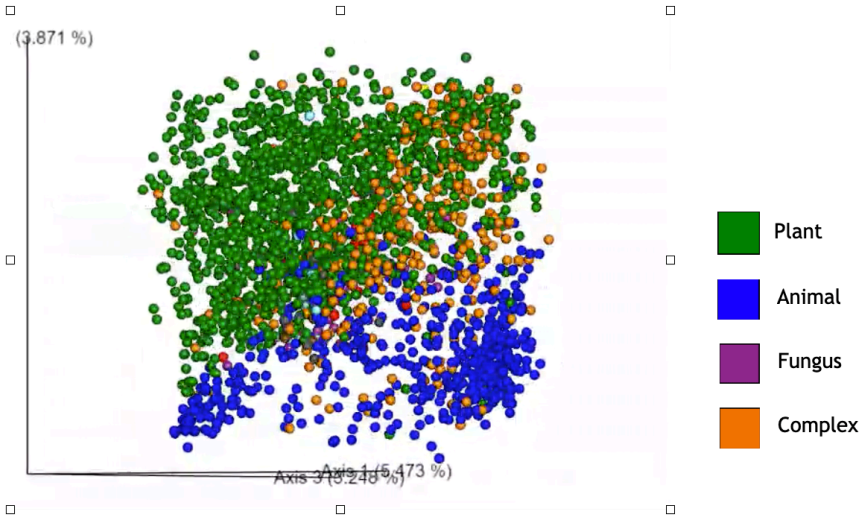
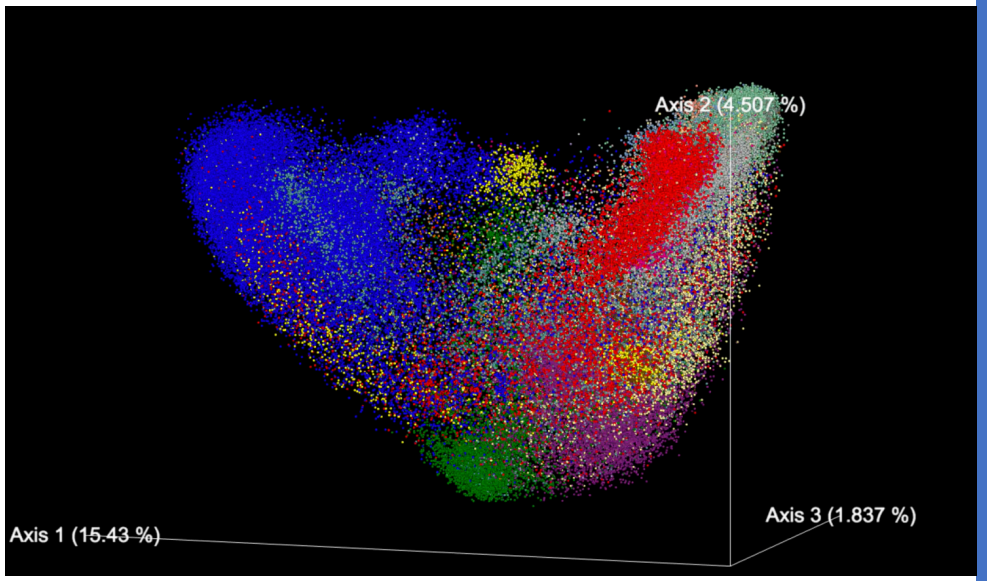




