Hardware/Software Vectorization for Closeness Centrality on Multi-/Many-Core Architectures

Ahmet Erdem Sarıyüce, **Erik Saule**, Kamer Kaya, Ümit V. Çatalyürek

The Ohio State University (BMI, CS, ECE)
University of North Carolina at Charlotte (CS)

MTAAP 2014

Outline

Centralities - Concept

Answer questions such as

- Who controls the flow in a network?
- Who is more important?
- Who has more influence?
- Whose contribution is significant for connections?

Different kinds of graph

- road networks
- social networks
- power grids
- mechanical mesh

Applications

- Covert network (e.g., terrorist identification)
- Contingency analysis (e.g., weakness/robustness of networks)
- Viral marketing (e.g., who will spread the word best)
- Traffic analysis
- Store locations

Closeness Centrality

Definition

Let $G = (V, E)$ be an unweighted graph with the vertex set V and edge set E.
$cc[v] = \sum_{u \in V} \frac{1}{d(v,u)}$ where $d(u, v)$ is the shortest path length between u and v.

The best known algorithm computes the shortest path graph rooted in each vertex of the graph. The complexity is $O(E)$ per source, $O(VE)$ in total, which makes its computationally expensive.

Closeness Centrality

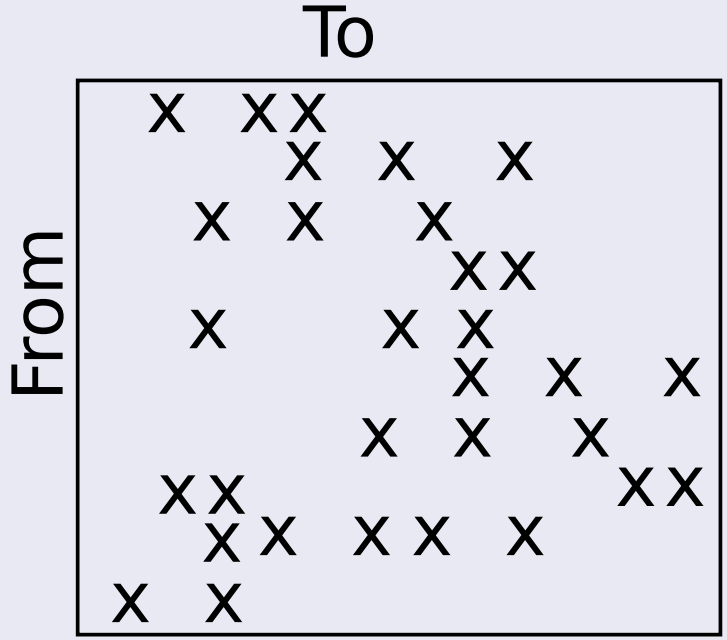
Definition

Let $G = (V, E)$ be an unweighted graph with the vertex set V and edge set E.
$cc[v] = \sum_{u \in V} \frac{1}{d(v,u)}$ where $d(u, v)$ is the shortest path length between u and v.

The best known algorithm computes the shortest path graph rooted in each vertex of the graph. The complexity is $O(E)$ per source, $O(VE)$ in total, which makes its computationally expensive.

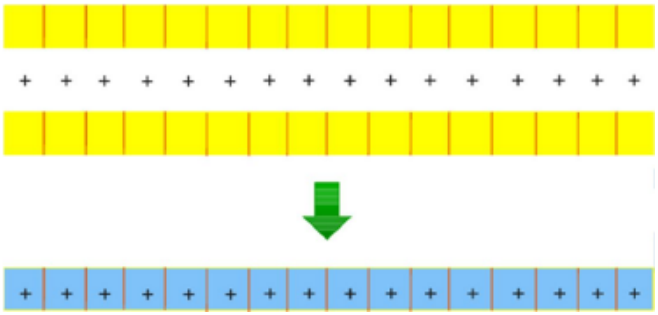
Typical Algorithms (one BFS per source)

Top-down or Bottom-Up.



Direction Optimizing.
Level synchronous bfs.
No regularity in the computation: no use of vector processing units.

Vector processing units



Operations

- add
- mul
- or
- and
- ...

SIMD: a key source of performance

- MMX (1996): 64 bit registers (x86)
- SSE (1999): 128 bit registers (x86)
- AVX (2008): 256 bit registers (x86)
- IMIC (2012): 512-bit registers (Xeon Phi)
- 512-bits register to come on x86

Ignoring vectorization is wasting 75% (SSE), 87% (AVX), 93% (MIC) of available performance in single precision.

Also it is often necessary to saturate memory bandwidth.

Outline

An SpMV-based approach

A simpler definition of level synchronous BFS

Vertex v is at level ℓ if and only if one of the neighbors of v is at level $\ell - 1$ and v is not at any level $\ell' < \ell$.

