Locality Sensitive Hashing: Implementação em CUDA

Autores: Douglas Rodrigues e Thiago Mohallem

Processador: i7-7700

Desempenho

Número de núcleos	4
Nº de threads	8
Frequência baseada em processador	3.60 GHz
Frequência turbo max	4.20 GHz
Cache	8 MB Intel® Smart Cache
Velocidade do barramento	8 GT/s
Nº de links de QPI	0
TDP	65 W

Placa de vídeo : NVIDIA GTX1060 3GB

	6 GB	3 GB
NVIDIA CUDA® Cores	**1280**	**1152**
Clock básico (MHz)	**1506**	**1506**
Boost Clock (MHz)	**1708**	**1708**
Especificações de memória:		
velocidade da memória	**8 Gbps**	**8 Gbps**
Configuração de memória padrão	**6 GB GDDR5**	**3 GB GDDR5**
Largura da interface de memória	**192-bit**	**192-bit**
Largura de banda de memória (GB/s)	**192**	**192**

Máquina da Nuvem

Desempenho

Número de núcleos	12
Nº de threads	24
Frequência baseada em processador	2.60 GHz
Frequência turbo max	3.50 GHz
Cache	30 MB Intel® Smart Cache
Velocidade do barramento	9.6 GT/s
Nº de links de QPI	2
TDP	135 W
Intervalo de voltagem VID	0.65V–1.30V

Intel Xeon E5-2690

SPECIFICATIONS

Virtualization Use Case	**Performance-Optimized Graphics Virtualization**
GPU Architecture	**NVIDIA Maxwell™**
GPUs per Board	**2**
Max User per Board	**32 (16 per GPU)**
NVIDIA CUDA® Cores	**4096 NVIDIA CUDA Cores (2048 per GPU)**
GPU Memory	**16 GB of GDDR5 Memory (8 per GPU)**
H.264 1080p30 streams	**36**
Max Power Consumption	**300 W**
Thermal Solution	**Active/Passive**
Form Factor	**PCIe 3.0 Dual Slot**

NVIDIA Tesla M60

Introdução

Locality Sensitive Hashing (LSH) é uma técnica que utiliza funções hash para diminuir a dimensionalidade de dados e, assim, reduzir o tempo computacional para pesquisa de dados com grande dimensionalidade.

A função hash h(x) deve atender ao seguinte requisito:

Se dois dados, d1 e d2, são similares, então a probabilidade de

$$h(d1) = h(d2) >= p1.$$

Se dois dados, d1 e d2, não são similares, então a probabilidade de

$$h(d1) = h(d2) <= p2$$

Introdução

- Não existe uma definição única de medida de similaridade. E, sim, várias medidas de similaridades de dados.
- Similaridade de cossenos.
- Similaridade euclidiana.
- Similaridade de Jaccard.
- Similaridade de Hamming.

Introdução

- Existem funções hash diferentes, cada uma correspondente a uma métrica de similaridade.
- Similaridade de cossenos -> Superbit.
- Similaridade de Jaccard -> Minhash.
- Similaridade de Hamming -> Bitsampling.

O problema

Funções LSH são usadas para dois problemas principais:

- Computar a assinatura de grande vetores de entradas. Essas assinaturas podem ser usadas para rapidamente estimar a similaridade dos vetores.

- Dado um número de baldes, colocar vetores similares no mesmo balde.

