High Performance Graph and Data Analytics Project

Alessandro La Conca - Pablo Giaccaglia

What we have done?

What we have done?

Acceleration of Personalized PageRank using GPUs

What is Personalized PageRank?

What is Personalized PageRank?

briefly!

What is Personalized PageRank?

- The algorithm originally behind Google Search

What is Personalized PageRank?

- The algorithm originally behind Google Search
- Given a web page, it gives a score to the others

What is Personalized PageRank?

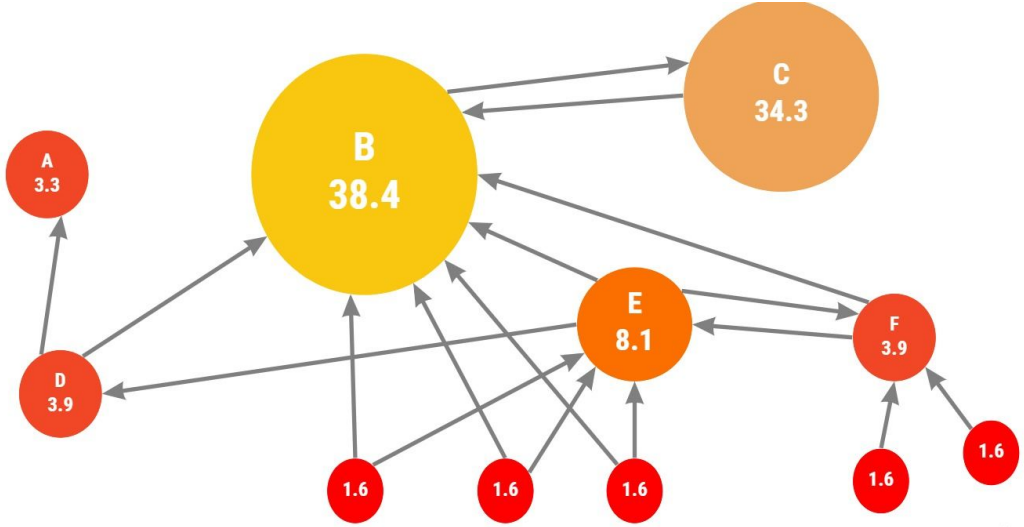
- The algorithm originally behind Google Search
- Given a web page, it gives a score to the others
- A ranking is generated, according to pages' relevance

What is Personalized PageRank?

Personalized PageRank works on graphs!

What is Personalized PageRank?

Personalized PageRank works on graphs!



The formula

The formula

$$\mathbf{p_{t+1}} = d\mathbf{X}\mathbf{p_t} + \frac{d}{|V|}\left(\overline{\mathbf{d}}\mathbf{p_t}\right)\mathbf{1} + (1-d)\overline{\mathbf{v}}$$

The formula

$$\mathbf{p_{t+1}} = d\mathbf{X}\mathbf{p_t} + \frac{d}{|V|}\left(\overline{\mathbf{d}}\mathbf{p_t}\right)\mathbf{1} + (1-d)\overline{\mathbf{v}}$$

Stochastic Matrix

1 for personalization vertex, 0 for other vertices

Personalized PageRank vector

1 for dangling vertices, 0 for other vertices

damping factor

Now, the fun part

Goals & Constraints

Goals & Constraints

- Goal: **make PPR run as fast as possible**

Goals & Constraints

- Goal: **make PPR run as fast as possible**
- **On Wikipedia graph**

Goals & Constraints

- Goal: make PPR run as fast as possible
- On Wikipedia graph
- Using C++ and Cuda

Goals & Constraints

- Goal: make PPR run as fast as possible
- On Wikipedia graph
- Using C++ and Cuda
- At least 80% precision on top-20 results

Goals & Constraints

- Goal: make PPR run as fast as possible
- On Wikipedia graph
- Using C++ and Cuda
- At least 80% precision on top-20 results
- Only ranking matters, not the score

Goals & Constraints

- Goal: make PPR run as fast as possible
- On Wikipedia graph
- Using C++ and Cuda
- At least 80% precision on top-20 results
- Only ranking matters, not the score
- Tests run on NVIDIA Laptop GeForce RTX 3080 GPU

01
Naive Implementation

PPR with Power method

computed once

$$\mathbf{p_{t+1}} = d\mathbf{X}\mathbf{p_t} + \frac{d}{|V|}\left(\overline{\mathbf{d}}\mathbf{p_t}\right)\mathbf{1} + (1-d)\overline{\mathbf{v}}$$

The formula

computed once

$$\mathbf{p_{t+1}} = d\mathbf{X}\mathbf{p_t} + \frac{d}{|V|}\left(\overline{\mathbf{d}}\mathbf{p_t}\right)\mathbf{1} + (1-d)\overline{\mathbf{v}}$$

Graph is stored in COO format

Implemented Kernels

Implemented Kernels

Just the meaningful ones!

Implemented Kernels

compute_dangling_factor_gpu(dangling_gpu, pr_old, pDanglingFact_gpu, *pV_gpu)

$$\left(\overline{\mathbf{d}}\mathbf{p}_{\mathbf{t}}\right)$$

Implemented Kernels

compute_dangling_factor_gpu(dangling_gpu, pr_old, pDanglingFact_gpu, *pV_gpu)

$$\left(\overline{\mathbf{d}}\mathbf{p}_{\mathbf{t}}\right)$$

=

Implemented Kernels

compute_dangling_factor_gpu(dangling_gpu, pr_old, pDanglingFact_gpu, *pV_gpu)

$$\left(\overline{\mathbf{d}}\mathbf{p}_t\right)$$

=

```
template<typename T1, typename T2> __global__ void compute_dangling_factor_gpu( T1 *dangling,  T2* pr, T2 *result, int V){
    for(size_t i = blockIdx.x * blockDim.x + threadIdx.x; i < V; i += blockDim.x * gridDim.x){
        T2 val = (T2) dangling[i] * pr[i];
        atomicAdd(result, val);
    }
}
```

Implemented Kernels

compute_dangling_factor_gpu(dangling_gpu, pr_old, pDanglingFact_gpu, *pV_gpu)

```
template<typename T1, typename T2> __global__ void compute_dangling_factor_gpu( T1 *dangling,  T2* pr, T2 *result, int V){
    for(size_t i = blockIdx.x * blockDim.x + threadIdx.x; i < V; i += blockDim.x * gridDim.x){
        T2 val = (T2) dangling[i] * pr[i];
        atomicAdd(result, val);
    }
}
```

Implemented Kernels

compute_dangling_factor_gpu(dangling_gpu, pr_old, pDanglingFact_gpu, *pV_gpu)

```
template<typename T1, typename T2> __global__ void compute_dangling_factor_gpu( T1 *dangling,  T2* pr, T2 *result, int V){
    for(size_t i = blockIdx.x * blockDim.x + threadIdx.x; i < V; i += blockDim.x * gridDim.x){
        T2 val = (T2) dangling[i] * pr[i];
        atomicAdd(result, val);
    }
}
```

- **Naive dot product kernel**

Implemented Kernels

compute_dangling_factor_gpu(dangling_gpu, pr_old, pDanglingFact_gpu, *pV_gpu)

```
template<typename T1, typename T2> __global__ void compute_dangling_factor_gpu( T1 *dangling,  T2* pr, T2 *result, int V){
    for(size_t i = blockIdx.x * blockDim.x + threadIdx.x; i < V; i += blockDim.x * gridDim.x){
        T2 val = (T2) dangling[i] * pr[i];
        atomicAdd(result, val);
    }
}
```