Accelerating microbiome research with OpenACC

Igor Sfiligoi – University of California San Diego

in collaboration with
Daniel McDonald and Rob Knight

KNIGHTLAB

UC San Diego

SDSC SAN DIEGO SUPERCOMPUTER CENTER

PRP PACIFIC RESEARCH PLATFORM

NIH

NSF

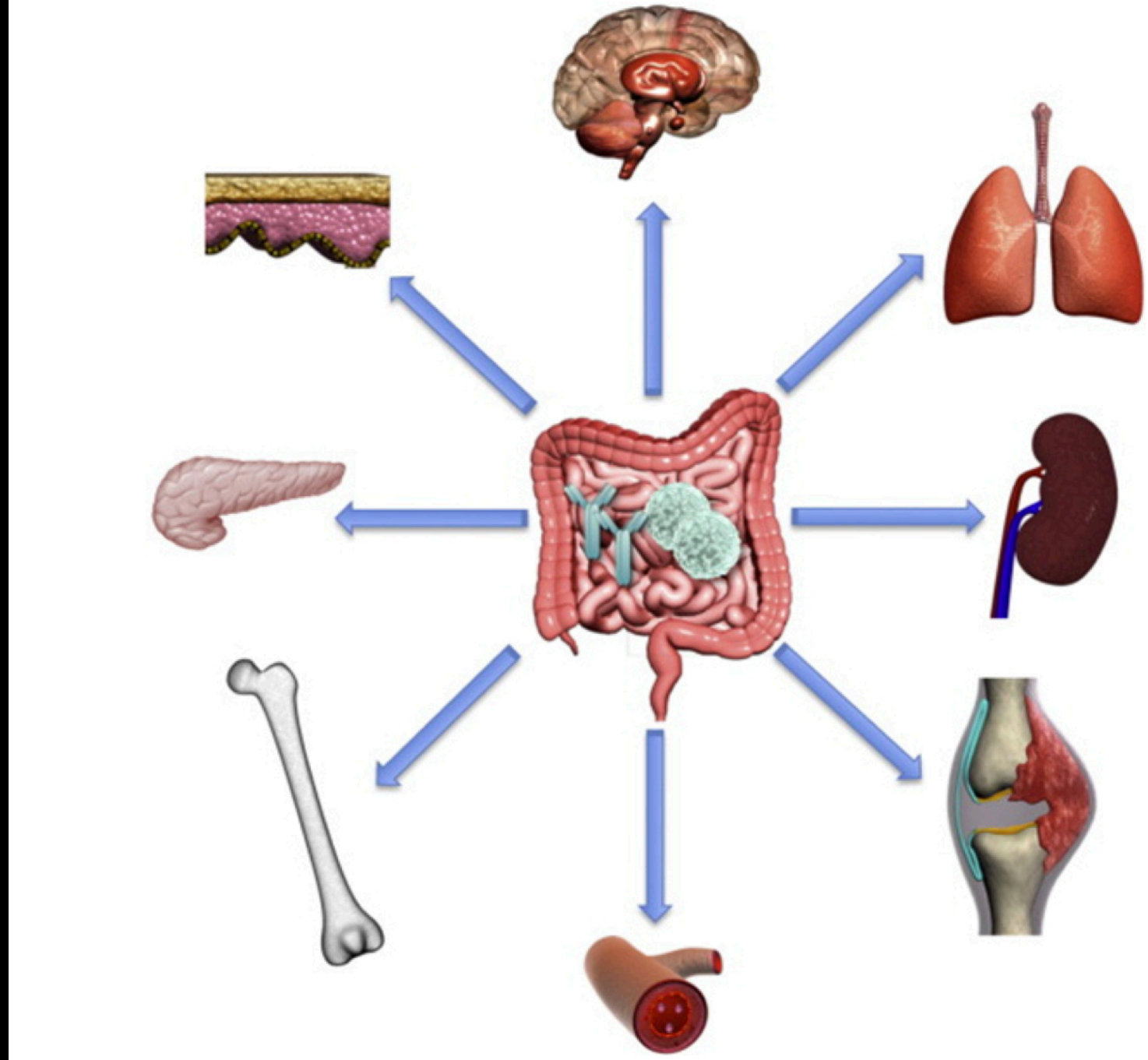
Microbiology connects the human body to the planet

- FreeLiving (239, 49.90%)
- VertGut (133, 27.77%)
- HumGut (37, 7.72%)
- TermGut (23, 4.80%)
- NonsalInvert (22, 4.59%)
- SalInvert (8, 1.67%)
- HumSkin (6, 1.25%)
- HumVulva (4, 0.84%)
- HumMouth (3, 0.63%)
- Plant (2, 0.42%)
- HumVagina (1, 0.21%)
- HumEar (1, 0.21%)

