Billion-scale similarity search with GPUs

Facebook AI Research

Presented by Jiayin

Faiss 使用场景

- 大量图像，视频检索
- item的embedding检索topK
- word2vec，CNN，etc
- embedding维度：50-1000+维

面临的问题

- 怎样最大化利用GPU资源?
- k-NN图的构建是最耗资源的操作
- K-NN是一个有向图，每个节点代表一个实体，实体的出边代表和它最近的k个邻居
- 亿级别的检索碰到“维度诅咒” — 穷尽检索和精确索引都不现实

Faiss 的贡献

- GPU上面的k-选择算法，在寄存器内存中完成，并且给出复杂度分析
- GPU上面找k近邻的近最优算法，搜索近似和精确解
- 实验证实Faiss效率出众

问题定义

$L = k\text{-argmin}_{i=0:\ell} \|x - y_i\|_2$

- 在yi中搜索x的k-近邻，用L2距离
- 批量化：在多个CPU线程或者GPU上面，并行搜索 $[x_j]_{j=0:n_q}$ $(x_j \in \mathbb{R}^d)$ q个查询的k-近邻
- 精确搜索：计算两两之间距离矩阵 $[\|x_j - y_i\|_2^2]_{j=0:n_q, i=0:\ell} \in \mathbb{R}^{n_q \times \ell}$
- 距离展开 $\|x_j - y_i\|_2^2 = \|x_j\|^2 + \|y_i\|^2 - 2\langle x_j, y_i \rangle$ 内积难算
- 等价于计算 $XY^\top$，然后选每行里的topK

问题定义(续)

- 压缩域搜索(近似近邻搜索)
- IVFADC索引结构 $y \approx q(y) = q_1(y) + q_2(y - q_1(y))$
- 其中q1是粗糙量化器，q2是精细量化器，q为编码器
- 不对称距离计算(ADC)搜索返回近似解 $L_{\mathrm{ADC}} = k\text{-}\mathrm{argmin}_{i=0:\ell} \|x - q(y_i)\|_2$
- IVF $L_{\mathrm{IVF}} = \tau\text{-}\mathrm{argmin}_{c \in \mathcal{C}_1} \|x - c\|_2.$
- $multi\text{-}probe\ parameter\ \tau$ ： q1的簇心个数

问题定义(续)

- IVFADC $L_{\mathrm{IVFADC}} = \underset{i=0:\ell \text{ s.t. } q_1(y_i)\in L_{\mathrm{IVF}}}{k\text{-argmin}} \|x - q(y_i)\|_2.$
- 数据结构：倒排文件
- 将yi分组到 $|\mathcal{C}_1|$ 个倒排列表中 $\mathcal{I}_1, ..., \mathcal{I}_{|\mathcal{C}_1|}$，其中对每个yi，q1是统一的
- 计算L_{IVFADC} 耗非常多内存，随后就是线性扫描τ 个倒排列表
- quantizer q1：q1的簇心个数一般取较小 $|C_1| \approx \sqrt{\ell}$, ℓ是item个数，用k-means计算簇
- quantizer q2：可以有更多的内存来表示（一个簇里的点个数有限）

问题定义(续)

- 乘积量化
- 把y分成b个子向量 $y = [y^0 ... y^{b-1}]$，其中b是向量维度d的因子
- $(q^0(y^0), ..., q^{b-1}(y^{b-1}))$ 然后将这b个值拼接起来，就是q2(y)
- q2(y) 为b-byte编码，每个分量一个byte（8位）
- $|\mathcal{C}_2| = 256^b$

GPU回顾和k-选择算法

- GPU单线程叫做lane，32个线程叫做warp
- 32个warp组成一个block，或者co-operate thread array，每个CUDA线程分配一个block relative ID，叫进程ID，可以用来分配任务
- 每个block运行在一个GPU核，也叫流式多处理器

K-选择算法

- 问题描述：一般数组元素个数较多，不能全部加载进内存，而k较小
- 通常来说，在CPU上，最大堆比较适合topK算法。但是数据结构堆没有并行特性，因为每次树更新是串行的，不能用SIMD单元
- GPU堆 — GPU并行优先队列可以达到并行特性