- Naive dot product kernel
- Linear global memory access pattern

Implemented Kernels

compute_dangling_factor_gpu(dangling_gpu, pr_old, pDanglingFact_gpu, *pV_gpu)

```
template<typename T1, typename T2> __global__ void compute_dangling_factor_gpu( T1 *dangling,  T2* pr, T2 *result, int V){
    for(size_t i = blockIdx.x * blockDim.x + threadIdx.x; i < V; i += blockDim.x * gridDim.x){
        T2 val = (T2) dangling[i] * pr[i];
        atomicAdd(result, val);
    }
}
```

- Naive dot product kernel
- Linear global memory access pattern
- atomicAdd synchronized for the whole GPU

Implemented Kernels

compute_dangling_factor_gpu(dangling_gpu, pr_old, pDanglingFact_gpu, *pV_gpu)

```
template<typename T1, typename T2> __global__ void compute_dangling_factor_gpu( T1 *dangling,  T2* pr, T2 *result, int V){
    for(size_t i = blockIdx.x * blockDim.x + threadIdx.x; i < V; i += blockDim.x * gridDim.x){
        T2 val = (T2) dangling[i] * pr[i];
        atomicAdd(result, val);
    }
}
```

- Naive dot product kernel
- Linear global memory access pattern
- atomicAdd synchronized for the whole GPU
- Many threads are stalled!

Implemented Kernels

cooSPMV(x_gpu,y_gpu,val_gpu,pPpr->E,pr_old,pr_temp)

$$d\mathbf{X}\mathbf{p}_{\mathbf{t}}$$

Implemented Kernels

cooSPMV(x_gpu,y_gpu,val_gpu,pPpr->E,pr_old,pr_temp)

$$d\mathbf{X}\mathbf{p}_{\mathbf{t}}$$

=

Implemented Kernels

cooSPMV(x_gpu,y_gpu,val_gpu,pPpr->E,pr_old,pr_temp)

$$d\mathbf{X}\mathbf{p}_t$$

=

```
template <typename T1, typename T2> __global__ void cooSPMV_math(
        const T1 *  __restrict__  x_gpu,
        const T1 *  __restrict__  y_gpu,
        const T2 * __restrict__ val_gpu,
        const T1 E,
        const T2 * __restrict__ pr_old,
        T2 * __restrict__ pr_temp)
{
    size_t index = blockIdx.x * blockDim.x + threadIdx.x;

    if (index < E){
        atomicAdd(&pr_temp[x_gpu[index]], val_gpu[index]*pr_old[y_gpu[index]]);

    }
}
```

Implemented Kernels

cooSPMV(x_gpu,y_gpu,val_gpu,pPpr->E,pr_old,pr_temp)

```
template <typename T1, typename T2> __global__ void cooSPMV_math(
        const T1 *  __restrict__  x_gpu,
        const T1 *  __restrict__  y_gpu,
        const T2 * __restrict__ val_gpu,
        const T1 E,
        const T2 * __restrict__ pr_old,
        T2 * __restrict__ pr_temp)
{
    size_t index = blockIdx.x * blockDim.x + threadIdx.x;

    if (index < E){
        atomicAdd(&pr_temp[x_gpu[index]], val_gpu[index]*pr_old[y_gpu[index]]);

    }
}
```

- **Same inefficiencies of the previous kernel**

Implemented Kernels

cooSPMV(x_gpu,y_gpu,val_gpu,pPpr->E,pr_old,pr_temp)

```
template <typename T1, typename T2> __global__ void cooSPMV_math(
        const T1 *  __restrict__  x_gpu,
        const T1 *  __restrict__  y_gpu,
        const T2 * __restrict__ val_gpu,
        const T1 E,
        const T2 * __restrict__ pr_old,
        T2 * __restrict__ pr_temp)
{
    size_t index = blockIdx.x * blockDim.x + threadIdx.x;

    if (index < E){
        atomicAdd(&pr_temp[x_gpu[index]], val_gpu[index]*pr_old[y_gpu[index]]);

    }
}
```

- Same inefficiencies of the previous kernel
- Works only with fixed number of threads

Implemented Kernels

cooSPMV(x_gpu,y_gpu,val_gpu,pPpr->E,pr_old,pr_temp)

```
template <typename T1, typename T2> __global__ void cooSPMV_math(
        const T1 *  __restrict__  x_gpu,
        const T1 *  __restrict__  y_gpu,
        const T2 * __restrict__ val_gpu,
        const T1 E,
        const T2 * __restrict__ pr_old,
        T2 * __restrict__ pr_temp)
{
    size t index = blockIdx.x * blockDim.x + threadIdx.x;

    if (index < E){
        atomicAdd(&pr_temp[x_gpu[index]], val_gpu[index]*pr_old[y_gpu[index]]);

    }
}
```

- Same inefficiencies of the previous kernel
- Works only with fixed number of threads
- Each thread does one load, computation and store

Implemented Kernels

cooSPMV(x_gpu,y_gpu,val_gpu,pPpr->E,pr_old,pr_temp)

```
template <typename T1, typename T2> __global__ void cooSPMV_math(
        const T1 *  __restrict__  x_gpu,
        const T1 *  __restrict__  y_gpu,
        const T2 * __restrict__ val_gpu,
        const T1 E,
        const T2 * __restrict__ pr_old,
        T2 * __restrict__ pr_temp)
{
    size_t index = blockIdx.x * blockDim.x + threadIdx.x;

    if (index < E){
        atomicAdd(&pr_temp[x_gpu[index]], val_gpu[index]*pr_old[y_gpu[index]]);

    }
}
```

- Same inefficiencies of the previous kernel
- Works only with fixed number of threads
- Each thread does one load, computation and store
- Thread setup cost not properly “hided”

Implemented Kernels

compute_square_error_gpu(pr_old, pr_gpu, pSquareError_gpu, pPpr->V)

Implemented Kernels

compute_square_error_gpu(pr_old, pr_gpu, pSquareError_gpu, pPpr->V)

```
template<typename T> __global__ void compute_square_error_gpu( T *old,  T *newVector, T *result, int V){
    for (size_t i = blockIdx.x * blockDim.x + threadIdx.x; i < V; i += blockDim.x * gridDim.x) {
        atomicAdd(result, (T)((old[i] - newVector[i]) * (old[i] - newVector[i])));
    }
}
```

Implemented Kernels

compute_square_error_gpu(pr_old, pr_gpu, pSquareError_gpu, pPpr->V)

```
template<typename T> __global__ void compute_square_error_gpu( T *old,  T *newVector, T *result, int V){
    for (size_t i = blockIdx.x * blockDim.x + threadIdx.x; i < V; i += blockDim.x * gridDim.x) {
        atomicAdd(result, (T)((old[i] - newVector[i]) * (old[i] - newVector[i])));
    }
}
```

- **Computes only sum of square of the difference, square root on CPU**

Implemented Kernels

compute_square_error_gpu(pr_old, pr_gpu, pSquareError_gpu, pPpr->V)

```
template<typename T> __global__ void compute_square_error_gpu( T *old,  T *newVector, T *result, int V){
    for (size_t i = blockIdx.x * blockDim.x + threadIdx.x; i < V; i += blockDim.x * gridDim.x) {
        atomicAdd(result, (T)((old[i] - newVector[i]) * (old[i] - newVector[i])));
    }
}
```

- Computes only sum of square of the difference, square root on CPU
- Same inefficiencies of the first kernel