Superbit LSH

SuperBit é uma melhora do Random Projection LSH

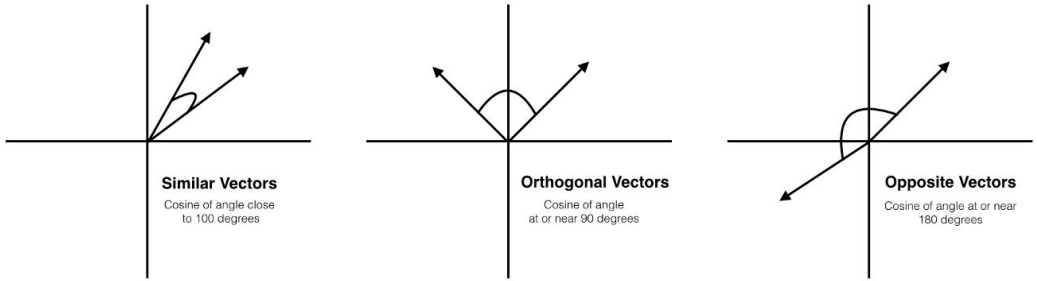
Computa uma similaridade de cosseno.

A similaridade de cosseno entre dois vetores de pontos é o cosseno do ângulo entre eles. E é computado por: v1 . v2 / (|v1| * |v2|)

Vetores com a mesma orientação tem a similaridade de cosseno = 1

Vetores a 90° tem a similaridade 0

Vetores opostos tem uma similaridade de -1.



SuperBit LSH

No Superbit, os K vetores aleatórios são ortogonalizados em L “lotes” de N vetores, onde:

- N é chamado de profundidade do Superbit
- L é o número de superbit.
- K = L * N é o tamanho da assinatura.

Oportunidades de Paralelismo

Nas operações aritméticas:

```
inline void normalize(double* v, int n) {
    double nm = norm(v, n);

    for (int i = 0; i < n; i++)
        v[i] /= nm;
}

inline static double dotProduct(double* v1, double* v2, int n) {
    double sum = 0;

    for (long i = 0; i < n; i++)
        sum += (v1[i] * v2[i]);

    return sum;
}
```

```
inline double* product(double x, double* v, int n) {
    double* result = new double[n];

    for (int i = 0; i < n; i++)
        result[i] = x * v[i];

    return result;
}

inline double* sub(double* v1, double* v2, int n) {
    double* result = new double[n];

    for (int i = 0; i < n; i++)
        result[i] = v1[i] - v2[i];

    return result;
}
```

Oportunidades de Paralelismo

Função principal do algoritmo

Nela a assinatura é feita e depois computada por uma função hash responsável por sua distribuição nos “baldes”

```
void LSH_Superbit::hash(long id, double* attributes) {

    bool* sig = sb→computeSignature(attributes);
    long siglen = sb→getSignatureLength();

    hashSign(id, sig, siglen);
}
```

Exemplo execução serial

Array[100000][50] - inicializado com uma distribuição(0.0,1.0)

```cpp
//começa a medir o tempo aqui
auto start = chrono::steady_clock::now();

LSH_Superbit* lsh = new LSH_Superbit(buckets, stages, ARRAY_SIZE);

auto end = chrono::steady_clock::now();

cout << "LSH Creation : "
    << chrono::duration_cast<chrono::milliseconds>(end - start).count()
    << " ms" << endl;

cout << "Processando entradas..." << endl;
for (long i = 0; i < ARRAY_COUNT; i++) {
    lsh->hash(i, mm[i]);
    delete mm[i];
}

end = chrono::steady_clock::now();
cout << "Computing and Hashing : "
    << chrono::duration_cast<chrono::milliseconds>(end - start).count()
    << " ms" << endl;
```

```
LSH Creation : 84 ms
Processando entradas...
Computing and Hashing : 66491 ms
```