Let $x_i^\ell =$ **true** if vertex i is a part of the frontier at level ℓ.

$y^{\ell+1}$ is the neighbors of level ℓ. $y_k^{\ell+1} = OR_{j \in \Gamma(k)} x_j^\ell$. ((OR, AND)-SpMV)

Compute the next level frontier $x_i^{\ell+1} = y_i^{\ell+1} \& \neg (OR_{\ell' \leq \ell} x_i^{\ell'})$.

Contribution of the source to $cc[i]$ is $\frac{x_i^\ell}{\ell}$.

top-down (scatter writes)

For each element of the frontier, touch the neighbors.

Complexity: $O(E)$

Writes are scattered in memory

Read are linear

bottom-up (gather reads)

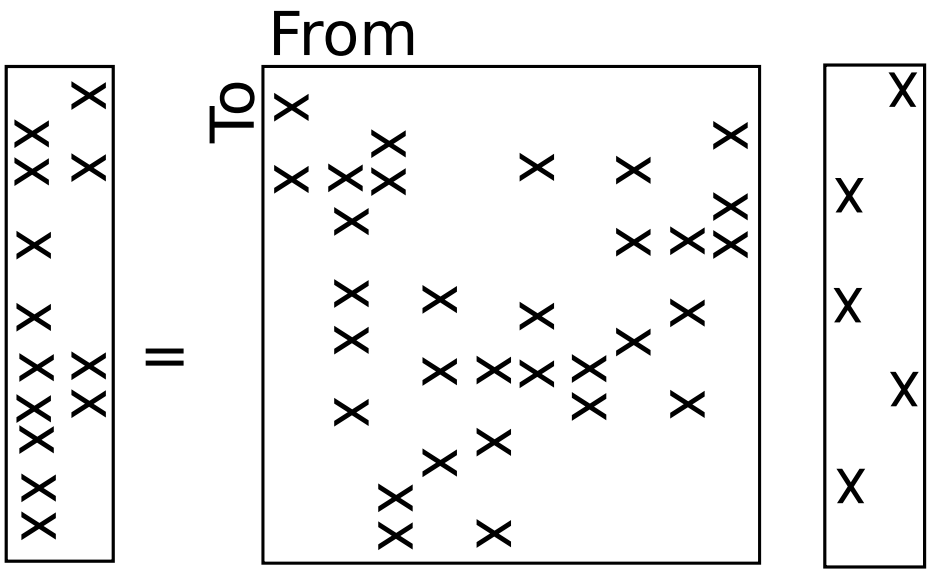
For each vertex, are the neighbors in the frontier?

Complexity $O(ED)$, where D is the diameter of the graph.

Writes are performed once linearly.

Reads are (hopefully) close-by.

From SpMV to SpMM



Data: $G = (V, E)$, b
Output: cc[.]
▷Init
$\mathrm{cc}[v] \leftarrow 0, \forall v \in V$
$\ell \leftarrow 0$
partition V into k batches $\Pi = \{V_1, V_2, \ldots, V_k\}$ of size b
for each batch *of vertices* $V_p \in \Pi$ **do**
  $x^0_{s,s} \leftarrow 1$ if $s \in V_p$, 0 otherwise
  while $\sum_i \sum_s x^{\ell}_{i,s} > 0$ **do**
    ▷SpMM
    $y^{\ell+1}_{i,s} = OR_{j \in \Gamma(i)} x^{\ell}_{j,s}, \forall s, \forall i$
    ▷Update
    $x^{\ell+1}_{i,s} = y^{\ell+1}_{i,s} \& \neg(OR_{\ell' \leq \ell} x^{\ell'}_{i,s}), \forall s, \forall i$
    $\ell \leftarrow \ell + 1$
    for all $v \in V$ **do**
      $\mathrm{cc}[v] \leftarrow \mathrm{cc}[v] + \frac{\sum_s x^{\ell}_{v,s}}{\ell}$
return cc[.]

Some simple analysis

- Complexity of $O(VED)$
  - Instead of $O(VE)$
  - But D is typically small
- Vectorizable
- The matrix is transferred $\frac{VD}{b}$ times
  - Instead of V
  - D is small and b can be big (512-bit registers on MIC)
- Increasing b increases the size of the right hand side
  - Potentially trash the cache
  - Regularize the memory access patterns

