Triton Join Code

For PCI-e

Github repo : https://github.com/hvdrk/Triton

Guide : https://github.com/hvdrk/Triton/asdfn

DistributedNumaMem

```rust
    let (cache_node, spill_node) =
        if let MemType::DistributedNumaMem { nodes, .. } = partitions_mem_type {
            if let [cache_node, spill_node] = *nodes {
                Ok((cache_node.node, spill_node.node))
            } else {
                Err(ErrorKind::InvalidArgument(
                    "Partitioned memory type define exactly two NUMA nodes".to_string(),
                ))
            }
        } else {
            Err(ErrorKind::InvalidArgument(
                "Partitioned memory type must be DistributedNumaMem".to_string(),
            ))
        }?;
```

radix-join/src/execution_methods/gpu_triton_join.rs

Only for DistributedNumaMem!!

```rust
#[derive(Clone, Debug, PartialEq)]
pub enum MemType {
    /// System memory allocated with Rust's global allocator
    SysMem,
    /// Aligned system memory allocated with Rust's global allocator
    ///
    /// Alignment is specified in bytes.
    AlignedSysMem { align_bytes: usize },
    /// NUMA memory allocated on the specified NUMA node and with the specified page type
    NumaMem { node: u16, page_type: PageType },
    /// NUMA memory allocated on the specified NUMA node and pinned with CUDA
    NumaPinnedMem { node: u16, page_type: PageType },
    /// NUMA memory distributed in proportion to a ratio over multiple NUMA nodes
    DistributedNumaMem {
        nodes: Box<[NodeRatio]>,
        page_type: PageType,
    },
    /// NUMA memory distributed over multiple NUMA nodes using a length per node
    DistributedNumaMemWithLen {
        nodes: Box<[NodeLen]>,
        page_type: PageType,
    },
    /// CUDA pinned memory (using cudaHostAlloc())
    CudaPinnedMem,
    /// CUDA unified memory
    CudaUniMem,
    /// CUDA device memory
    CudaDevMem,
}
```

Numa-gpu/src/runtime/allocator.rs

DistributedNumaMem

MemType::DistributedNumaMem

```rust
    DistributedNumaMem {
        nodes: Box<[NodeRatio]>,
        page_type: PageType,
    },
```

Numa-gpu/src/runtime/allocator.rs

Memory::DerefMem

```rust
/// A CPU-dereferencable memory type
///
/// These memory types can be directly accessed on the host.
#[derive(Debug)]
pub enum DerefMem<T: DeviceCopy> {
    /// System memory allocated with Rust's global allocator
    SysMem(Vec<T>),
    /// System memory allocated with Rust's global allocator but as a Rust Box
    ///
    /// Useful for wrapping aliged memory.
    BoxedSysMem(Box<[T]>),
    /// NUMA memory allocated on the specified NUMA node
    NumaMem(NumaMemory<T>),
    /// NUMA memory distributed over multiple NUMA nodes
    DistributedNumaMem(DistributedNumaMemory<T>),
    /// CUDA pinned memory (using cudaHostAlloc())
    CudaPinnedMem(LockedBuffer<T>),
    /// CUDA unified memory
    CudaUniMem(UnifiedBuffer<T>),
}
```

Numa-gpu/src/runtime/memory.rs

fn alloc_distributed_numa

```rust
/// Allocates memory on multiple, specified NUMA nodes.
fn alloc_distributed_numa<T: DeviceCopy>(
    len: usize,
    nodes: Box<[NodeRatio]>,
    page_type: PageType,
) -> DerefMem<T> {
    DerefMem::DistributedNumaMem(DistributedNumaMemory::new_with_ratio(len, nodes, page_type))
}
```

Numa-gpu/src/runtime/allocator.rs

Numa::DistributedNumaMemory<T>

```rust
#[derive(Debug)]
pub struct DistributedNumaMemory<T> {
    ptr: *mut T,
    len: usize,
    node_ratios: Box<[NodeRatio]>,
    page_type: PageType,
    is_memory_locked: bool,
    is_page_locked: bool,
}
```