Performance

Instances	Average	Minimum	Maximum	Kernel
900	15,5 ms	15 ms	16 ms	**cooSPMV**
900	6,7 ms	6,5 ms	7,6 ms	**compute_square_error_gpu**
900	6,7 ms	6,5 ms	11,4 ms	**compute_dangling_factor_gpu**

Performance

Instances	Average	Minimum	Maximum	Kernel
900	15,5 ms	15 ms	16 ms	**cooSPMV**
900	6,7 ms	6,5 ms	7,6 ms	**compute_square_error_gpu**
900	6,7 ms	6,5 ms	11,4 ms	**compute_dangling_factor_gpu**

- **Block size = 128**
- **Grid size k1 = (E + Block size - 1) / (Block size)**

Performance

Instances	Average	Minimum	Maximum	Kernel
900	15,5 ms	15 ms	16 ms	**cooSPMV**
900	6,7 ms	6,5 ms	7,6 ms	**compute_square_error_gpu**
900	6,7 ms	6,5 ms	11,4 ms	**compute_dangling_factor_gpu**

- **Block size = 128**
- **Grid size k1 = (E + Block size - 1) / (Block size)**
- **Grid size k2 = (V + Block size - 1) / (Block size)**
- **Grid size k3 = (V + Block size - 1) / (Block size)**

Performance

Instances	Average	Minimum	Maximum	Kernel
900	15,5 ms	15 ms	16 ms	**cooSPMV**
900	6,7 ms	6,5 ms	7,6 ms	**compute_square_error_gpu**
900	6,7 ms	6,5 ms	11,4 ms	**compute_dangling_factor_gpu**

- **Block size = 128**
- **Grid size k1 = (E + Block size - 1) / (Block size)**
- **Grid size k2 = (V + Block size - 1) / (Block size)**
- **Grid size k3 = (V + Block size - 1) / (Block size)**
- **Mean computation time (100 iterations, threshold 1e-6) = 1010 ms**
- **Mean accuracy (100 iterations, threshold 1e-6) = 100%**

After many optimizations...

02

Final coo Implementation

Some improvements

- Floats instead of doubles
- Usage of shared memory
- Some heuristics introduced
- Usage of single precision intrinsics
    - Device only
    - Compiler inline-able set of highly optimized instructions
    - Fast but has low numerical accuracy
    - Allows do things with a small instruction count

Implemented Kernels

Implemented Kernels

Just the meaningful ones!

Implemented Kernels

dangling_kernel(pDanglingIndexes_gpu, pr_old, pDanglingFact_gpu,dampingFract,danglingSize);

$$\left(\overline{\mathbf{d}}\mathbf{p}_{\mathbf{t}}\right)$$

Implemented Kernels

dangling_kernel(pDanglingIndexes_gpu, pr_old, pDanglingFact_gpu,dampingFract,danglingSize);

$$(\overline{\mathbf{d}}\mathbf{p}_t) =$$

Implemented Kernels

dangling_kernel(pDanglingIndexes_gpu, pr_old, pDanglingFact_gpu,dampingFract,danglingSize);

$$\left(\overline{\mathbf{d}}\mathbf{p_t}\right) =$$

```cpp
template<typename T1, typename T2> __global__ void dangling_kernel(T1*x, T2 *y, T2 *dot, T2 dampingFract, unsigned int n){

    unsigned int index = threadIdx.x + blockDim.x*blockIdx.x;
    unsigned int stride = blockDim.x*gridDim.x;

    extern __shared__ T2 cache[];

    T2 temp = 0.0;
    while(index < n){

        temp = __fadd_rd(temp, y[x[index]]);
        index += stride;
    }

    cache[threadIdx.x] = temp;

    __syncthreads();

    // reduction
    unsigned int i = blockDim.x/2;
    while(i != 0){
        if(threadIdx.x < i){
            cache[threadIdx.x] = __fadd_rd(cache[threadIdx.x], cache[threadIdx.x + i]);
        }
        __syncthreads();
        i /= 2;
    }


    if(threadIdx.x == 0){
        atomicAdd(dot, dampingFract * cache[0]);
    }
}
```

Implemented Kernels

dangling_kernel(pDanglingIndexes_gpu, pr_old, pDanglingFact_gpu,dampingFract,danglingSize);

- **Pseudo dot product**

```
template<typename T1, typename T2> __global__ void dangling_kernel(T1*x, T2 *y, T2 *dot, T2 dampingFract, unsigned int n){

    unsigned int index = threadIdx.x + blockDim.x*blockIdx.x;
    unsigned int stride = blockDim.x*gridDim.x;

    extern __shared__ T2 cache[];

    T2 temp = 0.0;
    while(index < n){

        temp = __fadd_rd(temp, y[x[index]]);
        index += stride;
    }

    cache[threadIdx.x] = temp;

    __syncthreads();

    // reduction
    unsigned int i = blockDim.x/2;
    while(i != 0){
        if(threadIdx.x < i){
            cache[threadIdx.x] = __fadd_rd(cache[threadIdx.x], cache[threadIdx.x + i]);
        }
        __syncthreads();
        i /= 2;
    }


    if(threadIdx.x == 0){
        atomicAdd(dot, dampingFract * cache[0]);
    }
}
```

Implemented Kernels

dangling_kernel(pDanglingIndexes_gpu, pr_old, pDanglingFact_gpu,dampingFract,danglingSize);

- Pseudo dot product
- Sum of 0s is avoided

```
template<typename T1, typename T2> __global__ void dangling_kernel(T1*x, T2 *y, T2 *dot, T2 dampingFract, unsigned int n){

    unsigned int index = threadIdx.x + blockDim.x*blockIdx.x;
    unsigned int stride = blockDim.x*gridDim.x;

    extern __shared__ T2 cache[];

    T2 temp = 0.0;
    while(index < n){

        temp = __fadd_rd(temp, y[x[index]]);
        index += stride;
    }

    cache[threadIdx.x] = temp;

    __syncthreads();

    // reduction
    unsigned int i = blockDim.x/2;
    while(i != 0){
        if(threadIdx.x < i){
            cache[threadIdx.x] = __fadd_rd(cache[threadIdx.x], cache[threadIdx.x + i]);
        }
        __syncthreads();
        i /= 2;
    }


    if(threadIdx.x == 0){
        atomicAdd(dot, dampingFract * cache[0]);
    }
}
```

Implemented Kernels

dangling_kernel(pDanglingIndexes_gpu, pr_old, pDanglingFact_gpu,dampingFract,danglingSize);

- Pseudo dot product
- Sum of 0s is avoided
- Shared memory to improve parallelism

```
template<typename T1, typename T2> __global__ void dangling_kernel(T1*x, T2 *y, T2 *dot, T2 dampingFract, unsigned int n){

    unsigned int index = threadIdx.x + blockDim.x*blockIdx.x;
    unsigned int stride = blockDim.x*gridDim.x;

    extern __shared__ T2 cache[];

    T2 temp = 0.0;
    while(index < n){

        temp = __fadd_rd(temp, y[x[index]]);
        index += stride;
    }

    cache[threadIdx.x] = temp;

    __syncthreads();

    // reduction
    unsigned int i = blockDim.x/2;
    while(i != 0){
        if(threadIdx.x < i){
            cache[threadIdx.x] = __fadd_rd(cache[threadIdx.x], cache[threadIdx.x + i]);
        }
        __syncthreads();
        i /= 2;
    }


    if(threadIdx.x == 0){
        atomicAdd(dot, dampingFract * cache[0]);
    }
}
```

Implemented Kernels

dangling_kernel(pDanglingIndexes_gpu, pr_old, pDanglingFact_gpu,dampingFract,danglingSize);