Vectorization

```
void cc_cpu_256_spmm (int* xadj, int* adj, int n, float* cc)
{
  int b = 256;
  size_t size_alloc = n * b / 8;
  char* neighbor = (char*)_mm_malloc(size_alloc, 32);
  char* current = (char*)_mm_malloc(size_alloc, 32);
  char* visited = (char*)_mm_malloc(size_alloc, 32);
  for (int s = 0; s < n; s += b) {
    //Init
#pragma omp parallel for schedule (dynamic, CC_CHUNK)
    for (int i = 0; i < n; ++i) {
      __m256i neigh = _mm256_setzero_si256();
      int il[8] = {0, 0, 0, 0, 0, 0, 0, 0};
      if (i >= s && i < s + b)
        il[(i-s)>>5] = 1 << ((i-s) & 0x1F);
      __m256i cu = _mm256_set_epi32(il[7], il[6], il[5], il[4],
                                    il[3], il[2], il[1], il[0]);
      _mm256_store_si256 ((__m256i *)(neighbor + 32 * i), neigh);
      _mm256_store_si256 ((__m256i *)(current + 32 * i), cu);
      _mm256_store_si256 ((__m256i *)(visited + 32 * i), cu);
    }
    int cont = 1;
    int level = 0;
    while (cont != 0) {
      cont = 0;
      level++;
      //SpMM
#pragma omp parallel for schedule (dynamic, CC_CHUNK)
      for (int i = 0; i < n; ++i) {
        __m256 vali = _mm256_setzero_ps();
        for (int j = xadj[i]; j<xadj[i+1]; ++j) {
          int v = adj[j];
          __m256 state_v = _mm256_load_ps((float*)(current + 32 * v));
          vali = _mm256_or_ps (vali, state_v);
        }
        _mm256_store_ps ((float*)(neighbor + 32 * i), vali);
      }
      //Update
      float flevel = 1.0f / (float) level;
#pragma omp parallel for schedule (dynamic, CC_CHUNK)
      for (int i = 0; i < n; ++i) {
        __m256 nei = _mm256_load_ps ((float *)(neighbor + 32 * i));
        __m256 vis = _mm256_load_ps ((float *)(visited + 32 * i));
        __m256 cu = _mm256_andnot_ps (vis, nei);
        vis = _mm256_or_ps (nei, vis);
        int bcnt = bitCount_256(cu);
        if (bcnt > 0) {
          cc[i] += bcnt * flevel;
          cont = 1;
        }
        _mm256_store_ps ((float *)(visited + 32 * i), vis);
        _mm256_store_ps ((float *)(current + 32 * i), cu);
      }
    }
  }
  _mm_free(neighbor);
  _mm_free(current);
  _mm_free(visited);
}
```

Vectorization

```
void cc_cpu_256_spmm (int* xadj, int* adj, int n, float* cc)
{
  int b = 256;
  size_t size_alloc = n * b / 8;
  char* neighbor = (char*)_mm_malloc(size_alloc, 32);
  char* current = (char*)_mm_malloc(size_alloc, 32);
  char* visited = (char*)_mm_malloc(size_alloc, 32);
  for (int s = 0; s < n; s += b) {
    //Init
#pragma omp parallel for schedule (dynamic, CC_CHUNK)
    for (int i = 0; i < n; ++i) {
      __m256i neigh = _mm256_setzero_si256();
      int il[8] = {0, 0, 0, 0, 0, 0, 0, 0};
      if (i >= s && i < s + b)
        il[(i-s)>>5] = 1 << ((i-s) & 0x1F);
      __m256i cu = _mm256_set_epi32(il[7], il[6], il[5], il[4],
                                    il[3], il[2], il[1], il[0]);
      _mm256_store_si256 ((__m256i *)(neighbor + 32 * i), neigh);
      _mm256_store_si256 ((__m256i *)(current + 32 * i), cu);
      _mm256_store_si256 ((__m256i *)(visited + 32 * i), cu);
    }
    int cont = 1;
    int level = 0;
    while (cont != 0) {
      cont = 0;
      level++;
      //SpMM
#pragma omp parallel for schedule (dynamic, CC_CHUNK)
      for (int i = 0; i < n; ++i) {
        __m256 vali = _mm256_setzero_ps();
        for (int j = xadj[i]; j<xadj[i+1]; ++j) {
          int v = adj[j];
          __m256 state_v = _mm256_load_ps((float*)(current + 32 * v));
          vali = _mm256_or_ps (vali, state_v);
        }
        _mm256_store_ps ((float*)(neighbor + 32 * i), vali);
      }
      //Update
      float flevel = 1.0f / (float) level;
#pragma omp parallel for schedule (dynamic, CC_CHUNK)
      for (int i = 0; i < n; ++i) {
        __m256 nei = _mm256_load_ps ((float *)(neighbor + 32 * i));
        __m256 vis = _mm256_load_ps ((float *)(visited + 32 * i));
        __m256 cu = _mm256_andnot_ps (vis, nei);
        vis = _mm256_or_ps (nei, vis);
        int bcnt = bitCount_256(cu);
        if (bcnt > 0) {
          cc[i] += bcnt * flevel;
          cont = 1;
        }
        _mm256_store_ps ((float *)(visited + 32 * i), vis);
        _mm256_store_ps ((float *)(current + 32 * i), cu);
      }
    }
  }
  _mm_free(neighbor);
  _mm_free(current);
  _mm_free(visited);
}
```

Variants

Similar SSE, MIC implementations. Also implemented in a generic way in C++ using various tags to inform the compiler of what it can do (restrict, unroll) and using templates to fix the number of BFS to generate dedicated assembly code for each variant.