Numa-gpu/src/runtime/numa.rs

DistributedNumaMem

1. Allocate memory in VA by mmap()

- void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

prot: protect. Read or write

flags: mapping method. MAP_ANONYMOS: not mapped with any file

hugetlb_flag: pagetype

fd: file descripter

offset: starting point in file mapping

```
let hugetlb_flags = match page_type {
    PageType::Huge2MB => libc::MAP_HUGETLB | libc::MAP_HUGE_2MB,
    PageType::Huge16MB => libc::MAP_HUGETLB | libc::MAP_HUGE_16MB,
    PageType::Huge1GB => libc::MAP_HUGETLB | libc::MAP_HUGE_1GB,
    PageType::Huge16GB => libc::MAP_HUGETLB | libc::MAP_HUGE_16GB,
    PageType::Default | PageType::Small | PageType::TransparentHuge => 0,
};
```

ptr

impl<T> DistributedNumaMemory<T>

fn new_with_pages

```
        let size = len * size_of::<T>();
        let ptr = unsafe {
            mmap(
                ptr::null_mut(),
                size,
                libc::PROT_READ | libc::PROT_WRITE,
                libc::MAP_PRIVATE | libc::MAP_ANONYMOUS | hugetlb_flags,
                0,
                0,
            )
        };
```

Numa-gpu/src/runtime/numa.rs

DistributedNumaMem

2. Set page by madvise()

- int madvise(void *addr, size_t length, int advice);

advice: memory policy

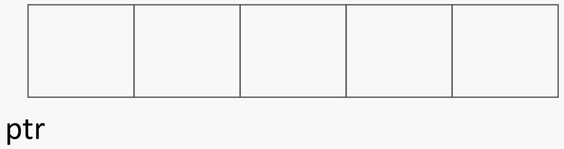
impl<T> DistributedNumaMemory<T>

fn new_with_pages

```rust
let advice = match page_type {
    PageType::Small => Some(libc::MADV_NOHUGEPAGE),
    PageType::TransparentHuge => Some(libc::MADV_HUGEPAGE),
    PageType::Default
    | PageType::Huge2MB
    | PageType::Huge16MB
    | PageType::Huge1GB
    | PageType::Huge16GB => None,
};

if let Some(advice_flag) = advice {
    unsafe {
        if madvise(ptr, size, advice_flag) == -1 {
            let err = IoError::last_os_error();
            std::result::Result::Err::<(), _>(err).expect("Failed to madvise memory");
        }
    }
}
```

Numa-gpu/src/runtime/numa.rs



DistributedNumaMem

3. Binding memory with each Numa node by mbind()

- int mbind(void *start, unsigned long len, int mode, const unsigned long *nodemask,

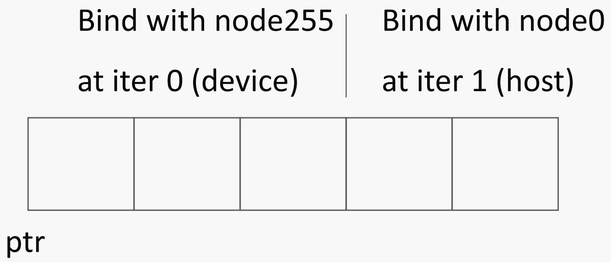
  unsigned long maxnode, unsigned int flags);

