argument names to reset to zero before evaluating configurations.
5. **restore_value (list[str])**:
   - A list of argument names to restore to their original values after configurations are evaluated.
6. **pre_hook (lambda args, reset_only)**:
   - A function called before kernel execution.
     - `args`: Dictionary of all arguments passed to the kernel.
     - `reset_only`: Boolean indicating whether only reset logic should be applied.
7. **post_hook (lambda args, exception)**:
   - A function called after kernel execution.
     - `args`: Dictionary of all arguments passed.
     - `exception`: Exception raised by the kernel if any error occurs.
8. **do_bench (lambda fn, quantiles)**:
   - A benchmark function to measure kernel execution time at various quantiles.

If the environment variable **TRITON_PRINT_AUTOTUNING=1**, Triton will print a message to stdout after autotuning each kernel, including the time spent autotuning and the best configuration.

Calling a Triton Kernel from Python

```python
import triton
import triton.language as tl
@triton.jit
def add_kernel(
    x_ptr,
    y_ptr,
    output_ptr,
    n_elements,
    BLOCK_SIZE: tl.constexpr,
):
    pid = tl.program_id(axis=0)

    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n_elements

    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)

    output = x + y

    tl.store(output_ptr + offsets, output, mask=mask)
```

```python
def add(x: torch.Tensor, y: torch.Tensor):
    assert x.is_cuda and y.is_cuda
    output = torch.empty_like(x)
    n_elements = x.numel()
    grid = lambda meta: (triton.cdiv(n_elements, meta['BLOCK_SIZE']),)
    add_kernel[grid](x, y, output, n_elements, BLOCK_SIZE=1024)
    return output
```

```python
if __name__ == "__main__":
    x = torch.tensor([1, 2, 3, 4], dtype=torch.float32, device='cuda')
    y = torch.tensor([10, 20, 30, 40], dtype=torch.float32, device='cuda')
    output = add(x, y)
    print(output)  # tensor([11., 22., 33., 44.], device='cuda:0')
```

Basic Syntax of Triton - autotune

```python
import torch
import triton
import triton.language as tl

@triton.autotune(
    configs=[
        triton.Config({"BLOCK_SIZE": 128}, num_stages=3, num_warps=8),
        triton.Config({"BLOCK_SIZE": 256}, num_stages=3, num_warps=8),
        triton.Config({"BLOCK_SIZE": 512}, num_stages=4, num_warps=4),
        triton.Config({"BLOCK_SIZE": 64}, num_stages=4, num_warps=4),
        triton.Config({"BLOCK_SIZE": 32}, num_stages=4, num_warps=4),
        triton.Config({"BLOCK_SIZE": 32}, num_stages=2, num_warps=2),
        triton.Config({"BLOCK_SIZE": 16}, num_stages=4, num_warps=4),
    ],
    key=[],  # No arguments are used to select configurations
)
@triton.jit
def add_kernel(
    x_ptr,
    y_ptr,
    output_ptr,
    n_elements,
    BLOCK_SIZE: tl.constexpr,
):
    pid = tl.program_id(axis=0)
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n_elements
    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)
    output = x + y
    tl.store(output_ptr + offsets, output, mask=mask)

```

```python
def add(x: torch.Tensor, y: torch.Tensor):
    assert x.is_cuda and y.is_cuda
    output = torch.empty_like(x)
    n_elements = x.numel()
    grid = lambda meta: (triton.cdiv(n_elements, meta['BLOCK_SIZE']),)
    add_kernel[grid](x_ptr=x, y_ptr=y, output_ptr=output, n_elements=n_elements)
    return output

if __name__ == "__main__":
   x = torch.tensor([1, 2, 3, 4], dtype=torch.float32, device='cuda')
   y = torch.tensor([10, 20, 30, 40], dtype=torch.float32, device='cuda')
   output = add(x, y)
   print(output)  # tensor([11., 22., 33., 44.], device='cuda:0')
```

NB: When using autotune in Triton, you do not need to explicitly specify the following parameters, as they are automatically determined: BLOCK_SIZE, num_stages, num_warps.

