Performance analysis of HNSW algorithm under parallel execution: C++ vs CUDA

원종빈 2025-23160

Introduction

최근 대규모 임베딩 기반 검색은 추천 시스템, 정보 검색, 그리고 RAG(Retrieval-Augmented Generation)와 같은 현대 AI 서비스의 핵심 구성 요소로 자리 잡았다. 이러한 응용에서 근사 최근접 탐색(Approximate Nearest Neighbor, ANN)은 검색 정확도와 지연 시간 사이의 균형을 결정하는 중요한 문제이며, 그중에서도 Hierarchical Navigable Small World(HNSW) 알고리즘은 높은 탐색 정확도와 우수한 성능으로 널리 채택되고 있다.

그러나 기존의 CPU 기반 HNSW 구현은 멀티스레드 병렬화에 제약이 있으며, 대규모 병렬 연산에 강점을 지닌 GPU를 활용한 성능 개선 가능성은 충분히 탐구되지 않았다. HNSW의 핵심 연산인 고차원 벡터 간 거리 계산과 그래프 탐색 과정은 수천 개의 코어를 가진 GPU 아키텍처에서 대규모 병렬화의 이점을 누릴 수 있는 여지가 있다.

본 연구에서는 널리 사용되는 C++ 기반 HNSW 구현을 GPU(CUDA) 레벨로 병렬화하고, 그에 따른 성능을 비교 분석한다. 구체적으로 warp 단위 거리 계산과 block 단위 검색 병렬화라는 두 가지 최적화 기법을 적용하고, 10,000개의 Wikipedia 문서 임베딩을 활용한 실험을 통해 latency와 throughput 관점에서 GPU 병렬화의 실질적인 성능 이득과 한계를 규명한다.

Background: HNSW algorithm

HNSW(Hierarchical Navigable Small World)는 계층적 그래프 구조를 활용한 근사 최근접 이웃 탐색 알고리즘으로, 각 계층에서 탐욕적(greedy) 탐색을 통해 상위 레벨에서 하위 레벨로 점진적으로 최근접 이웃을 찾아간다. 이 과정에서 후보 노드들과의 거리 계산을 반복하며, 실용적인 탐색 속도와 높은 검색 정확도를 동시에 달성한다.

HNSW의 성능 특성은 고차원 벡터 간 거리 계산이 핵심 연산이지만, 계산량보다는 그래프 탐색 중 발생하는 불규칙한 메모리 접근 패턴이 병목으로 작용한다. 탐색 경로가 데

이터에 따라 동적으로 결정되기 때문에 예측 불가능한 메모리 접근이 빈번하게 발생하며, 이는 CPU와 GPU에서 서로 다른 성능 특성으로 나타난다. CPU는 강력한 캐시 계층과 분기 예측 메커니즘을 통해 이러한 불규칙성을 효과적으로 처리하는 반면, GPU는 수천 개의 코어를 활용한 대규모 병렬 처리로 거리 계산과 다중 쿼리 동시 처리에서 강점을 보인다. 본 연구에서는 이러한 아키텍처 차이가 HNSW 알고리즘의 실제 성능에 미치는 영향을 실험적으로 분석한다.

Experiment setup

본 연구는 GSDS 공용 컴퓨팅 서버에서 launch-shell 명령어를 통해 할당받은 독립 터미널에서 수행되었다. 실험에 사용된 하드웨어 및 소프트웨어 환경은 다음과 같다.

- CPU: 1x AMD EPYC 7502
- GPU: 1x NVIDIA RTX 3090 (24GB VRAM)
- RAM: 50GB RAM (Default value)
- S/W: Ubuntu 20.04.6 LTS, CUDA 11.7

CPU 기반 HNSW 구현으로는 가장 널리 사용되는 오픈소스 라이브러리인 hnswlib(https://github.com/nmslib/hnswlib)을 활용하였다. 이 구현은 SIMD 명령어를 통해 벡터 연산을 가속화하며, 멀티스레드 기반 인덱스 구축과 검색을 지원한다. 이때 HNSW 알고리즘의 파라미터로는 해당 레포지토리에서 설정해둔 기본값, M=16, ef_construction=200, ef_search=50)를 사용하였다.