GPU上的快速k-选择算法

- 寄存器内排序

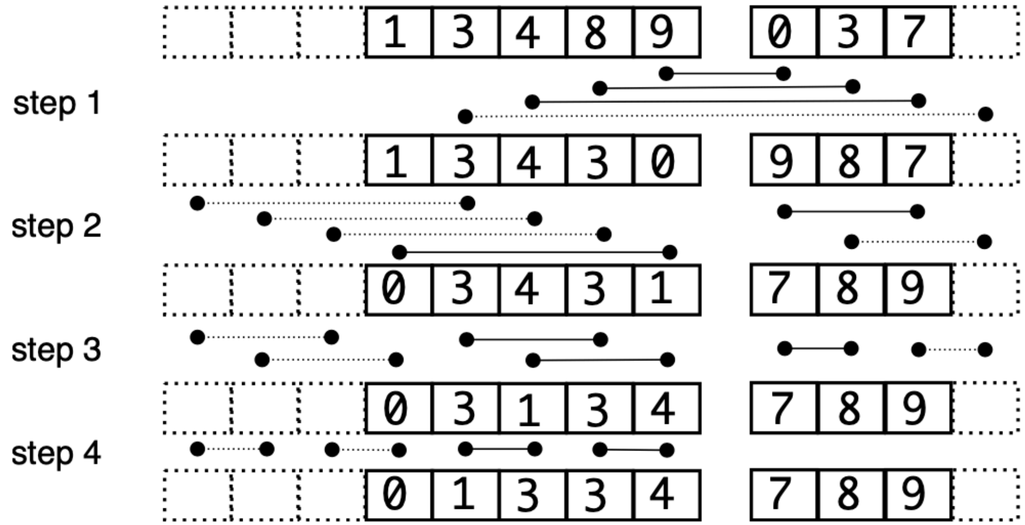

Figure 1: Odd-size network merging arrays of sizes 5 and 3. Bullets indicate parallel compare/swap. Dashed lines are elided elements or comparisons.

寄存器内排序

Algorithm 1 Odd-size merging network

function MERGE-ODD($[L_i]_{i=0:\ell_L}, [R_i]_{i=0:\ell_R}$)
  parallel for $i \leftarrow 0 : \min(\ell_L, \ell_R)$ **do**
    ▷ inverted 1st stage; inputs are already sorted
    COMPARE-SWAP($L_{\ell_L - i - 1}, R_i$)
  end for
  parallel do
    ▷ If $\ell_L = \ell_R$ and a power-of-2, these are equivalent
    MERGE-ODD-CONTINUE($[L_i]_{i=0:\ell_L}$, `left`)
    MERGE-ODD-CONTINUE($[R_i]_{i=0:\ell_R}$, `right`)
  end do
end function
function MERGE-ODD-CONTINUE($[x_i]_{i=0:\ell}, p$)
  if $\ell > 1$ **then**
    $h \leftarrow 2^{\lceil \log_2 \ell \rceil - 1}$ ▷ largest power-of-2 $< \ell$
    parallel for $i \leftarrow 0 : \ell - h$ **do**
      ▷ Implemented with warp shuffle butterfly
      COMPARE-SWAP(x_i, x_{i+h})
    end for
    parallel do
      if $p =$ `left` **then** ▷ left side recursion
        MERGE-ODD-CONTINUE($[x_i]_{i=0:\ell-h}$, `left`)
        MERGE-ODD-CONTINUE($[x_i]_{i=\ell-h:\ell}$, `right`)
      else ▷ right side recursion
        MERGE-ODD-CONTINUE($[x_i]_{i=0:h}$, `left`)
        MERGE-ODD-CONTINUE($[x_i]_{i=h:\ell}$, `right`)
      end if
    end do
  end if
end function