  mode: memory policy. BIND->allocate

  nodemask: node bitmask. Ex) 101 -> bind node 0,2

- In code, ptr ~ ptr+slice -> cache node -> device numa mem

  ptr +slice ~ -> spill node -> host numa mem



impl<T> DistributedNumaMemory<T>

fn new_with_pages

```rust
        let final_node_ratios = node_pages
            .iter()
            .scan(
                0,
                |page_offset,
                 &NodeLen {
                     node,
                     len: page_len,
                 }| {
                    let old = *page_offset;
                    *page_offset = *page_offset + page_len;
                    Some((node, old, page_len))
                },
            )
            .map(|(node, page_offset, page_len)| {
                let mut node_set = CpuSet::new();
                node_set.add(node);

                unsafe {
                    let slice = slice::from_raw_parts(
                        ptr.add(page_offset as usize * page_size),
                        page_len as usize * page_size,
                    );

                    // Note that mbind fails with `EINVAL` for HugeTLB mappings if `size`
                    // isn't a multiple of the page size
                    mbind(slice, MemPolicyModes::BIND, node_set, MemBindFlags::STRICT)?;
                }

                Ok(NodeRatio {
                    node,
                    ratio: Ratio::<usize>::new(page_len, pages),
                })
            })
            .collect::<Result<Box<[NodeRatio]>>>()
            .expect("Failed to mbind memory to the specified NUMA nodes");
```

Numa-gpu/src/runtime/numa.rs

PCI-e

Problem

- In PCI-e, device memory doesn’t be catched as Numa node.

  -> but DistributedNumaMemory need two Numa node.

  One for device, the other for host

PCI-e

```rust
    let mut inner_rel_partitions = PartitionedRelation::new(
        data.build_relation_key.len(),
        histogram_algorithm_fst.either(|cpu| cpu.into(), |gpu| gpu.into()),
        radix_bits.pass_radix_bits(RadixPass::First).unwrap(),
        max_chunks_1st,
        inner_rel_alloc(cache_bytes_inner / mem::size_of::<Tuple<T, T>>()),     //distributednuma allocator
        Allocator::mem_alloc_fn(offsets_mem_type.clone()),  //numa allocator
    );
```

radix-join/src/execution_methods/gpu_triton_join.rs

```rust
    radix_prnr.partition(
        RadixPass::First,
        data.build_relation_key.as_launchable_slice(),
        data.build_relation_payload.as_launchable_slice(),
        &mut inner_rel_partition_offsets,
        &mut inner_rel_partitions,
        &stream,
    )?;
```

```rust
    let mut radix_prnr = GpuRadixPartitioner::new(
        histogram_algorithm_fst.gpu_or_else(|cpu_algo| cpu_algo.into()),
        partition_algorithm_fst,
        radix_bits.clone(),
        grid_size,
        block_size,
        dmem_buffer_bytes,
    )?;
    radix_prnr.preallocate_partition_state::<T>(RadixPass::First)?;
```

```rust
#[derive(Debug)]
pub struct PartitionedRelation<T: DeviceCopy> {
    pub relation: Mem<T>,
    pub offsets: Mem<u64>,
    len: usize,
    chunks: u32,
    radix_bits: u32,
}

impl<T: DeviceCopy> PartitionedRelation<T> {
    /// Creates a new partitioned relation, and automatically includes the
    /// necessary padding and metadata.
    pub fn new(
        len: usize,
        histogram_algorithm_type: HistogramAlgorithmType,
        radix_bits: u32,
        max_chunks: u32,
        partition_alloc_fn: MemAllocFn<T>,
        offsets_alloc_fn: MemAllocFn<u64>,
    ) -> Self {
        let chunks: u32 = match histogram_algorithm_type {
            HistogramAlgorithmType::Chunked => max_chunks,
            HistogramAlgorithmType::Contiguous => 1,
        };

        let padding_len = padding_len::<T>();
        let num_partitions = fanout(radix_bits) as usize;
        let relation_len = len + (num_partitions * chunks as usize) * padding_len as usize;

        let relation = partition_alloc_fn(relation_len);
        let offsets = offsets_alloc_fn(num_partitions * chunks as usize);

        Self {
            relation,
            offsets,
            chunks,
            radix_bits,
            len,
        }
    }
```