- Pseudo dot product
- Sum of 0s is avoided
- Shared memory to improve parallelism
- Reduction of partial results

```
template<typename T1, typename T2> __global__ void dangling_kernel(T1*x, T2 *y, T2 *dot, T2 dampingFract, unsigned int n){

    unsigned int index = threadIdx.x + blockDim.x*blockIdx.x;
    unsigned int stride = blockDim.x*gridDim.x;

    extern __shared__ T2 cache[];

    T2 temp = 0.0;
    while(index < n){

        temp = __fadd_rd(temp, y[x[index]]);
        index += stride;
    }

    cache[threadIdx.x] = temp;

    __syncthreads();

    // reduction
    unsigned int i = blockDim.x/2;
    while(i != 0){
        if(threadIdx.x < i){
            cache[threadIdx.x] = __fadd_rd(cache[threadIdx.x], cache[threadIdx.x + i]);
        }
        __syncthreads();
        i /= 2;
    }


    if(threadIdx.x == 0){
        atomicAdd(dot, dampingFract * cache[0]);
    }
}
```

Implemented Kernels

dangling_kernel(pDanglingIndexes_gpu, pr_old, pDanglingFact_gpu,dampingFract,danglingSize);

shared memory

```
template<typename T1, typename T2> __global__ void dangling_kernel(T1*x, T2 *y, T2 *dot, T2 dampingFract, unsigned int n){

    unsigned int index = threadIdx.x + blockDim.x*blockIdx.x;
    unsigned int stride = blockDim.x*gridDim.x;

    extern __shared__ T2 cache[];

    T2 temp = 0.0;
    while(index < n){

        temp = __fadd_rd(temp, y[x[index]]);
        index += stride;
    }

    cache[threadIdx.x] = temp;

    __syncthreads();

    // reduction
    unsigned int i = blockDim.x/2;
    while(i != 0){
        if(threadIdx.x < i){
            cache[threadIdx.x] = __fadd_rd(cache[threadIdx.x], cache[threadIdx.x + i]);
        }
        __syncthreads();
        i /= 2;
    }


    if(threadIdx.x == 0){
        atomicAdd(dot, dampingFract * cache[0]);
    }
}
```

Intrinsic function

- Pseudo dot product
- Sum of 0s is avoided
- Shared memory to improve parallelism
- Reduction of partial results

Implemented Kernels

__spmv_coo_flat(x_gpu, y_gpu, val_gpu, pr_old, pr_gpu, pPpr->E, num_blocks, ...)

$$\left(\overline{\mathbf{d}}\mathbf{p}_{\mathbf{t}}\right)$$

- Adaptation from CUSP Library
- Flattens data irregularity (variable number of 0s per row)
- Works with COO format sorted by row to exploit parallelism

Implemented Kernels

__spmv_coo_flat(x_gpu, y_gpu, val_gpu, pr_old, pr_gpu, pPpr->E, num_blocks, ...)

$$\left(\overline{\mathbf{d}}\mathbf{p}_t\right)$$

- Adaptation from CUSP Library
- Flattens data irregularity (variable number of 0s per row)
- Works with COO format sorted by row to exploit parallelism
- Each warp processes an interval of the non-zero values

How it works

__spmv_coo_flat(x_gpu, y_gpu, val_gpu, pr_old, pr_gpu, pPpr->E, num_blocks, ...)

How it works

__spmv_coo_flat(x_gpu, y_gpu, val_gpu, pr_old, pr_gpu, pPpr->E, num_blocks, ...)

- Each warp processes an interval of non-zeros
  - The last active warp may not process the full interval

How it works

__spmv_coo_flat(x_gpu, y_gpu, val_gpu, pr_old, pr_gpu, pPpr->E, num_blocks, ...)

- Each warp processes an interval of non-zeros
  - The last active warp may not process the full interval
- (Example) Given interval of 16 and warp size of 32, for each thread:

How it works

__spmv_coo_flat(x_gpu, y_gpu, val_gpu, pr_old, pr_gpu, pPpr->E, num_blocks, ...)

- Each warp processes an interval of non-zeros
  - The last active warp may not process the full interval
- (Example) Given interval of 16 and warp size of 32, for each thread:
  - Fetch rowIdx, colIdx, value of matrix, fetch vector value, compute product,store result in shared memory.

How it works

__spmv_coo_flat(x_gpu, y_gpu, val_gpu, pr_old, pr_gpu, pPpr->E, num_blocks, ...)

- Each warp processes an interval of non-zeros
  - The last active warp may not process the full interval
- (Example) Given interval of 16 and warp size of 32, for each thread:
  - Fetch rowIdx, colIdx, value of matrix, fetch vector value, compute product,store result in shared memory.
  - If a row extends over the 32 elements boundary, then the partial sum of a warp is carried over into the next warp, otherwise the result is written in global memory. The decision is taken by the first thread of the next warp.

How it works

__spmv_coo_flat(x_gpu, y_gpu, val_gpu, pr_old, pr_gpu, pPpr->E, num_blocks, ...)

- Each warp processes an interval of non-zeros
  - The last active warp may not process the full interval
- (Example) Given interval of 16 and warp size of 32, for each thread:
  - Fetch rowIdx, colIdx, value of matrix, fetch vector value, compute product,store result in shared memory.
  - If a row extends over the 32 elements boundary, then the partial sum of a warp is carried over into the next warp, otherwise the result is written in global memory. The decision is taken by the first thread of the next warp.
  - Each warp conduct a segmented scan by looking if 2 values belong to the same row.

```
            0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15   # thread_lane
      idx [ 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2]  # row indices
      val [ 4, 6, 5, 0, 8, 3, 2, 8, 3, 1, 4, 9, 2, 5, 2, 4]  # M(i,j) * x(j)
→ result: val [ 4,10,15,15,23,26, 2,10,13,14, 4,13,15,20,22,26]  # A(i,j) * x(j)
```

How it works

__spmv_coo_flat(x_gpu, y_gpu, val_gpu, pr_old, pr_gpu, pPpr->E, num_blocks, ...)

- Each thread in a warp, except the last one, checks if it is the last of the row. The last thread writes the sum in the output vector

How it works

__spmv_coo_flat(x_gpu, y_gpu, val_gpu, pr_old, pr_gpu, pPpr->E, num_blocks, ...)

- Each thread in a warp, except the last one, checks if it is the last of the row. The last thread writes the sum in the output vector

- The last thread in each warp writes its row index and partial sum in 2 global arrays (at an index corresponding to that warp).

How it works

__spmv_coo_flat(x_gpu, y_gpu, val_gpu, pr_old, pr_gpu, pPpr->E, num_blocks, ...)

- Each thread in a warp, except the last one, checks if it is the last of the row. The last thread writes the sum in the output vector
- The last thread in each warp writes its row index and partial sum in 2 global arrays (at an index corresponding to that warp).
- These steps are repeated until the warp reaches the end of its interval

How it works

__spmv_coo_flat(x_gpu, y_gpu, val_gpu, pr_old, pr_gpu, pPpr->E, num_blocks, ...)

- Each thread in a warp, except the last one, checks if it is the last of the row. The last thread writes the sum in the output vector
- The last thread in each warp writes its row index and partial sum in 2 global arrays (at an index corresponding to that warp).
- These steps are repeated until the warp reaches the end of its interval
- Finally, the values to carry between warps are added to the result through reduction.