Thus, these parameters are handled internally by the configuration when using autotune.

```
Triton autotuning for function add_kernel finished after 2.38s; best config selected: BLOCK_SIZE: 32, num_warps: 4, num_ctas: 1, num_stages: 4, maxnreg: None;
tensor([11., 22., 33., 44.], device='cuda:0')
```

Basic Syntax of Triton - inline_asm_elementwise

Generate vectorized assembly code for tensors. The input tensors are assumed to have the same shape after broadcasting. The output type can be a tuple of types, with each element being a separate tensor. The assembly processes blocks of data, handling multiple elements at once. An empty return value is not allowed – at least one tensor must be returned for compatibility.

```python
import torch
import triton
import triton.language as tl
@triton.jit
def kernel(A, B, C, D, BLOCK: tl.constexpr):
    a = tl.load(A + tl.arange(0, BLOCK))
    b = tl.load(B + tl.arange(0, BLOCK))

    (c, d) = tl.inline_asm_elementwise(
        asm="""
        {
            .reg .b8 tmp<4>;
            mov.b32 {tmp0, tmp1, tmp2, tmp3}, $8;
            cvt.u32.u8 $0, tmp0;
            cvt.u32.u8 $1, tmp1;
            cvt.u32.u8 $2, tmp2;
            cvt.u32.u8 $3, tmp3;
        }
        cvt.rn.f32.s32 $4, $0;
        cvt.rn.f32.s32 $5, $1;
        cvt.rn.f32.s32 $6, $2;
        cvt.rn.f32.s32 $7, $3;
        max.f32 $4, $4, $9;
        max.f32 $5, $5, $10;
        max.f32 $6, $6, $11;
        max.f32 $7, $7, $12;
        """,
        constraints=(
            "=r,=r,=r,=r,=r,=r,=r,=r,"
            "r,r,r,r,r"),
        args=[a, b],
        dtype=(tl.int32, tl.float32),
        is_pure=True,
        pack=4,
    )
    tl.store(C + tl.arange(0, BLOCK), c)
    tl.store(D + tl.arange(0, BLOCK), d)

```

```python
def run_kernel(A, B):
    assert A.is_cuda and B.is_cuda

    N = A.numel()
    BLOCK = 128
    C = torch.empty_like(A, dtype=torch.int32, device='cuda')
    D = torch.empty_like(A, dtype=torch.float32, device='cuda')

    grid = lambda meta: (triton.cdiv(N, meta['BLOCK']),)
    kernel[grid](A, B, C, D, BLOCK=BLOCK)
    return C, D

if __name__ == "__main__":
    A = torch.tensor([40, 50, 3, 4], dtype=torch.uint8, device='cuda')
    B = torch.tensor([10.0, 20.0, 30.0, 40.0], dtype=torch.float32, device='cuda')
    C, D = run_kernel(A, B)
    print("A (input):", A)
    print("B (input):", B)
    print("C (int32 part):", C)
    print("D (float32 part):", D)
```

```
A (input): tensor([40, 50,  3,  4], device='cuda:0', dtype=torch.uint8)
B (input): tensor([10., 20., 30., 40.], device='cuda:0')
C (int32 part): tensor([40, 50,  3,  4], device='cuda:0', dtype=torch.int32)
D (float32 part): tensor([40., 50., 30., 40.], device='cuda:0')
```

Basic Syntax of Triton - static_print

```python
@triton.jit
def add_kernel(
    x_ptr,
    y_ptr,
    output_ptr,
    n_elements,
    BLOCK_SIZE: tl.constexpr,
):
    pid = tl.program_id(axis=0)
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n_elements
    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)
    output = x + y
    tl.store(output_ptr + offsets, output, mask=mask)

    tl.static_print(f"BLOCK_SIZE:\t\t\t",BLOCK_SIZE)
    tl.static_print("pid:\t\t\t\t",pid)
    tl.static_print("block_start:\t\t\t",block_start)
    tl.static_print("tl.arange(0, BLOCK_SIZE):\t",tl.arange(0, BLOCK_SIZE))
    tl.static_print("mask:\t\t\t\t",mask)
    tl.static_print("x:\t\t\t\t",x)
    tl.static_print("x_ptr:\t\t\t\t",x_ptr)
    tl.static_print("output:\t\t\t\t",output)
    tl.static_print("output_ptr+offsets\t\t", output_ptr + offsets)
```

```
BLOCK_SIZE:                 32
pid:                        int32[]
block_start:                int32[]
tl.arange(0, BLOCK_SIZE):   int32[constexpr[32]]
mask:                       int1[constexpr[32]]
x:                          fp32[constexpr[32]]
x_ptr:                      pointer<fp32>[]
output:                     fp32[constexpr[32]]
output_ptr+offsets          pointer<fp32>[constexpr[32]]
```