sql-ops/src/partition/partitioned_relation.rs

```rust
#[derive(Debug)]
pub struct GpuRadixPartitioner {
    radix_bits: RadixBits,
    prefix_sum_algorithm: GpuHistogramAlgorithm,
    partition_algorithm: GpuRadixPartitionAlgorithm,
    prefix_sum_state: PrefixSumState,
    partition_state: RadixPartitionState,
    grid_size: GridSize,
    block_size: BlockSize,
    rp_block_size: BlockSize,
    dmem_buffer_bytes: usize,
}
```

```rust
    pub fn partition<T: DeviceCopy + GpuRadixPartitionable>(
        &mut self,
        pass: RadixPass,
        partition_attr: LaunchableSlice<'_, T>,
        payload_attr: LaunchableSlice<'_, T>,
        partition_offsets: &mut PartitionOffsets<Tuple<T, T>>,
        partitioned_relation: &mut PartitionedRelation<Tuple<T, T>>,
        stream: &Stream,
    ) -> Result<()> {
        T::partition_impl(
            self,
            pass,
            partition_attr,
            payload_attr,
            partition_offsets,
            partitioned_relation,
            stream,
        )
    }
```

```rust
                GpuRadixPartitionAlgorithm::SSWWCv2 => {
                    let name = std::ffi::CString::new(
                        stringify!([<gpu_chunked_sswwc_radix_partition_v2_ $Suffix _ $Suffix>])
                        ).unwrap();
                    let mut function = module.get_function(&name)?;
                    function.set_max_dynamic_shared_size_bytes(max_shared_mem_bytes)?;

                    unsafe {
                        launch!(
                            function<<<
                            grid_size,
                            rp_block_size,
                            max_shared_mem_bytes,
                            stream
                            >>>(
                                args.clone(),
                                max_shared_mem_bytes
                                ))?;
                    }
                },
```

Function in

sql-ops/cudautils/radix_partition.cu

PCI-e

```
template <typename K, typename V>
__device__ void gpu_chunked_sswwc_radix_partition_v2(
    RadixPartitionArgs &args, uint32_t shared_mem_bytes) {
```

Sql-ops/cudautils/radix_partition.cu

```
struct RadixPartitionArgs {
  // Inputs
  const void *const __restrict__ join_attr_data;
  const void *const __restrict__ payload_attr_data;
  std::size_t const data_length;
  uint32_t const padding_length;
  uint32_t const radix_bits;
  uint32_t const ignore_bits;
  const unsigned long long *const __restrict__ partition_offsets;

  // State
  uint32_t *const __restrict__ tmp_partition_offsets;
  char *const __restrict__ l2_cache_buffers;
  char *const __restrict__ device_memory_buffers;
  uint64_t const device_memory_buffer_bytes;

  // Outputs
  void *const __restrict__ partitioned_relation;
};
```

Sql-ops/include/gpu_radix_partition.h

- Just use partitioned_relation in continuous VA
- Except for the NUMA node approach, different buffers can’t be mapped into a contiguous VA

Solution

- 1) use UnifiedBuffer
- 2) Adaptive memory management
- 3) use custom Buffer

Change only rust code

Change almost every code

Solution 1. UnifiedBuffer

rustcuda::memory::UnifiedBuffer

- https://bheisler.github.io/RustaCUDA/rustacuda/memory/struct.UnifiedBuffer.html
- Use Nvidia's API cuMemAllocManaged()

- Works well …, but extremely slow when exceeding device's memory
- Can't map manually, and it is assumed that a page fault occurs with every one access for exceeded

Solution 2. Adaptive memory management

- Use only host memory when exceed device’s memory capacity.

- When the data size exceeds the device memory capacity, there is a slowdown
- However, when the data size is sufficiently large, the performance difference becomes negligible.