Implemented Kernels

euclidean_kernel_math(pr_old, pr_gpu, pSquareError_gpu, pPpr->V)

```
template<typename T> __global__ void euclidean_kernel_math(const T * __restrict__ x, const T * __restrict__ y, T * __restrict__ result, unsigned int n)
{
    size_t index = threadIdx.x + blockDim.x * blockIdx.x;
    size_t stride = blockDim.x * gridDim.x;

    extern __shared__ T cache1[];

    T temp = 0.0;
    while (index < n) {
        temp =  __fadd_rd(temp, (
                                __fmul_rd(
                                        __fsub_rd(x[index],y[index]),
                                        __fsub_rd(x[index],y[index])
                                )
                        )
        );

        index += stride;
    }

    cache1[threadIdx.x] = temp;

    __syncthreads();

    // reduction
    size_t i = blockDim.x / 2;
    while (i != 0) {
        if (threadIdx.x < i) {
            cache1[threadIdx.x] = __fadd_rd(cache1[threadIdx.x], cache1[threadIdx.x + i]);
        }
        __syncthreads();
        i /= 2;
    }


    if (threadIdx.x == 0) {
        atomicAdd(result, cache1[0]); //
    }

}
```

Implemented Kernels

euclidean_kernel_math(pr_old, pr_gpu, pSquareError_gpu, pPpr->V)

- Adaptation of dot product kernel

```
template<typename T> __global__ void euclidean_kernel_math(const T * __restrict__ x, const T * __restrict__ y, T * __restrict__ result, unsigned int n)
{
    size_t index = threadIdx.x + blockDim.x * blockIdx.x;
    size_t stride = blockDim.x * gridDim.x;

    extern __shared__ T cache1[];

    T temp = 0.0;
    while (index < n) {
        temp =  __fadd_rd(temp, (
                                    __fmul_rd(
                                            __fsub_rd(x[index],y[index]),
                                            __fsub_rd(x[index],y[index])
                                    )
                            )
        );

        index += stride;
    }

    cache1[threadIdx.x] = temp;

    __syncthreads();

    // reduction
    size_t i = blockDim.x / 2;
    while (i != 0) {
        if (threadIdx.x < i) {
            cache1[threadIdx.x] = __fadd_rd(cache1[threadIdx.x], cache1[threadIdx.x + i]);
        }
        __syncthreads();
        i /= 2;
    }


    if (threadIdx.x == 0) {
        atomicAdd(result, cache1[0]); //
    }

}
```

Implemented Kernels

euclidean_kernel_math(pr_old, pr_gpu, pSquareError_gpu, pPpr->V)

- Adaptation of dot product kernel
- Square root computed on CPU

```
template<typename T> __global__ void euclidean_kernel_math(const T * __restrict__ x, const T * __restrict__ y, T * __restrict__ result, unsigned int n)
{
    size_t index = threadIdx.x + blockDim.x * blockIdx.x;
    size_t stride = blockDim.x * gridDim.x;

    extern __shared__ T cache1[];

    T temp = 0.0;
    while (index < n) {
        temp =  __fadd_rd(temp, (
                                __fmul_rd(
                                        __fsub_rd(x[index],y[index]),
                                        __fsub_rd(x[index],y[index])
                                )
                        )
        );

        index += stride;
    }

    cache1[threadIdx.x] = temp;

    __syncthreads();

    // reduction
    size_t i = blockDim.x / 2;
    while (i != 0) {
        if (threadIdx.x < i) {
            cache1[threadIdx.x] = __fadd_rd(cache1[threadIdx.x], cache1[threadIdx.x + i]);
        }
        __syncthreads();
        i /= 2;
    }


    if (threadIdx.x == 0) {
        atomicAdd(result, cache1[0]); //
    }

}
```

Implemented Kernels

euclidean_kernel_math(pr_old, pr_gpu, pSquareError_gpu, pPpr->V)

- Adaptation of dot product kernel
- Square root computed on CPU
- Shared memory to improve parallelism

```
template<typename T> __global__ void euclidean_kernel_math(const T * __restrict__ x, const T * __restrict__ y, T * __restrict__ result, unsigned int n)
{
    size_t index = threadIdx.x + blockDim.x * blockIdx.x;
    size_t stride = blockDim.x * gridDim.x;

    extern __shared__ T cache1[];

    T temp = 0.0;
    while (index < n) {
        temp =  __fadd_rd(temp, (
                                __fmul_rd(
                                        __fsub_rd(x[index],y[index]),
                                        __fsub_rd(x[index],y[index])
                                )
                        )
        );

        index += stride;
    }

    cache1[threadIdx.x] = temp;

    __syncthreads();

    // reduction
    size_t i = blockDim.x / 2;
    while (i != 0) {
        if (threadIdx.x < i) {
            cache1[threadIdx.x] = __fadd_rd(cache1[threadIdx.x], cache1[threadIdx.x + i]);
        }
        __syncthreads();
        i /= 2;
    }


    if (threadIdx.x == 0) {
        atomicAdd(result, cache1[0]); //
    }

}
```

Implemented Kernels

euclidean_kernel_math(pr_old, pr_gpu, pSquareError_gpu, pPpr->V)

- Adaptation of dot product kernel
- Square root computed on CPU
- Shared memory to improve parallelism
- Reduction of partial results

```
template<typename T> __global__ void euclidean_kernel_math(const T * __restrict__ x, const T * __restrict__ y, T * __restrict__ result, unsigned int n)
{
    size_t index = threadIdx.x + blockDim.x * blockIdx.x;
    size_t stride = blockDim.x * gridDim.x;

    extern __shared__ T cache1[];

    T temp = 0.0;
    while (index < n) {
        temp =  __fadd_rd(temp, (
                                __fmul_rd(
                                        __fsub_rd(x[index],y[index]),
                                        __fsub_rd(x[index],y[index])
                                )
                        )
        );

        index += stride;
    }

    cache1[threadIdx.x] = temp;

    __syncthreads();

    // reduction
    size_t i = blockDim.x / 2;
    while (i != 0) {
        if (threadIdx.x < i) {
            cache1[threadIdx.x] = __fadd_rd(cache1[threadIdx.x], cache1[threadIdx.x + i]);
        }
        __syncthreads();
        i /= 2;
    }


    if (threadIdx.x == 0) {
        atomicAdd(result, cache1[0]); //
    }

}
```

Performance

Instances	Average	StdDev	Minimum	Maximum	Kernel
900	8,5 ms	0,18 ms	8 ms	9 ms	**__spmv_coo_flat**
900	0,1 ms	0,09 ms	0,1 ms	0,13 ms	**euclidean_kernel_math**
900	0,015 ms	0,0098 ms	0,013 ms	0,018 ms	**dangling_kernel**

- **Block size = 128**

Performance

Instances	Average	StdDev	Minimum	Maximum	Kernel
900	8,5 ms	0,18 ms	8 ms	9 ms	**__spmv_coo_flat**
900	0,1 ms	0,09 ms	0,1 ms	0,13 ms	**euclidean_kernel_math**
900	0,015 ms	0,0098 ms	0,013 ms	0,018 ms	**dangling_kernel**

- **Block size = 128**
- **Grid size k1 = (E + Block size - 1) / (Block size)**
- **Grid size k2 = (V + Block size - 1) / (Block size)**
- **Grid size k3 = (D + Block size - 1) / (Block size) , D = |Dangling Vertexes|**

Performance

Instances	Average	StdDev	Minimum	Maximum	Kernel
900	8,5 ms	0,18 ms	8 ms	9 ms	**__spmv_coo_flat**
900	0,1 ms	0,09 ms	0,1 ms	0,13 ms	**euclidean_kernel_math**
900	0,015 ms	0,0098 ms	0,013 ms	0,018 ms	**dangling_kernel**