Implementar a computeSignature em Cuda:

```
bool* Superbit::computeSignature(double* v) {
    bool* sig = new bool[hyperp_length];
    double result = 0;
    double *d_v;
    double *d_hyperplanes;
    double *d_result;
    int *d_dimensions;
    int blockSize = 512;
    int numBlocks = (dimensions + blockSize - 1) / blockSize;

    // cudaMallocManaged(&result,sizeof(double));

    cudaMalloc((void **)&d_dimensions,sizeof(int));
    cudaMalloc((void **)&d_result, sizeof(double));
    cudaMalloc((void **)&d_v,dimensions*sizeof(double));
    cudaMalloc((void **)&d_hyperplanes,sizeof(double) * dimensions);

    cudaMemcpy(d_dimensions,&dimensions,sizeof(int),cudaMemcpyHostToDevice);
    cudaMemcpy(d_v,v,sizeof(double) * dimensions,cudaMemcpyHostToDevice);

    for (int i = 0; i < hyperp_length; i++){

        cudaMemcpy(d_hyperplanes,hyperplanes[i],sizeof(double) * dimensions,cudaMemcpyHostToDevice);
        cudaDotProduct<<<numBlocks,blockSize>>>(d_hyperplanes,d_v,d_dimensions,d_result);
        cudaMemcpy(&result, d_result, sizeof(double), cudaMemcpyDeviceToHost);
        sig[i] = (result >= 0.0);

    }

    cudaFree(d_result);
    cudaFree(d_v);
    cudaFree(d_hyperplanes);
    cudaFree(d_dimensions);

    return sig;
}
```

Comparação dotProduct() serial e paralelo

```
__global__
void cudaDotProduct(double* v1, double* v2, int *n,double* result ){
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    *result = 0.0;
    for (int i = index; i < *n; i+=stride){
        // *result += (v1[i] * v2[i]);
        atomicAdds(result, (v1[i]*v2[i]));
    }
}
```

```
inline static double dotProduct(double* v1, double* v2, int n) {
    double sum = 0;

    for (long i = 0; i < n; i++)
        sum += (v1[i] * v2[i]);

    return sum;
}
```

Implementação buildHyperplanes()

```
void Superbit::buildHyperplanes(hpbuilder_t *builderdata) {
    int i, j, k;
    std::default_random_engine generator(builderdata->seed);
    std::normal_distribution<long double> distribution(0.0, 1.0);
    double** v = builderdata->v;
    double** w = builderdata->w;
    int* d_pos;
    int* d_wpos;
    double* d_w;
    double* d_w2;
    double* d_v;
    int* d_dimensions;
    double* retorno;
    int blockSize = 256;
    int numBlocks = (dimensions + blockSize - 1) / blockSize;

    cudaMalloc((void**)&d_pos,sizeof(int));
    cudaMalloc((void**)&d_wpos,sizeof(int));
    cudaMalloc((void**)&d_dimensions,sizeof(int));
    cudaMalloc((void**)&d_w,sizeof(double)*dimensions);
    cudaMalloc((void**)&d_w2,sizeof(double)*dimensions);
    cudaMalloc((void**)&d_v,sizeof(double)*dimensions);
    cudaMalloc((void**)&retorno,sizeof(double)*dimensions);

    for (i = 0; i < hyperp_length; i++) {
        for (j = 0; j < dimensions; j++)
            v[i][j] = distribution(generator);
        Math::normalize(v[i], dimensions);
    }
```

Implementação buildHyperplanes()

```
for (i = 0; i ⩽ (builderdata→length-1); i++) {
    for (j = 1; j ⩽ builderdata→superbit; j++) {
        int pos = i * builderdata→superbit + j - 1;
        // cudaMemcpy(d_pos,&pos,sizeof(int),cudaMemcpyHostToDevice);
        Array::copy(v[pos], w[pos], dimensions);
        cudaMemcpy(d_v,v[pos],sizeof(double)*dimensions,cudaMemcpyHostToDevice);
        for (k = 1; k ⩽ (j-1); k++) {
            int wpos = i * builderdata→superbit + k - 1;
            // cudaMemcpy(d_wpos,&d_wpos,sizeof(int),cudaMemcpyHostToDevice);
            cudaMemcpy(d_w,w[wpos],sizeof(double)*dimensions,cudaMemcpyHostToDevice);
            cudaMemcpy(d_w2,w[pos],sizeof(double)*dimensions,cudaMemcpyHostToDevice);

            cudaBuildHyperplanes<<<1,1>>>(d_w,d_w2,d_v,d_dimensions,retorno);
            cudaMemcpy(w[pos],retorno,sizeof(double)*dimensions,cudaMemcpyDeviceToHost);

        }
        Math::normalize(w[pos], dimensions);
    }
}
```