We are what we eat

Studies demonstrated clear link between

- Gut microbiome
- General human health



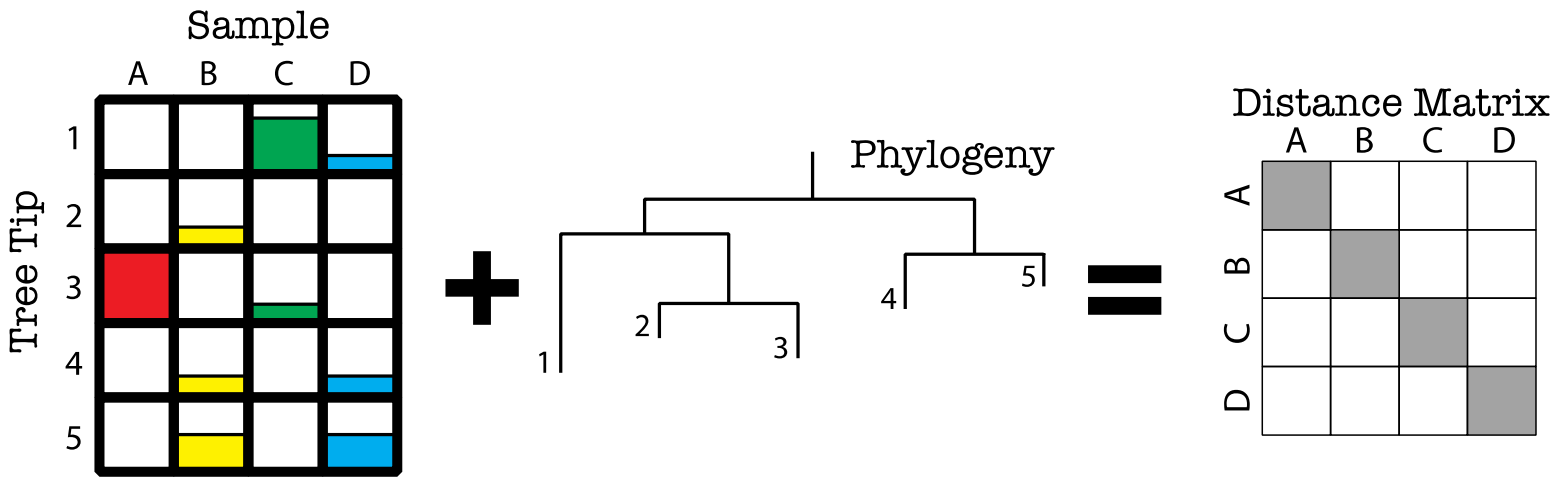
https://www.biotechniques.com/multimedia/archive/00252/microbiome2_252150a.jpg

UniFrac distance

Need to understand how similar pairs of microbiome samples are with respect to the evolutionary histories of the organisms.

UniFrac distance matrix

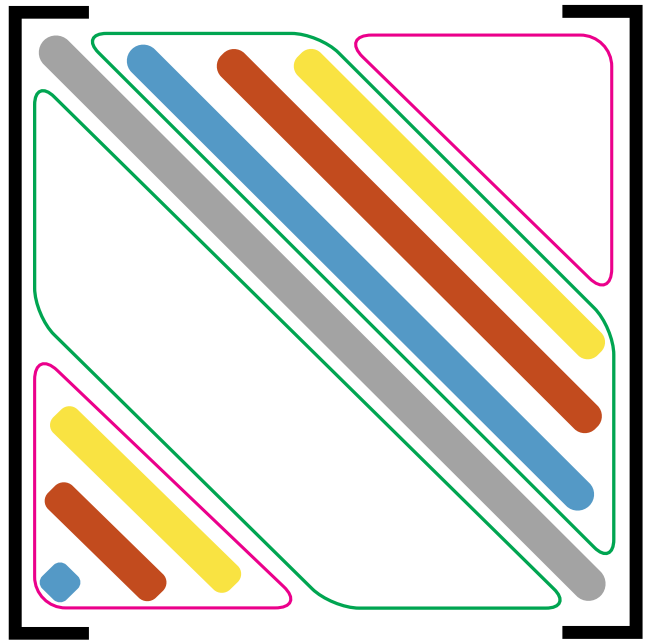
- Samples where the organisms are all very **similar** from an evolutionary perspective will have a **small** UniFrac distance.
- On the other hand, two samples composed of very **different** organisms will have a **large** UniFrac distance.