Software vectorization

Observation

Performing multiple BFS at once does not only allow to utilize vector registers. It also reduces the number of times the graph is traversed

Idea

Why limit the number of concurrent sources to the size of the vector register?

We use the compiler vectorized code to generate kernels for different number of concurrent BFS. We call this technique software vectorization.

Outline

Experimental Setting

Instances

Graph	$\|V\|$	$\|E\|$	$Avg\|\Gamma(v)\|$	$Max\|\Gamma(v)\|$	Diam.
Amazon	403K	4,886K	12.1	2,752	19
Gowalla	196K	1,900K	9.6	14,730	12
Google	855K	8,582K	10.0	6,332	18
NotreDame	325K	2,180K	6.6	10,721	27
WikiTalk	2,388K	9,313K	3.8	100,029	10
Orkut	3,072K	234,370K	76.2	33,313	9
LiveJournal	4,843K	85,691K	17.6	20,333	15

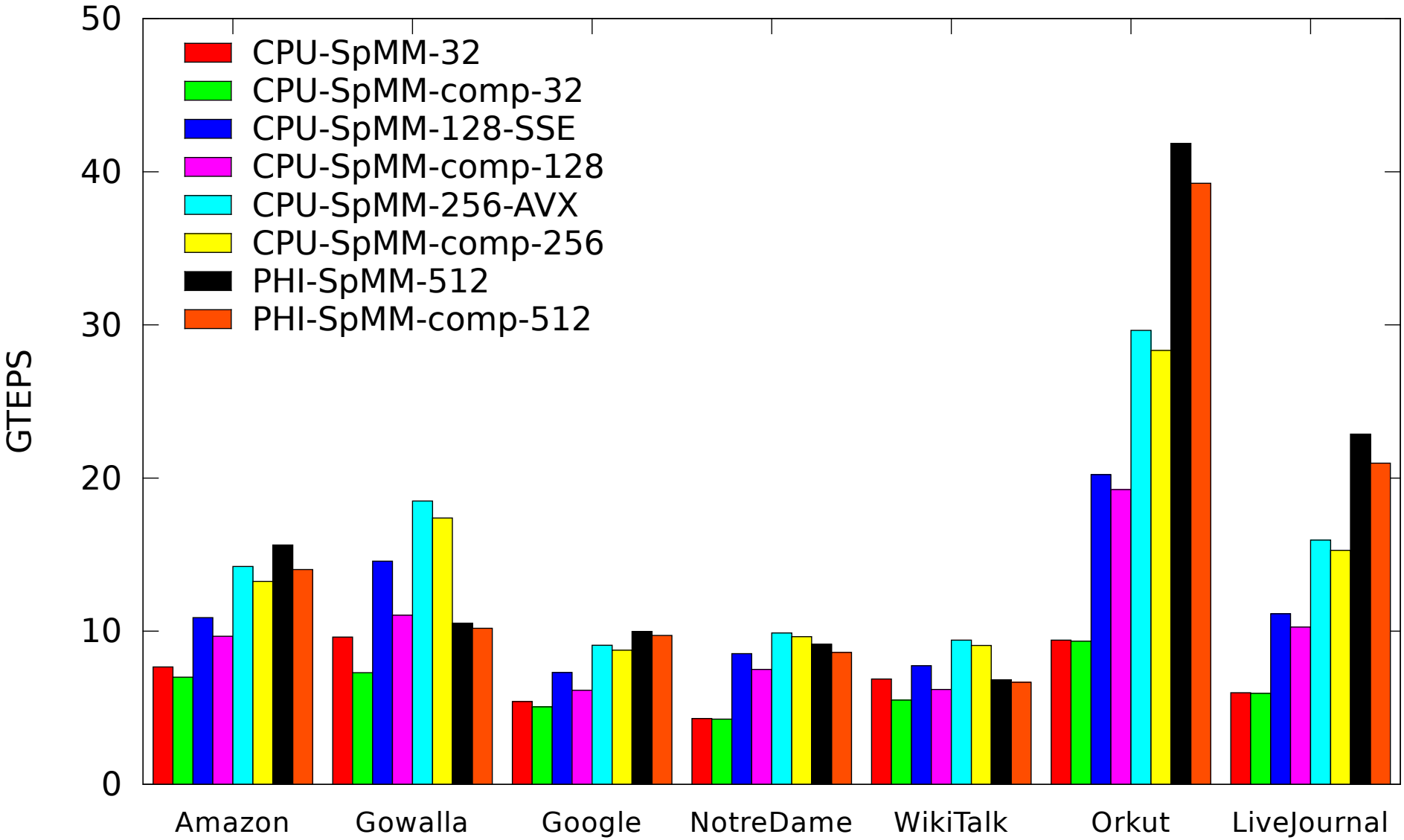
Machines

- Two eight-core Sandybridge EP CPU clocked at 2Ghz. (SSE, AVX)
- One Intel Xeon Phi with 61 cores clocked at 1.05Ghz. (IMIC)