Basic Syntax of Triton - device_print

```python
@triton.jit
def add_kernel(
    x_ptr,
    y_ptr,
    output_ptr,
    n_elements,
    BLOCK_SIZE: tl.constexpr,
):
    pid = tl.program_id(axis=0)
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n_elements
    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)
    output = x + y
    tl.store(output_ptr + offsets, output, mask=mask)

    tl.device_print(f"BLOCK_SIZE:\t\t\t",BLOCK_SIZE)
    tl.device_print("pid:\t\t\t\t",pid)
    tl.device_print("block_start:\t\t\t",block_start)
    tl.device_print("tl.arange(0, BLOCK_SIZE):\t",tl.arange(0, BLOCK_SIZE))
    tl.device_print("mask:\t\t\t\t",mask)
    tl.device_print("x:\t\t\t\t",x)
    tl.device_print("x_ptr:\t\t\t\t",x_ptr)
    tl.device_print("output:\t\t\t\t",output)
    tl.device_print("output_ptr+offsets\t\t", output_ptr + offsets)
```

```
pid (0, 0, 0) idx () BLOCK_SIZE:                    : 32
pid (0, 0, 0) idx () pid:                           : 0
pid (0, 0, 0) idx () block_start:                   : 0

pid (0, 0, 0) idx ( 0) tl.arange(0, BLOCK_SIZE):    : 0
pid (0, 0, 0) idx ( 1) tl.arange(0, BLOCK_SIZE):    : 1
...
pid (0, 0, 0) idx ( 0) tl.arange(0, BLOCK_SIZE):    : 31

pid (0, 0, 0) idx ( 0) mask:                        : 4294967295
pid (0, 0, 0) idx ( 1) mask:                        : 4294967295
pid (0, 0, 0) idx ( 2) mask:                        : 4294967295
pid (0, 0, 0) idx ( 3) mask:                        : 4294967295
pid (0, 0, 0) idx ( 4) mask:                        : 0
pid (0, 0, 0) idx ( 5) mask:                        : 0

pid (0, 0, 0) idx ( 0) x:                           : 1.000000
pid (0, 0, 0) idx ( 1) x:                           : 2.000000
pid (0, 0, 0) idx ( 2) x:                           : 3.000000
pid (0, 0, 0) idx ( 3) x:                           : 4.000000
pid (0, 0, 0) idx ( 4) x:                           : 0.000000
...
pid (0, 0, 0) idx (31) x:                           : 0.000000

pid (0, 0, 0) idx () x_ptr:                         : 0x7f9122600000

pid (0, 0, 0) idx ( 0) output:                      : 11.000000
pid (0, 0, 0) idx ( 1) output:                      : 22.000000
pid (0, 0, 0) idx ( 2) output:                      : 33.000000
pid (0, 0, 0) idx ( 3) output:                      : 44.000000
pid (0, 0, 0) idx ( 4) output:                      : 0.000000
...
pid (0, 0, 0) idx (31) output:                      : 0.000000

pid (0, 0, 0) idx ( 0) output_ptr+offsets           : 0x7f9122600400
pid (0, 0, 0) idx ( 1) output_ptr+offsets           : 0x7f9122600404
pid (0, 0, 0) idx ( 2) output_ptr+offsets           : 0x7f9122600408
pid (0, 0, 0) idx ( 3) output_ptr+offsets           : 0x7f912260040c
...
pid (0, 0, 0) idx (31) output_ptr+offsets           : 0x7f912260047c
```

Basic Syntax of Triton - device_print

On CUDA, printfs are streamed through a buffer of limited size (on one host, we measured the default as 6912 KiB, but this may not be consistent across GPUs and CUDA versions). If you notice some printfs are being dropped, you can increase the buffer size by calling

```
triton.runtime.driver.active.utils.set_printf_fifo_size(size_bytes)
```

CUDA may raise an error if you try to change this value after running a kernel that uses printfs. The value set here may only affect the current device (so if you have multiple GPUs, you'd need to call it multiple times).

References

1. (Documentation) https://triton-lang.org/main/index.html
2. (Documentation Tutorials) https://triton-lang.org/main/getting-started/tutorials/index.html
3. (Git repo) https://github.com/triton-lang/triton
4. (Official publication) https://www.eecs.harvard.edu/~htk/publication/2019-mapl-tillet-kung-cox.pdf
5. (Triton puzzles) https://github.com/triton-lang/triton
6. (Solved Triton puzzles) https://github.com/alexzhang13/Triton-Puzzles-Solutions
7. (Tutorial Part1) https://readmedium.com/understanding-the-triton-tutorials-part-1-6191b59ba4c
8. (Tutorial Part2) https://readmedium.com/understanding-triton-tutorials-part-2-f6839ce50ae7
9. (Triton autocompiler) https://dev-discuss.pytorch.org/t/torchinductor-a-pytorch-native-compiler-with-define-by-run-ir-and-symbolic-shapes/747
10. (Pointer arithmetic in C) https://metanit.com/c/tutorial/
11. (Use example) https://github.com/bitsandbytes-foundation/bitsandbytes/tree/main/bitsandbytes/triton
12. (Practical tasks) https://docs.google.com/spreadsheets/d/1bD2Nr1UEzj2hsJVqkB0uNLkP6dkepcgYgN_7HK2F5HQ/edit?usp=sharing
13. (OpenAI overview): https://openai.com/index/triton/