Algorithm 2 Odd-size sorting network

function SORT-ODD($[x_i]_{i=0:\ell}$)
  if $\ell > 1$ **then**
    parallel do
      SORT-ODD($[x_i]_{i=0:\lfloor \ell/2 \rfloor}$)
      SORT-ODD($[x_i]_{i=\lfloor \ell/2 \rfloor:\ell}$)
    end do
    MERGE-ODD($[x_i]_{i=0:\lfloor \ell/2 \rfloor}, [x_i]_{i=\lfloor \ell/2 \rfloor:\ell}$)
  end if
end function

Warp选择

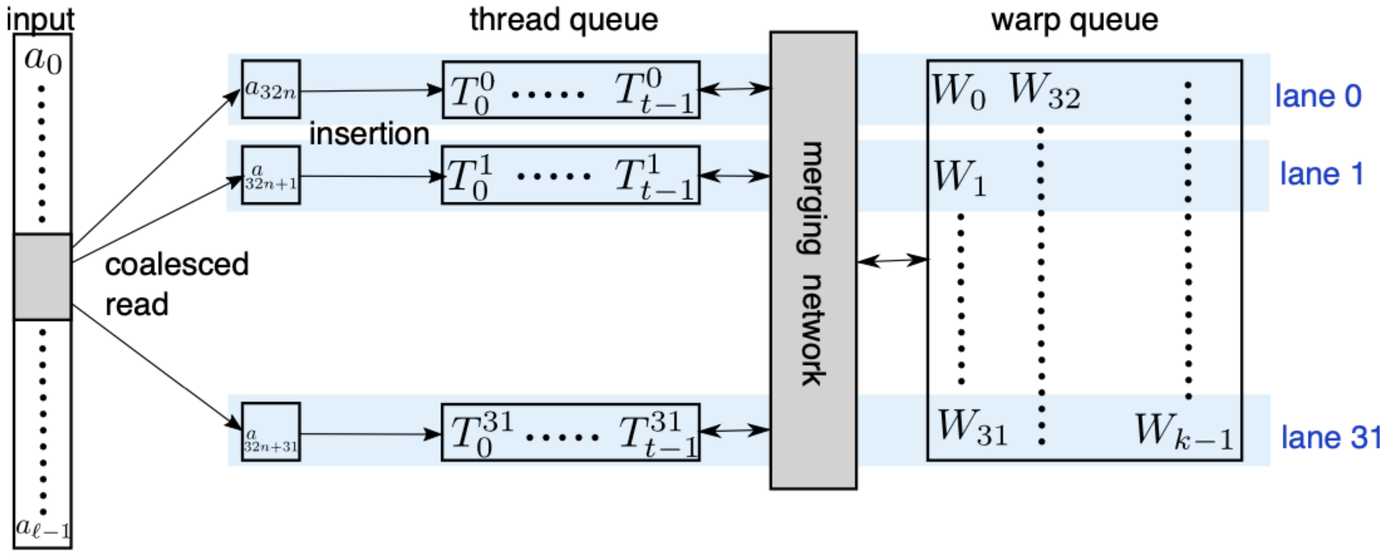

Figure 2: Overview of WarpSelect. The input values stream in on the left, and the warp queue on the right holds the output result.

Warp选择

Algorithm 3 WarpSelect pseudocode for lane j

function WarpSelect(a)
  if $a < T_0^j$ **then**
    insert a into our $[T_i^j]_{i=0:t}$
  end if
  if WARP-BALLOT($T_0^j < W_{k-1}$) **then**
    ▷ Reinterpret thread queues as lane-stride array
    $[\alpha_i]_{i=0:32t} \leftarrow \text{CAST}([T_i^j]_{i=0:t,j=0:32})$
    ▷ concatenate and sort thread queues
    SORT-ODD($[\alpha_i]_{i=0:32t}$)
    MERGE-ODD($[W_i]_{i=0:k}, [\alpha_i]_{i=0:32t}$)
    ▷ Reinterpret lane-stride array as thread queues
    $[T_i^j]_{i=0:t,j=0:32} \leftarrow \text{CAST}([\alpha_i]_{i=0:32t})$
    REVERSE-ARRAY($[T_i]_{i=0:t}$)
    ▷ Back in thread queue order, invariant restored
  end if
end function