실험 데이터셋으로는 open-domain QA 시스템의 지식 베이스로 널리 활용되는 Wikipedia 텍스트 데이터셋(Salesforce/wikitext)을 사용하였다. 원본 데이터셋은 약 180만 개의 문서로 구성되어 있으나, 본 연구에서는 빠른 실험 수행을 위해 10,000개의 문서를 샘플링하고 이를 128차원 벡터로 임베딩하여 인덱스 구축에 사용하였다. 검색 성능 평가를 위해서는 별도로 분리된 1,940개의 테스트 쿼리를 활용하였다.

Experiment

본 실험에서는 기존 C++ 기반의 HNSW 알고리즘에서 다음 부분들 위주로 개선했다.

1. Warp 단위 거리 계산

먼저 기존 C++ 코드에서 SIMD를 통해서 16-way로 처리하던 거리 계산 연산을, warp 단위의 병렬화를 적용해 32-way 병렬화로 거리 계산을 진행하는 CUDA 커널을 작성했다.

```cpp
// hnswlib/hnswlib/space_l2.h:26-53
static float L2SqrSIMD16ExtAVX512(const void *pVect1v, const void *pVect2v,
                                  const void *qty_ptr) {
    float *pVect1 = (float *) pVect1v;
    float *pVect2 = (float *) pVect2v;
    __m512 sum = _mm512_set1_ps(0);

    while (pVect1 < pEnd1) {
        v1 = _mm512_loadu_ps(pVect1);
        v2 = _mm512_loadu_ps(pVect2);
        diff = _mm512_sub_ps(v1, v2);
        sum = _mm512_add_ps(sum, _mm512_mul_ps(diff, diff));
        pVect1 += 16; pVect2 += 16;
    }
    return TmpRes[0] + ... + TmpRes[15];
}
```

```cpp
__inline__ __device__
cuda_scalar squaresum(const cuda_scalar* a, const cuda_scalar* b,
                      const int num_dims) {
    int warp = threadIdx.x / WARP_SIZE;
    int lane = threadIdx.x % WARP_SIZE;
    static __shared__ cuda_scalar shared[32];

    cuda_scalar val = 0;
    for (int i = threadIdx.x; i < num_dims; i += blockDim.x) {
        cuda_scalar _val = sub(a[i], b[i]);
        val = add(val, mul(_val, _val));
    }

    val = warp_reduce_sum(val);

    if (lane == 0) shared[warp] = val;
    __syncthreads();

    if (warp == 0) val = warp_reduce_sum(val);
    return shared[0];
}
```

Figure 1: (좌) 원본 구현 (C++) / (우) warp 단위 병렬화 커널 구현 (CUDA)