Solution 3. Custom Buffer

- The method is quite simple, but it remains unimplemented due to the large number of required modifications

- Make new data struct include device buffer and host numa memory
- Use .get() , .set(), … and adjust for every file include cuda file(.cu), header file(.h), cpp file(.cpp)

Solution 2. Changes

radix-join/Cargo.toml

Add cust = "0.3.2" in [dependencies]

- cust : use CUDA in rust
- For get device’s free memory

radix-join > Cargo.toml

```toml
edition = "2018"

[dependencies]
cstr = "0.2.8"
csv = "~1.1.1"
hostname = "~0.1.5"
itertools = "0.9"
num-rational = "~0.2.0"
num-traits = "~0.2.0"
rayon = "~1.2.0"
rustacuda = { git = "https://github.com/LutzCle/RustaCUDA", branch = "custom_mods_10_2" }
serde = "~1.0.76"
serde_derive = "~1.0.76"
serde_repr = "~0.1"
structopt = "0.3"
cust = "0.3.2"
```

radix-join/src/main.rs

Change fn set_partitions_mem

- Triton join arise error when memory type of 1st partitioned_relation is not DistributedNuma
- Change to NumaPinned when PCI-e

fn set_partitions_mem

```rust
impl CmdOpt {
    fn set_partitions_mem(
        &mut self,
        cache_location: Option<u16>,
        overflow_location: u16,
    ) -> Result<()> {
        if self.execution_method == ArgExecutionMethod::GpuTritonJoinTwoPass {
            if cache_location != None {
                let cache_location = cache_location.ok_or_else(|| {
                    ErrorKind::RuntimeError(
                        "Failed to set the cache NUMA location. Are you using PCI-e?".to_string(),
                    )
                })?;

                if ArgMemType::DistributedNuma != self.partitions_mem_type {
                    self.partitions_mem_type = ArgMemType::DistributedNuma;
                    self.partitions_location = vec![cache_location, overflow_location];
                    self.partitions_proportions = vec![0, 0];
                } else if self.partitions_location.len() != 2 {
                    let e = format!(
                        "Invalid argument: --partitions-location must specify \
                        exactly two locations when combined with --execution-method \
                        GpuTritonJoin\n\
                        The default locations are: --partitions-location {},{}",
                        cache_location, overflow_location
                    );
                    Err(ErrorKind::InvalidArgument(e))?;
                }
            }
            else {
                self.partitions_mem_type = ArgMemType::NumaPinned;
                self.partitions_location = vec![0, 0];
                self.partitions_proportions = vec![0, 0];
            }
        }

        Ok(())
    }
}
```

radix-join/src/gpu_triton_join.rs

Add library

- sql_ops::partition::{fanout, HistogramAlgorithmType},

  sql_ops::partition::partitioned_relation::padding_len

  -> compute 1st partitioned_relation’s total size

- cust::memory::mem_get_info

  -> get device’s free memory

radix-join/src/gpu_triton_join.rs

Change fn gpu_triton_join

- Change parameter partitions_mem_type to mut

  : can be changed after

```rust
pub fn gpu_triton_join<T>(
    data: &mut JoinData<T>,
    hashing_scheme: HashingScheme,
    histogram_algorithm_fst: DeviceType<CpuHistogramAlgorithm, GpuHistogramAlgorithm>,
    histogram_algorithm_snd: DeviceType<CpuHistogramAlgorithm, GpuHistogramAlgorithm>,
    partition_algorithm_fst: DeviceType<CpuRadixPartitionAlgorithm, GpuRadixPartitionAlgorithm>,
    partition_algorithm_snd: DeviceType<CpuRadixPartitionAlgorithm, GpuRadixPartitionAlgorithm>,
    radix_bits: &RadixBits,
    dmem_buffer_bytes: usize,
    max_partitions_cache_bytes: Option<usize>,
    threads: usize,
    cpu_affinity: CpuAffinity,
    mut partitions_mem_type: MemType,
    stream_state_mem_type: MemType,
    page_type: PageType,
    partition_dim: (&GridSize, &BlockSize),
    join_dim: (&GridSize, &BlockSize),
) -> Result<(i64, RadixJoinPoint)>
```