- **Block size = 128**
- **Grid size k1 = (E + Block size - 1) / (Block size)**
- **Grid size k2 = (V + Block size - 1) / (Block size)**
- **Grid size k3 = (D + Block size - 1) / (Block size) , D = |Dangling Vertexes|**
- **Mean computation time (100 iterations, threshold 1e-6) = 750 ms**
- **Mean accuracy (100 iterations, threshold 1e-6) = 100%**

Now, the heuristics

Some improvements

Some improvements

- Square error computed every 5 iterations

Some improvements

- Square error computed every 5 iterations
  - Mean execution time: 613 ms
  - Mean accuracy: 100%

Some improvements

- Square error computed every 5 iterations
  - Mean execution time: 613 ms
  - Mean accuracy: 100%

Rationale: faster runs take ~ 10 iterations, slower runs take ~ 30 iterations

Some improvements

- Square error computed every 5 iterations
  - Mean execution time: 613 ms
  - Mean accuracy: 100%

Rationale: faster runs take ~ 10 iterations, slower runs take ~ 30 iterations

Easily adaptable with different threshold

Some improvements

- Only the sum of difference of squares is considered

Some improvements

- Only the sum of difference of squares is considered
  - Mean execution time: 650 ms
  - Mean accuracy: 100%

Some improvements

- Only the sum of difference of squares is considered
  - Mean execution time: 650 ms
  - Mean accuracy: 100%

Rationale: $\sqrt{10^{-12}} = 10^{-6}$

Some improvements

- Only the sum of difference of squares is considered
  - Mean execution time: 650 ms
  - Mean accuracy: 100%

Rationale: $\sqrt{10^{-12}} = 10^{-6}$

Easily adaptable with different threshold

Some improvements

- If the vertex is dangling, the ranking is fixed

Some improvements

- If the vertex is dangling, the ranking is fixed
  - Dangling vertexes of Wikipedia : 2.8%
  - Execution time : ~3ms

Some improvements

- If the vertex is dangling, the ranking is fixed
  - Dangling vertexes of Wikipedia : 2.8%
  - Execution time : ~3ms (100% accuracy)

Wikipedia top19 : {7030, 9008, 24716, 27566, 28020, 28196, 195101, 500469, 577659, 689491, 932394, 1518892, 1702309, 1835017, 2144742, 2257865, 2532493, 2979297, 2984189}

Some improvements

- If the vertex is dangling, the ranking is fixed
  - Dangling vertexes of Wikipedia : 2.8%
  - Execution time : ~3ms (100% accuracy)

Wikipedia top19 : {7030, 9008, 24716, 27566, 28020, 28196, 195101, 500469, 577659, 689491, 932394, 1518892, 1702309, 1835017, 2144742, 2257865, 2532493, 2979297, 2984189}

Rationale: The algorithms adds 1-alpha to the pr of the personalized vertex at each iteration. If the node is dangling the added pr won't directly propagate to other nodes. So the ranking is the same apart for the dangling.

Some improvements

- If the vertex is dangling, the ranking is fixed
  - Dangling vertexes of Wikipedia : 2.8%
  - Execution time : ~3ms (100% accuracy)

Wikipedia top19 : {7030, 9008, 24716, 27566, 28020, 28196, 195101, 500469, 577659, 689491, 932394, 1518892, 1702309, 1835017, 2144742, 2257865, 2532493, 2979297, 2984189}

Rationale: The algorithms adds 1-alpha to the pr of the personalized vertex at each iteration. If the node is dangling the added pr won't directly propagate to other nodes. So the ranking is the same apart for the dangling.

Easily adaptable with different graphs

Final combination

- Pre-computed result on dangling indexes

Final combination

- Pre-computed result on dangling indexes
- Only the sum of difference of squares is considered

Final combination

- Pre-computed result on dangling indexes
- Only the sum of difference of squares is considered
- Custom convergence threshold

Final combination

- Pre-computed result on dangling indexes
- Only the sum of difference of squares is considered
- Custom convergence threshold
  - $c = 6 \cdot 10^{-9}$
  - Not easy to tune, a lot of time is required.
  - Highly dependant on the graph and the real threshold
  - Based on empirical results, theoretical bounds may exist

Final combination

- Pre-computed result on dangling indexes
- Only the sum of difference of squares is considered
- Custom convergence threshold
  - $c = 6 \cdot 10^{-9}$
  - Not easy to tune, a lot of time is required.
  - Highly dependant on the graph and the real threshold
  - Based on empirical results, theoretical bounds may exist
- Skip of sum of difference of squares computation

Final combination

- Pre-computed result on dangling indexes
- Only the sum of difference of squares is considered
- Custom convergence threshold
  - $c = 6 \cdot 10^{-9}$
  - Not easy to tune, a lot of time is required.
  - Highly dependant on the graph and the real threshold
  - Based on empirical results, theoretical bounds may exist
- Skip of sum of difference of squares computation
  - Compute if (iter > 3 and iter < 12) or (iter > 12 and iter % 3 == 0)
  - Adaptable with different threshold, depends on previous heuristic

Final combination

- Pre-computed result on dangling indexes
- Only the sum of difference of squares is considered
- Custom convergence threshold
  - $c = 6 \cdot 10^{-9}$
  - Not easy to tune, a lot of time is required.
  - Highly dependant on the graph and the real threshold
  - Based on empirical results, theoretical bounds may exist
- Skip of sum of difference of squares computation
  - Compute if (iter > 3 and iter < 12) or (iter > 12 and iter % 3 == 0)
  - Adaptable with different threshold, depends on previous heuristic

Mean execution time: 100 ms **Mean accuracy: 95%**

Performance

Implementation	Mean total time (reset + exec)	Mean exec time	Accuracy
Naive	1023 ms	1010.62 ms	100%
Cublas coo, cusparse of dot	258 ms	255.081 ms	100%
Final implementation	103 ms	100 ms	>80%

Tested on rtx 3080 laptop

Performance

```
 Time (%)  Total Time (ns)  Instances    Avg (ns)      Med (ns)     Min (ns)    Max (ns)   StdDev (ns)                                          Name

 --------  ---------------  ---------  ------------  ------------  ----------  ----------  -----------  ----------------------------------------------------------------------------------------
--------
     92.0    4,818,817,005        300  16,062,723.4  16,025,571.5  15,933,219  20,742,032    401,228.2  void cooSPMV_math<int, float>(const T1 *, const T1 *, const T2 *, T2, const T2 *
, T2 *)
      6.0      314,028,668        300   1,046,762.2   1,044,380.0   1,038,236   1,440,922     32,342.9  void euclidean_kernel<float>(T1 *, T1 *, T1 *, unsigned int)

      0.6       29,358,324        300      97,861.1      97,504.0      97,216     128,736      3,080.1  void dot_product_kernel_math<int, float>(const T1 *, const T2 *, T2 *, T2, unsig
ned int)
      0.5       24,811,682        300      82,705.6      82,703.5      77,344      85,312        737.6  void vectorScalarAddAndIncrement_math<float>(T1, T1 *, int, int, T1)

      0.4       23,175,290        300      77,251.0      77,183.0      75,776      79,392        650.6  void copy_vector<float>(T1 *, T1 *, int)

      0.3       13,469,514         10   1,346,951.4   1,332,683.0   1,322,139   1,487,226     49,449.1  void vectorScalarMul<float>(T1, T1 *, int)

      0.2       12,004,502        300      40,015.0      39,968.0      39,296      41,344        324.3  void init_vector<float>(T1 *, int, T1)
```

For all the implementations seen until now the Time (%) were similar.
Can we skip the SPMV multiplication?