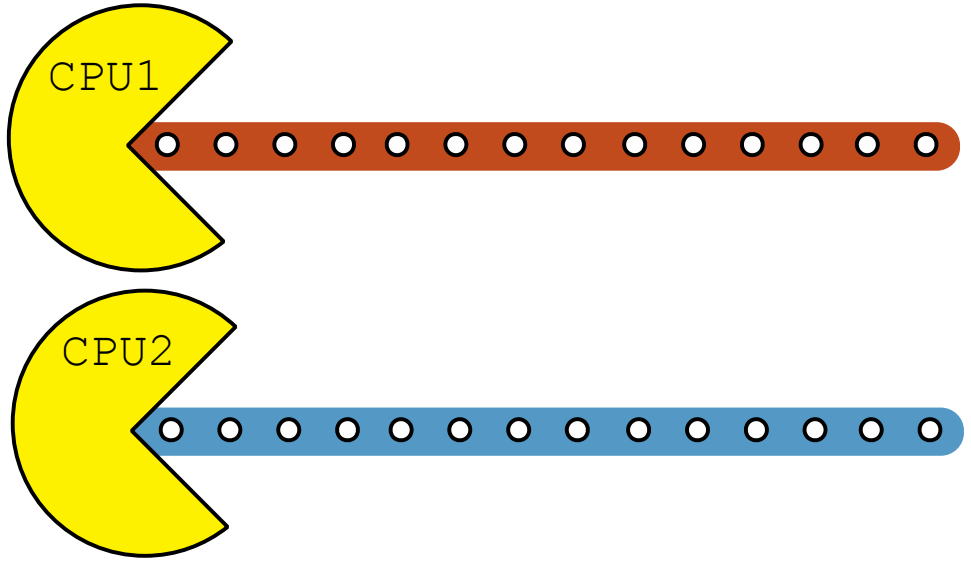


Lozupone and Knight Applied and environmental microbiology 2005

Computing UniFrac

- Matrix can be computed using a striped pattern
- Each stripe can be computed independently
- Easy to distribute over many compute units





Computing UniFrac

- Most compute localized in a tight loop
- Operating on a stripe range

```
for(unsigned int stripe = start;
    stripe < stop; stripe++) {
  dm_stripe = dm_stripes[stripe];
  for(unsigned int j = 0;
      j < n_samples / 4; j++) {
    int k = j * 4;
    double u1 = emb[k];
    double u2 = emb[k+1];
    double v1 = emb[k + stripe + 1];
    double v2 = emb[k + stripe + 2];
    …
    dm_stripe[k]   += (u1-v1)*length;
    dm_stripe[k+1] += (u2-v2)*length;
   }
}
```

Computing UniFrac

- Most compute localized in a tight loop
- Operating on a stripe range

Modest size EMP dataset

Intel Xeon E5-2680 v4 CPU
(using all 14 cores)
800 minutes (13 hours)

```
for(unsigned int stripe = start;
    stripe < stop; stripe++) {
  dm_stripe = dm_stripes[stripe];
  for(unsigned int j = 0;
      j < n_samples / 4; j++) {
    int k = j * 4;
    double u1 = emb[k];
    double u2 = emb[k+1];
    double v1 = emb[k + stripe + 1];
    double v2 = emb[k + stripe + 2];
    …
    dm_stripe[k]   += (u1-v1)*length;
    dm_stripe[k+1] += (u2-v2)*length;
   }
}
```

Porting to GPU

- OpenACC makes it trivial to port to GPU compute.
  - Just decorate with a pragma.
- But needed minor refactoring to have a unified buffer. (Was array of pointers)

```
#pragma acc parallel loop collapse(2) \
        present(emb,dm_stripes_buf)
for(unsigned int stripe = start;
    stripe < stop; stripe++) {
  for(unsigned int j = 0;
      j < n_samples / 4; j++) {
    int idx =(stripe-start_idx)*n_samples;
    double *dm_stripe =dm_stripes_buf+idx;
    int k = j * 4;
    double u1 = emb[k];
    double u2 = emb[k+1];
    double v1 = emb[k + stripe + 1];
    double v2 = emb[k + stripe + 2];
    …
    dm_stripe[k]   += (u1-v1)*length;
    dm_stripe[k+1] += (u2-v2)*length;
   }
}
```