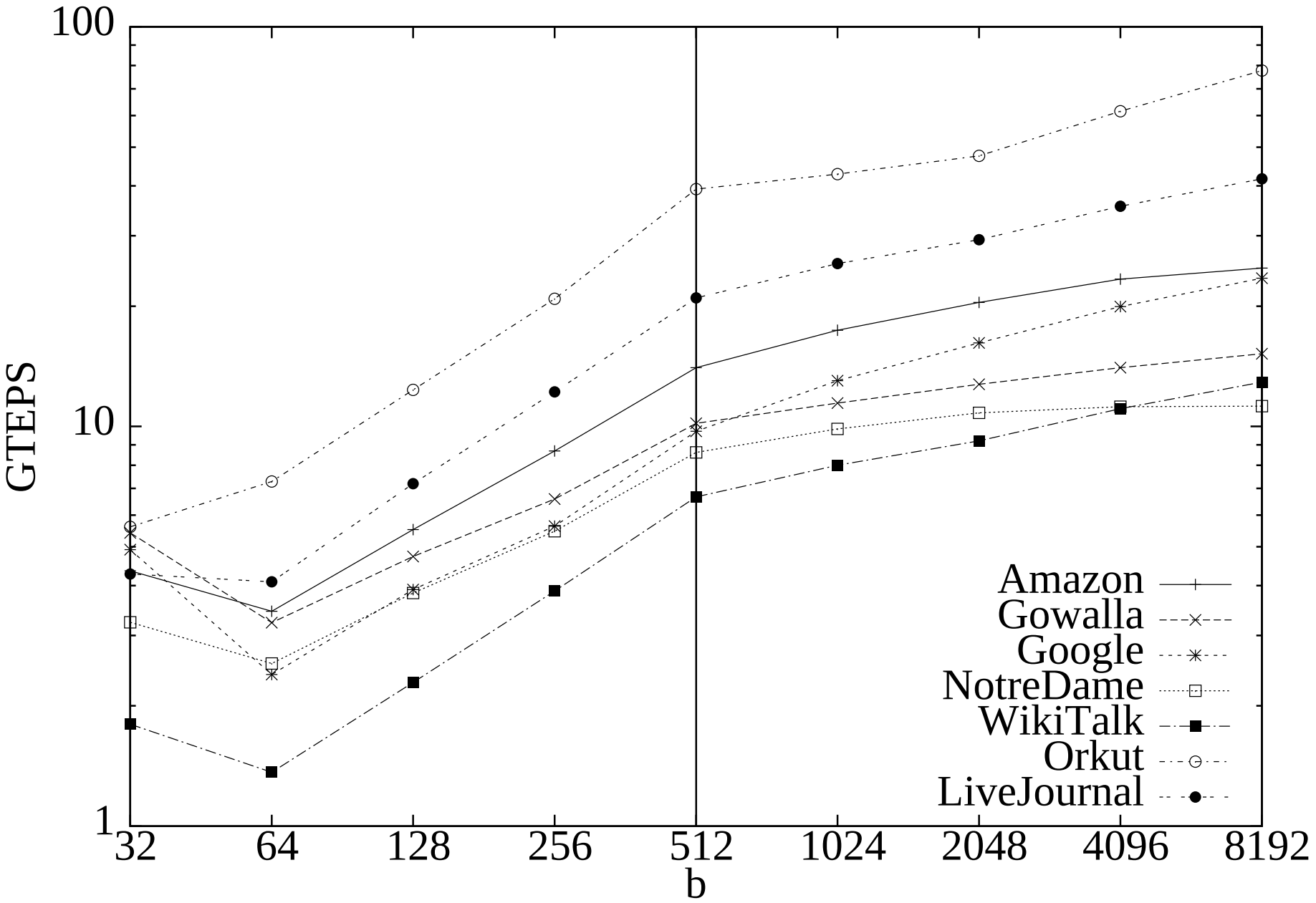
Metric

Traversed Edge Per Second: $\frac{VE}{time}$.