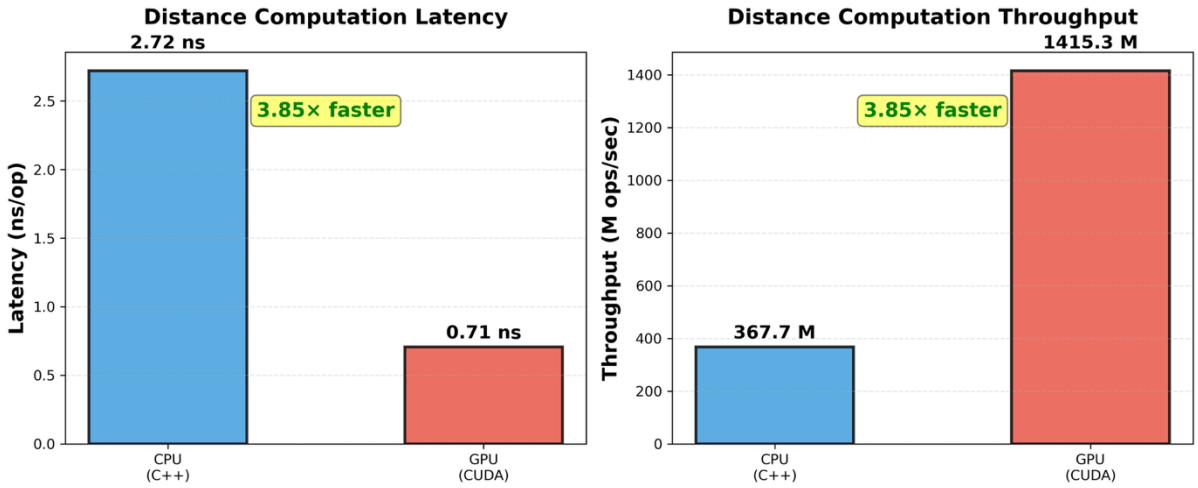

Figure 2: (좌) 거리 계산 연산 latency / (우) 거리 계산 연산 throughput

GPU 병렬화를 통한 거리 계산 연산 속도는 다음과 같이 변화되었다. 먼저, CPU만을 활용한 연산에서는 2.72 ns/op만큼의 계산 지연이 발생했으나, 이를 GPU로 warp 수준에서 병렬화 및 최적화한 결과, 해당 속도가 0.71 ns/op로 감소하여, 결과적으로 3.85배 더 빠른 연산을 가능하게 했다. 한편, 초당 쓰루풋의 경우, CPU에서는 367.7백만개 만큼 처리 가능했으나, GPU 가속화를 통하여 이를 1415.3백만개 만큼 더 많이 처리할 수 있게 되었다.

2. Block-level 검색 최적화

```cpp
// hnswlib/hnswlib/hnswalg.h:311-390
std::priority_queue<...> searchBaseLayerST(tableint ep_id,
                                           const void *data_point, size_t ef) {
    std::priority_queue<...> top_candidates;
    std::priority_queue<...> candidate_set;

    while (!candidate_set.empty()) {
        auto current_node_pair = candidate_set.top();
        candidate_set.pop();

        for (size_t j = 1; j <= size; j++) {
            int candidate_id = *(data + j);
            if (visited_array[candidate_id] == visited_array_tag) continue;

            dist_t dist = fstdistfunc_(data_point, currObj1, dist_func_param_);
            if (top_candidates.size() < ef || lowerBound > dist) {
                candidate_set.emplace(-dist, candidate_id);
                top_candidates.emplace(dist, candidate_id);
            }
        }
    }
    return top_candidates;
}
```

```cpp
__global__ void SearchGraphKernel(const cuda_scalar* qdata,
                                  const int num_qnodes, ...) {
    for (int i = blockIdx.x; i < num_qnodes; i += gridDim.x) {
        const cuda_scalar* src_vec = qdata + i * num_dims;

        int idx = GetCand(ef_search_pq, size, reverse_cand);
        while (idx >= 0) {
            int entry = ef_search_pq[idx].nodeid;

            for (int j = max_m * entry; j < max_m * entry + deg[entry]; ++j) {
                int dstid = graph[j];
                if (CheckVisited(...)) continue;

                const cuda_scalar* dst_vec = data + num_dims * dstid;
                cuda_scalar dist = GetDistanceByVec(src_vec, dst_vec,
                                                    num_dims, dist_type);
                PushNodeToSearchPq(ef_search_pq, &size, ef_search, ...);
            }
            idx = GetCand(ef_search_pq, size, reverse_cand);
        }
    }
}
```

Figure 3: (좌) 직렬 검색 (C++) / (우) Block-level 병렬화 검색 (CUDA)

다음으로, HNSW 알고리즘의 검색 과정에서도 최적화를 진행하였다. CPU 버전이 4개의 스레드로 쿼리를 순차적으로 나누어 처리하는 반면, GPU 버전은 하나의 CUDA 블록이 하나의 쿼리를 담당하며 블록 내 모든 스레드(128-256개)가 협력하여 검색을 수행한다. 이를 통해 1,940개의 쿼리를 동시에 처리할 수 있도록 대규모 병렬화를 구현하였다."