Performance

Implementation	Mean total time (reset + exec)	Mean exec time	Accuracy
Naive	1023 ms	1010.62 ms	100%
Cublas coo, cusparse of dot	258 ms	255.081 ms	100%
Final implementation	103 ms	100 ms	>80%
Monte Carlo complete path	17 ms	14.2666 ms	>80%

Tested on rtx 3080 laptop, 100 runs

03
Monte Carlo Implementation

Monte Carlo Complete path

- Simulate m random walks starting from the personalized vertex.
- At each step the walker can terminate with probability 1-alpha.
- At each step the walker move to a random vertex connected to an outgoing edge with probability alpha.

Let N_j be the number of visits by all walkers to vertex j, then the pr of vertex j is defined as following:

$$pr_j(personalized\,vertex, alpha) = (1 - alpha)\,\frac{1}{m} N_j\,(personalized\,vertex)$$

https://link.springer.com/content/pdf/10.1007/978-3-642-21286-4.pdf page 50

Monte Carlo Complete path

Since we don't care about the values we can skip the multiplication and division

$$pr_j(personalized\,vertex, alpha) = (1 - alpha)\,\frac{1}{m} N_j\,(personalized\,vertex)$$

$$\Downarrow$$

$$pr_j(personalized\,vertex, alpha) = N_j\,(personalized\,vertex)$$

MC Complete path on CPU

- Uses CSC (Compressed sparse column matrix format)
- Walkers run sequentially.

```
int walkLen = 0;
for (int run = 0; run < nWalkers; run++) {
    int vIdx = s;
    while (walkLen < maxWalkLen){
        pPpr->pr[vIdx]+=1.0f;
        bool terminate = dist(mt) < 1.0f-pPpr->alpha;//Terminate with probability 1-c
        if (terminate) {
            break;
        } else {
            int neighStartIdx = csc.xPtr_cpu[vIdx];
            int neighEndIdx = csc.xPtr_cpu[vIdx+1];
            int neighSize = neighEndIdx - neighStartIdx;
            if (neighSize == 0) {
                break;}

            //select a random connected vertex
            int rIdx = vJump(mt)%neighSize;
            vIdx=csc.x_cpu[neighStartIdx+rIdx];
        }
        walkLen++;
    }
    //std::cout << "Walk length: " << walkLen << std::endl;
    walkLen = 0;
}
```

MC Complete path on GPU naive

- 1 thread = 1 walker
- If we try to run the walkers in parallel we need an atomic add and in the first step each walker thread will add to the same position

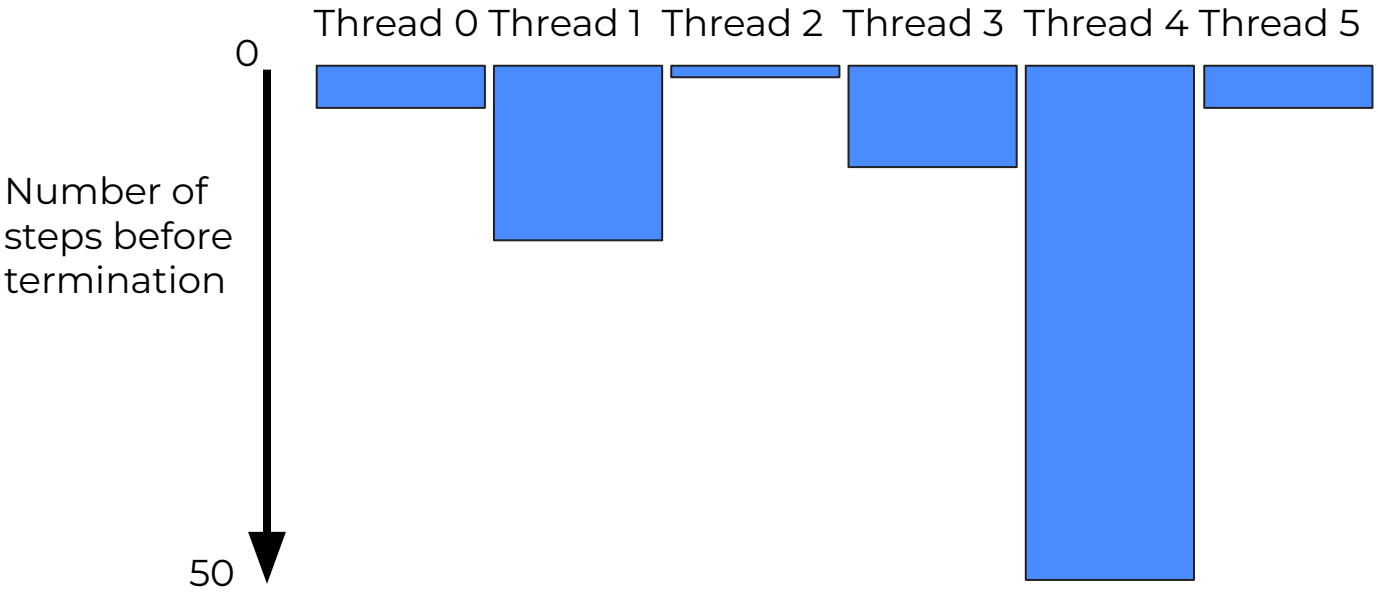
```
const unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
if (i < walkers) {
    int vIdx = s;
    while (walkLen < maxWalkLen){
        atomicAdd( address: &pr[vIdx],  val: 1.0f);//Add 1 to each visited node
        bool terminate = dist( &: mt) < 1.0f-pPpr->alpha;//Terminate with probability 1-c
        if (terminate) {
            break;
        } else {
            int neighStartIdx = csc.xPtr_cpu[vIdx];
            int neighEndIdx = csc.xPtr_cpu[vIdx+1];
            int neighSize = neighEndIdx - neighStartIdx;
            if (neighSize == 0) {
                break;}

            //select a random connected vertex
            int rIdx = vJump( &: mt)%neighSize;
            vIdx=csc.x_cpu[neighStartIdx+rIdx];
        }
        walkLen++;
    }
    walkLen = 0;
}
```

MC Complete path on GPU naive

- Solution:
  Each thread starts from a random vertex between the outgoing edges.
  In this way the atomic adds act on different memory locations