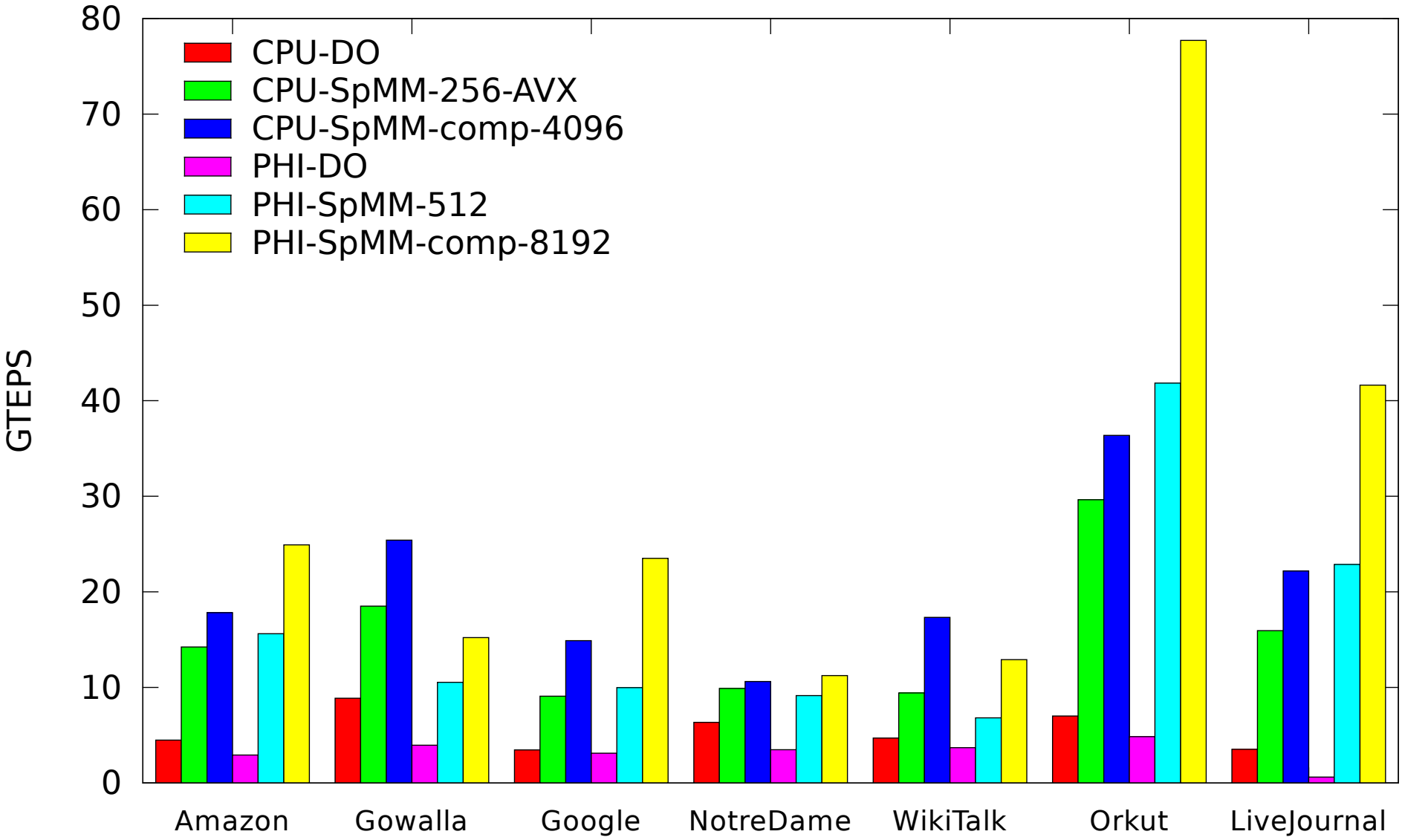
Compiler vectorized is just as good as manually vectorized



Impact of the number of BFS (Intel Xeon Phi)