Problema, Matriz[40000][500] como exemplo

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   98.59%  13.9989s    129679  107.95us  105.12us  137.67us  cudaDotProduct(double*, double*, int*, dou
le*)
                    1.11%  157.79ms    194537     811ns     480ns  13.696us  [CUDA memcpy HtoD]
                    0.27%  38.230ms    129685     294ns     256ns  2.4640us  [CUDA memcpy DtoH]
                    0.03%  3.8753ms         6  645.88us  638.58us  661.34us  cudaBuildHyperplanes(double*, double*, dou
le*, int*, double*)
      API calls:   90.76%  23.5881s    324221  72.753us  6.3000us  21.104ms  cudaMemcpy
                    3.84%  999.18ms    129676  7.7050us  5.7000us  1.0343ms  cudaFree
                    2.75%  715.52ms    129685  5.5170us  4.1000us  1.1745ms  cudaLaunchKernel
                    2.64%  685.99ms    129687  5.2890us  3.1000us  156.35ms  cudaMalloc
                    0.00%  241.10us        97  2.4850us     100ns  118.60us  cuDeviceGetAttribute
                    0.00%  12.700us         1  12.700us  12.700us  12.700us  cuDeviceTotalMem
                    0.00%  6.4000us         1  6.4000us  6.4000us  6.4000us  cuDeviceGetPCIBusId
                    0.00%  1.4000us         3     466ns     300ns     700ns  cuDeviceGetCount
                    0.00%     800ns         2     400ns     100ns     700ns  cuDeviceGet
                    0.00%     700ns         1     700ns     700ns     700ns  cuDeviceGetName
                    0.00%     300ns         1     300ns     300ns     300ns  cuDeviceGetLuid
                    0.00%     200ns         1     200ns     200ns     200ns  cuDeviceGetUuid
```

Resultados do Exemplo anterior

Tempo Total da Execução na GPU: 32046ms

Tempo Total da Execução na CPU: 220ms

Tentativas sendo feitas

(1)Retirada de vetores de duas dimensões para uma dimensão [feito]

(Exemplo dado) Código Serial rodando com uma grande quantidade de dados [feito]

Implementação em Cuda do novo Código Serial [~tentando~]

(1)

Depois

```
bool* Superbit::computeSignature(double* v) {
    long pos;
    bool* sig = new bool[hyperp_length];

    for (long i = 0; i < hyperp_length; i++) {
        pos = i * dimensions;
        sig[i] = (Math::dotProduct(hyperplanes + pos, v, dimensions) ≥ 0.0);
    }

    return sig;
}
```

Antes

```
bool* Superbit::computeSignature(double* v) {
    bool* sig = new bool[hyperp_length];

    for (int i = 0; i < hyperp_length; i++)
        sig[i] = (Math::dotProduct(hyperplanes[i], v, dimensions) ≥ 0.0);

    return sig;
}
```

Dificuldades

Dificuldade em implementar o código na GPU (Alocação de memória, cópia, número de blocos, número de threads por bloco)

Código serial demandou grande parte do tempo para ser implementado em c++, com toda a correção de bugs e conferência de resultados.

Problemas com overflow e etc..

Continuar implementado em Cuda para tentar um bom resultado para o relatório

Não tem como fazer uma análise de speedup e escalabilidade com os resultados que temos.

Referências

- A Java implementation of Locality Sensitive Hashing (LSH).<https://github.com/tdebatty/java-LSH>
- Ji, Jianqiu, et al. "Super-bit locality-sensitive hashing." *Advances in Neural Information Processing Systems*. 2012.