그 결과, 기존 CPU 기반 검색에서는 쿼리당 30.41μs가 걸린 한편, GPU로 최적화한 뒤 이는 4.13 μs까지 줄어들어 최대 7.35배까지 성능이 향상된 것을 확인할 수 있었다. 이에 따라서 쓰루풋 역시 기존 초당 32.9천개의 쿼리에서 향상되었다 241.8천개의 쿼리로 향상된 것을 확인할 수 있었다.

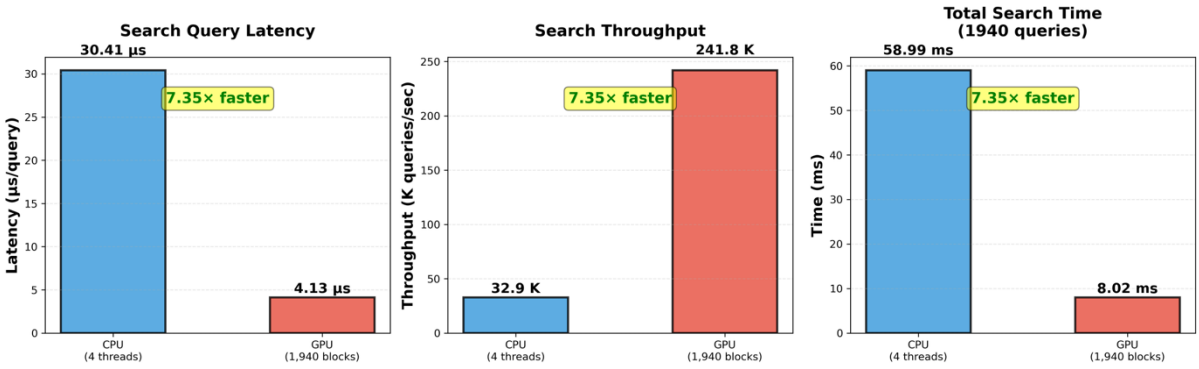

Figure 4: 쿼리당 검색 latency/throughput/total search time 비교 (C++ vs CUDA)

Conclusion & Limitations

본 연구에서는 HNSW 알고리즘의 CPU(C++) 구현을 GPU(CUDA)로 병렬화하여 성능을 비교 분석하였다. 먼저 Warp 단위에서 거리 계산을 최적화하여 연산 지연 시간을 2.72 ns/op에서 0.71 ns/op로 단축시키고(3.85배 가속)켰다. 그리고 검색시에는 block 단위에서 병렬화를 적용해 쿼리당 지연 시간을 30.41 μs에서 4.13 μs로 감소시켰다(7.35배 가속). 이는 GPU의 대규모 병렬 처리가 HNSW의 핵심 연산에서 실질적인 성능 이득을 제공함을 보여준다.

그러나 본 실험에서는 결과를 빠르게 얻기 위해서 10,000개 벡터의 제한된 데이터셋과 128차원의 단순 임베딩을 사용했기에 실제 검색 환경과는 차이가 있다는 한계점이 존재한다. 또한, 검색 속도만을 비교할 뿐 그에 따른 검색 정확도(Recall@K)는 측정하지 않았다는 한계점 또한 내포하고있다. 향후 대규모 데이터셋과 실제로 활용되는 768차원 애상의 고차원 semantic embedding을 활용한 확장성 검증, 그리고 정확도를 포함한 종합적 평가가 필요하다.