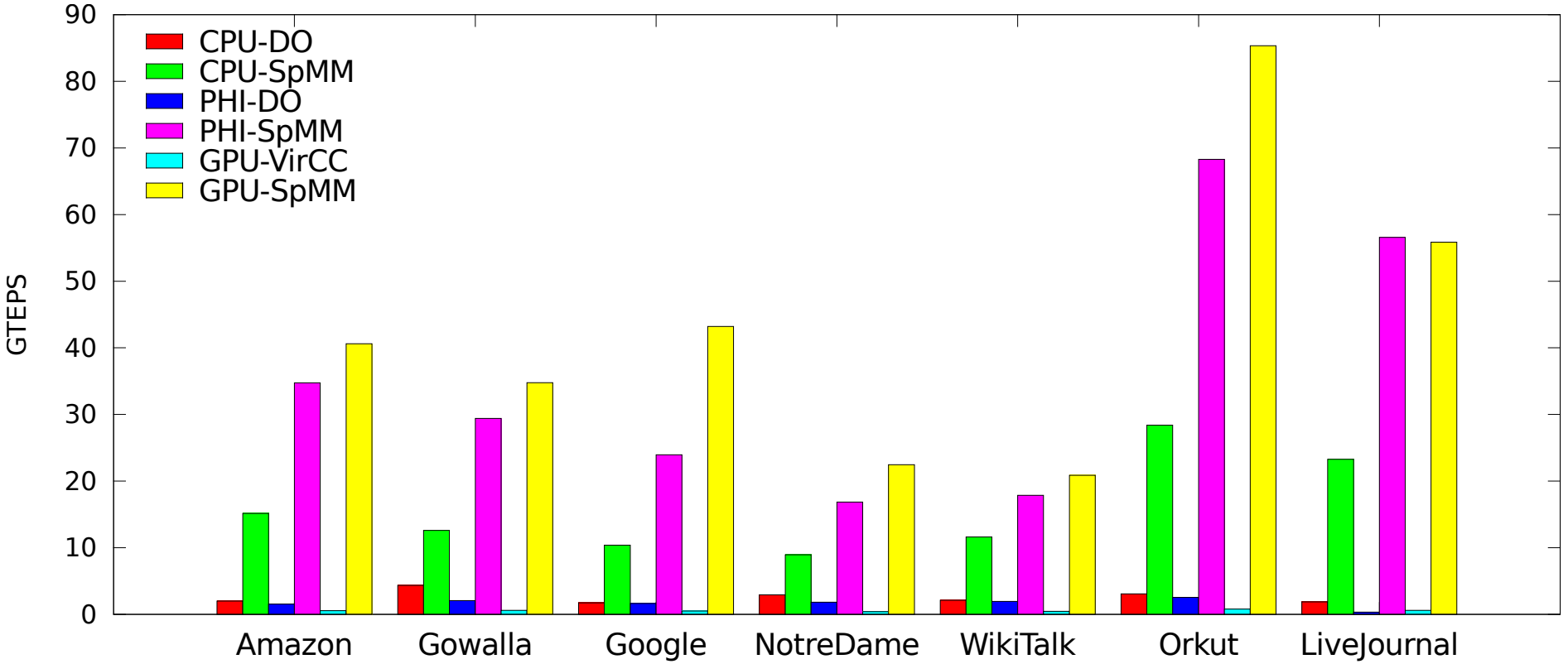


Hardware/Software vectorization is the way to go

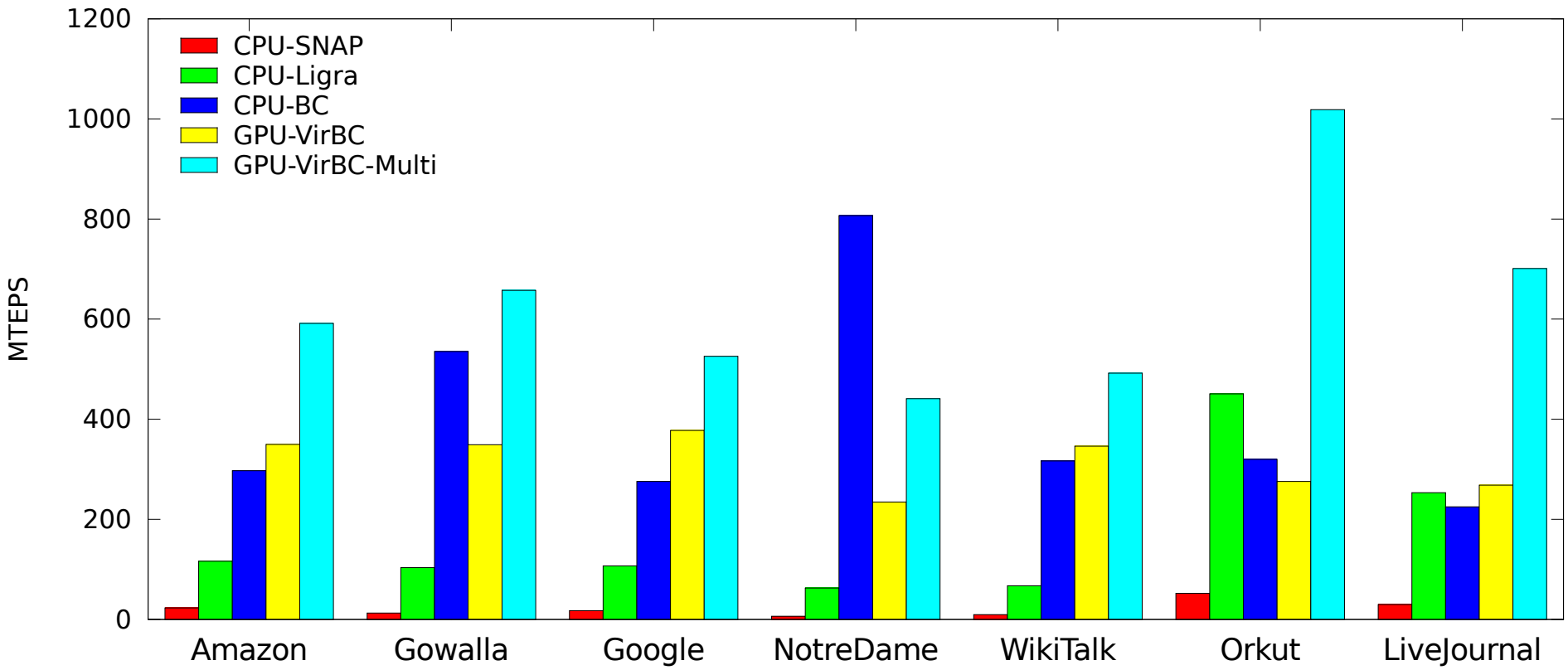


Outline

Work of the future past - Work skipping for CC and GPU



Work of the future past - Multiple sources for BC GPU



Conclusion

- Top-down (or direction-optimized) BFS is work efficient in $O(E)$ but does not easily leverage vectorization

Conclusion

- Top-down (or direction-optimized) BFS is work efficient in $O(E)$ but does not easily leverage vectorization
- BFS can be written as a $O(ED)$ bottom-up algorithm using $O(D)$ SpMV between the adjacency matrix and a bit vector

Conclusion

- Top-down (or direction-optimized) BFS is work efficient in $O(E)$ but does not easily leverage vectorization
- BFS can be written as a $O(ED)$ bottom-up algorithm using $O(D)$ SpMV between the adjacency matrix and a bit vector
- Adding right side vectors allows to do multi-source BFS which vectorizes well and reduces the number of matrix movement

Conclusion

- Top-down (or direction-optimized) BFS is work efficient in $O(E)$ but does not easily leverage vectorization
- BFS can be written as a $O(ED)$ bottom-up algorithm using $O(D)$ SpMV between the adjacency matrix and a bit vector
- Adding right side vectors allows to do multi-source BFS which vectorizes well and reduces the number of matrix movement
- This algorithm can be written to let the compiler do the tedious vectorization using pragmas and C++ templates

Conclusion

- Top-down (or direction-optimized) BFS is work efficient in $O(E)$ but does not easily leverage vectorization
- BFS can be written as a $O(ED)$ bottom-up algorithm using $O(D)$ SpMV between the adjacency matrix and a bit vector
- Adding right side vectors allows to do multi-source BFS which vectorizes well and reduces the number of matrix movement
- This algorithm can be written to let the compiler do the tedious vectorization using pragmas and C++ templates