- Set cache_node, spill node to 0(not used) when not DistributedNumaMem

```rust
    let (cache_node, spill_node) =
        if let MemType::DistributedNumaMem { nodes, .. } = partitions_mem_type.clone() {
            if let [cache_node, spill_node] = *nodes {
                Ok((cache_node.node, spill_node.node))
            } else {
                Err(ErrorKind::InvalidArgument(
                    "Partitioned memory type define exactly two NUMA nodes".to_string(),
                ))
            }
        } else {
            Ok((0,0))
        }?;
```

radix-join/src/gpu_triton_join.rs

Change fn gpu_triton_join

- Change offsets_mem_type to NumaPinnedMem when not DistributedNumaMem

  : device can’t access to NumaMem

```rust
    let offsets_mem_type = match partitions_mem_type {
        MemType::DistributedNumaMem => MemType::NumaMem {
            node: spill_node,
            page_type,
        },
        _ => partitions_mem_type.clone()
    }
```

- Get device’s free memory

```rust
    // get device's free memory
    let free_mem = if let Ok((free_mem, _)) = mem_get_info() {
        free_mem - GPU_MEM_SLACK_BYTES
    } else {
        0
    };
```

radix-join/src/gpu_triton_join.rs

Change fn gpu_triton_join

- Compute 1st partitioned relation's total size

```rust
    // compute 1st partitioned relation's total size
    let padding_len = padding_len::<Tuple<T, T>>();
    let chunks: u32 = match histogram_algorithm_fst.either(|cpu| cpu.into(), |gpu| gpu.into()) {
        HistogramAlgorithmType::Chunked => max_chunks_1st,
        HistogramAlgorithmType::Contiguous => 1,
    };
    let num_partitions = fanout(radix_bits.pass_radix_bits(RadixPass::First).unwrap()) as usize;
    let inner_relation_len = data.build_relation_key.len() + (num_partitions * chunks as usize) * padding_len as usize;
    let inner_relation_size = inner_relation_len * (mem::size_of::<Tuple<T, T>>() as usize);
    let outer_relation_len = data.probe_relation_key.len() + (num_partitions * chunks as usize) * padding_len as usize;
    let outer_relation_size = outer_relation_len * (mem::size_of::<Tuple<T, T>>() as usize);
    let total_relation_size = inner_relation_size + outer_relation_size;
```

- Change partitions_mem_type when device's free memory is larger than total relation size

```rust
    // if device's free memory is larger then total_relation_size, use device's memory
    if free_mem > total_relation_size {
        partitions_mem_type = MemType::CudaDevMem;
    }
```

radix-join/src/gpu_triton_join.rs

Change fn gpu_triton_join

- Revise inner/outer relation allocator to not spill when not DistributedNumaMem

```rust
    let (inner_rel_alloc, cached_build_tuples) = match partitions_mem_type {
        MemType::DistributedNumaMem {..} => Allocator::mem_spill_alloc_fn(CacheSpillType::CacheAndSpill {
            cache_node,
            spill_node,
            page_type,
        }),
        _ => Allocator::mem_spill_alloc_fn(CacheSpillType::NoSpill(partitions_mem_type.clone()))
    };
```

```rust
    let (outer_rel_alloc, cached_probe_tuples) = match partitions_mem_type {
        MemType::DistributedNumaMem {..} => Allocator::mem_spill_alloc_fn(CacheSpillType::CacheAndSpill {
            cache_node,
            spill_node,
            page_type,
        }),
        _ => Allocator::mem_spill_alloc_fn(CacheSpillType::NoSpill(partitions_mem_type.clone()))
    };
```

sql-ops/src/

Change private fn to public

- Used for compute 1st partitioned relation's total size

```rust
pub mod cpu_radix_partition;
pub mod gpu_radix_partition;
mod partition_input_chunk;
pub mod partitioned_relation;
```

```rust
pub fn fanout(radix_bits: u32) -> u32 {
    1 << radix_bits
}
```

sql-ops/src/partitions.rs

```rust
/// Convert padding bytes into padding length for the type `T`
pub fn padding_len<T: Sized>() -> u32 {
    crate::constants::PADDING_BYTES / mem::size_of::<T>() as u32
}
```

sql-ops/src/partition/partitioned_relation.rs