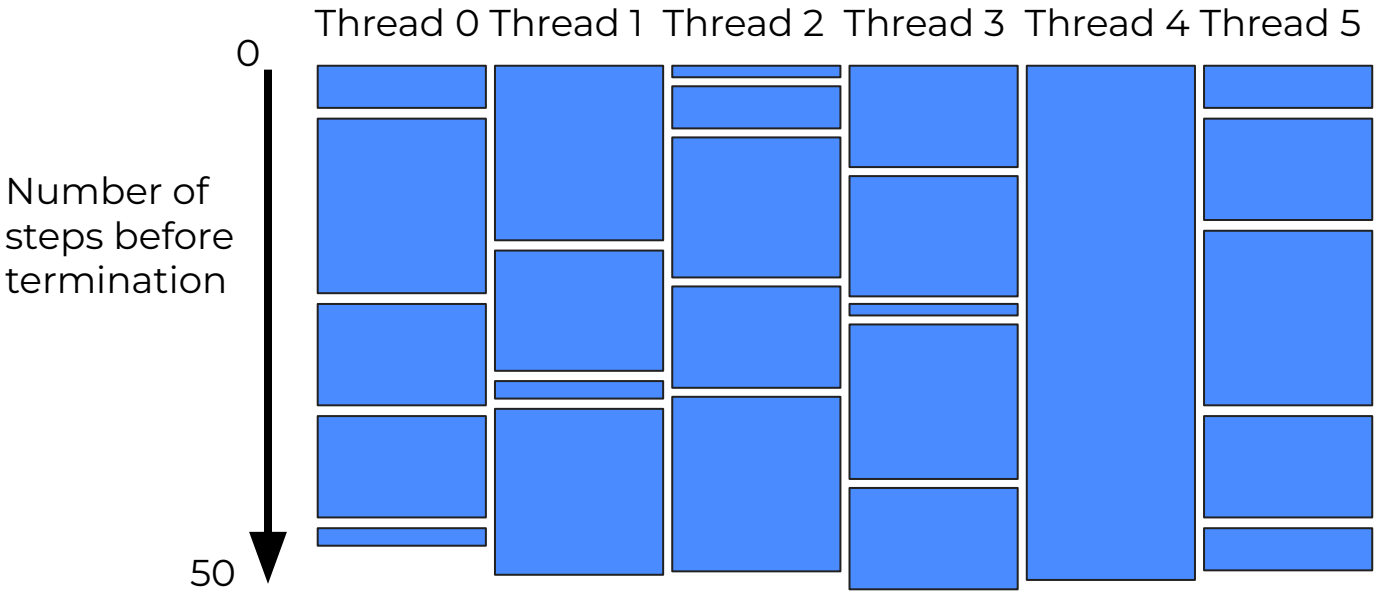
```
//Starting from neighbors of s
unsigned int rIdx0 = curand( state: &localState) % neighSizee[s];
int vIdx =  x[neighStartPos + rIdx0];
int cumulativeWalks = 1;
```

Thread utilization



Most threads do only a few steps and then terminate the random walk

Thread utilization solution



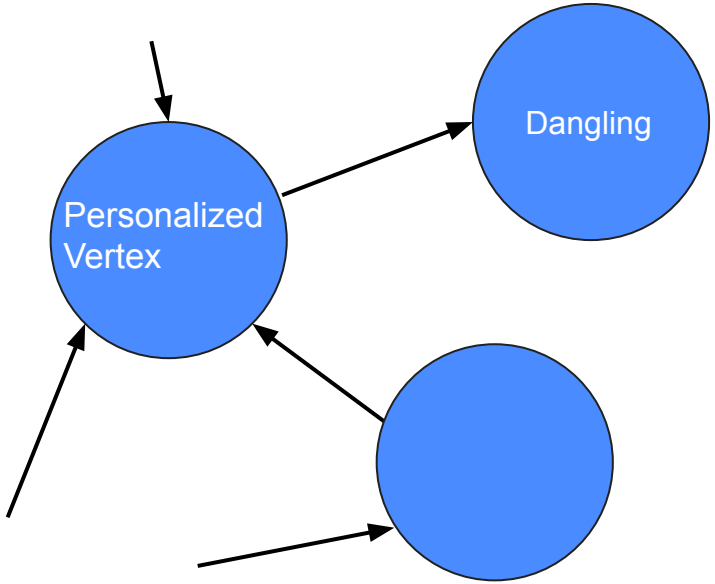
Use a thread that has terminated the computation as a new walker

```
if (cumulativeWalks<spawnWalkerThreshold)
    vIdx=s;
else
    break;
```

Leads to a much higher accuracy due to the larger number of samples

Dangling nodes in MC

We have to consider an additional case




```
error in value! (0) correct=1288741 (val=0.15), found=1288741 (val=7.68e+05)
error in value! (1) correct=975307 (val=0.1275), found=975307 (val=6.53e+05)
error in rank! (2) correct=1702309 (val=0.001989), found=3566906 (val=2.804e-07)
error in rank! (3) correct=500469 (val=0.0009716), found=3566905 (val=2.804e-07)
error in rank! (4) correct=1518892 (val=0.0009589), found=3566904 (val=2.804e-07)
error in rank! (5) correct=24716 (val=0.0008311), found=3566903 (val=2.804e-07)
error in rank! (6) correct=689491 (val=0.0008219), found=3566902 (val=2.804e-07)
error in rank! (7) correct=195101 (val=0.0007462), found=3566901 (val=2.804e-07)
error in rank! (8) correct=932394 (val=0.00071), found=3566900 (val=2.804e-07)
error in rank! (9) correct=2984189 (val=0.0006991), found=3566899 (val=2.804e-07)
error in rank! (10) correct=1835017 (val=0.0006721), found=3566898 (val=2.804e-07)
error in rank! (11) correct=577659 (val=0.0006673), found=3566897 (val=2.804e-07)
error in rank! (12) correct=28196 (val=0.0006478), found=3566896 (val=2.804e-07)
error in rank! (13) correct=7030 (val=0.0006427), found=3566895 (val=2.804e-07)
error in rank! (14) correct=2257865 (val=0.0006064), found=3566894 (val=2.804e-07)
error in rank! (15) correct=2532493 (val=0.0005939), found=3566893 (val=2.804e-07)
error in rank! (16) correct=28020 (val=0.0005909), found=3566892 (val=2.804e-07)
error in rank! (17) correct=2979297 (val=0.0005655), found=3566891 (val=2.804e-07)
error in rank! (18) correct=9008 (val=0.0004983), found=3566890 (val=2.804e-07)
error in rank! (19) correct=27566 (val=0.0004868), found=3566889 (val=2.804e-07)
```

Dangling nodes, MC

Correct first two places

Same ranking as dangling but shifted by one towards the bottom

```
error in value (0) correct=1288741 (val=0.15), found=1288741 (val=7.68e+05)
error in value (1) correct=975307 (val=0.1275), found=975307 (val=6.53e+05)
error in rank! (2) correct=1702309 (val=0.001989),
error in rank! (3) correct=500469 (val=0.0009716),
error in rank! (4) correct=1518892 (val=0.0009589),
error in rank! (5) correct=24716 (val=0.0008311), f
error in rank! (6) correct=689491 (val=0.0008219),
error in rank! (7) correct=195101 (val=0.0007462),
error in rank! (8) correct=932394 (val=0.00071), fo
error in rank! (9) correct=2984189 (val=0.0006991),
error in rank! (10) correct=1835017 (val=0.0006721)
error in rank! (11) correct=577659 (val=0.0006673),
error in rank! (12) correct=28196 (val=0.0006478),
error in rank! (13) correct=7030 (val=0.0006427), f
error in rank! (14) correct=2257865 (val=0.0006064)
error in rank! (15) correct=2532493 (val=0.0005939)
error in rank! (16) correct=28020 (val=0.0005909),
error in rank! (17) correct=2979297 (val=0.0005655)
error in rank! (18) correct=9008 (val=0.0004983), f
error in rank! (19) correct=27566 (val=0.0004868),
```

Initial pr values

Solution: Init pr with values for dangling

100% accuracy for vertices with outgoing edges towards dangling

100% accuracy and
2ms reset+execution time for danglings

Other MC optimizations

- Copy curand state in local memory
- Use an array to store outgoing edges size

Q&A

Additional slides

Performance of every implementation

Implementation	Mean total time (reset + exec)	Mean exec time	Accuracy
Naive	1023 ms	1010.62 ms	100%
Cublas bcr test	1304 ms	1215.45 ms	100%
Improved naive	814 ms	800.903 ms	100%
Cublas coo, cusparse of dot	258 ms	255.081 ms	100%
Final implementation	103 ms	100 ms	>80%
Monte Carlo complete path	17 ms	14.2666 ms	>80%

Tested on rtx 3080 laptop, 100 runs