- Performing as many concurrent BFS as the size of the vector register (Hardware vectorization) provides significant improvement.

Conclusion

- Top-down (or direction-optimized) BFS is work efficient in $O(E)$ but does not easily leverage vectorization
- BFS can be written as a $O(ED)$ bottom-up algorithm using $O(D)$ SpMV between the adjacency matrix and a bit vector
- Adding right side vectors allows to do multi-source BFS which vectorizes well and reduces the number of matrix movement
- This algorithm can be written to let the compiler do the tedious vectorization using pragmas and C++ templates
- Performing as many concurrent BFS as the size of the vector register (Hardware vectorization) provides significant improvement.
- But even more can be achieved using even more concurrent sources (Software vectorization)

Conclusion

- Top-down (or direction-optimized) BFS is work efficient in $O(E)$ but does not easily leverage vectorization
- BFS can be written as a $O(ED)$ bottom-up algorithm using $O(D)$ SpMV between the adjacency matrix and a bit vector
- Adding right side vectors allows to do multi-source BFS which vectorizes well and reduces the number of matrix movement
- This algorithm can be written to let the compiler do the tedious vectorization using pragmas and C++ templates
- Performing as many concurrent BFS as the size of the vector register (Hardware vectorization) provides significant improvement.
- But even more can be achieved using even more concurrent sources (Software vectorization)
- Improves best tested implementation by a factor of 6 on CPU, 20 on Xeon Phi and 70 on GPU

Thank you

Other centrality works (with Sarıyüce, Kaya and Çatalyürek)

- Compression using graph properties (SDM 2013)
- GPU optimization (GPGPU 2013)
- Incremental algorithm (BigData 2013)
- Distributed memory incremental framework (Cluster 2013)

More information

Contact : esaule@uncc.edu

Visit: `http://webpages.uncc.edu/~esaule`