Porting to GPU

- OpenACC makes it trivial to port to GPU compute.
  - Just decorate with a pragma.
- But needed minor refactoring to have a unified buffer. (Was array of pointers)

Modest size EMP dataset

NVIDIA Tesla V100
(using all 84 SMs)
92 minutes (1.5 hours)

Was 13h on CPU

```
#pragma acc parallel loop collapse(2) \
        present(emb,dm_stripes_buf)
for(unsigned int stripe = start;
    stripe < stop; stripe++) {
  for(unsigned int j = 0;
      j < n_samples / 4; j++) {
    int idx =(stripe-start_idx)*n_samples;
    double *dm_stripe =dm_stripes_buf+idx;
    int k = j * 4;
    double u1 = emb[k];
    double u2 = emb[k+1];
    double v1 = emb[k + stripe + 1];
    double v2 = emb[k + stripe + 2];
    …
    dm_stripe[k]   += (u1-v1)*length;
    dm_stripe[k+1] += (u2-v2)*length;
   }
}
```

Optimization step 1

- Cluster reads and minimize writes
  - Memory writes much more expensive than memory reads.
- Also undo manual unrolls
  - Were optimal for CPU
  - Bad for GPU
- Properly align memory buffers
  - Up to 5x slowdown when not aligned

Modest size EMP dataset

NVIDIA Tesla V100
(using all 84 SMs)
33 minutes

92 mins before, was 13h on CPU

```
#pragma acc parallel loop collapse(2) \
        present(emb,dm_stripes_buf,length)
for(unsigned int stripe = start;
    stripe < stop; stripe++) {
  for(unsigned int k = 0;
      k < n_samples ; k++) {
    …
    double my_stripe = dm_stripe[k];
#pragma acc loop seq
    for (unsigned int e=0;
         e<filled_embs; e++) {
      uint64_t offset = n_samples*e;
      double u = emb[offset+k];
      double v = emb[offset+k+stripe+ 1];
      my_stripe += (u-v)*length[e];
    }
    …
    dm_stripe[k]   += (u1-v1)*length;
  }
}
```

Optimization step 2

- Reorder loops to maximize cache reuse.

Modest size
EMP dataset

NVIDIA Tesla V100
(using all 84 SMs)
12 minutes

Was 33 mins

```
#pragma acc parallel loop collapse(3) \
        present(emb,dm_stripes_buf,length)
for(unsigned int sk = 0;
    sk < sample_steps ; sk++) {
  for(unsigned int stripe = start;
      stripe < stop; stripe++) {
    for(unsigned int ik = 0;
        ik < step_size ; ik++) {
      unsigned int k = sk*step_size + ik;
      …
      double my_stripe = dm_stripe[k];
#pragma acc loop seq
      for (unsigned int e=0;
           e<filled_embs; e++) {
        uint64_t offset = n_samples*e;
        double u = emb[offset+k];
        double v = emb[offset+k+stripe+ 1];
        my_stripe += (u-v)*length[e];
      }
      …
      dm_stripe[k]   += (u1-v1)*length;
    }
  }
}
```

Optimization step 2

- Reorder loops to maximize cache reuse.
- CPU code also benefitted from optimization

Originally 13h on the same CPU

Xeon E5-2680 v4 CPU (using all 14 cores)
193minutes (~3 hours)

Modest size EMP dataset

NVIDIA Tesla V100 (using all 84 SMs)
12 minutes

Was 33 mins

```
#pragma acc parallel loop collapse(3) \
        present(emb,dm_stripes_buf,length)
for(unsigned int sk = 0;
    sk < sample_steps ; sk++) {
  for(unsigned int stripe = start;
      stripe < stop; stripe++) {
    for(unsigned int ik = 0;
        ik < step_size ; ik++) {
      unsigned int k = sk*step_size + ik;
      …
      double my_stripe = dm_stripe[k];
#pragma acc loop seq
      for (unsigned int e=0;
           e<filled_embs; e++) {
        uint64_t offset = n_samples*e;
        double u = emb[offset+k];
        double v = emb[offset+k+stripe+ 1];
        my_stripe += (u-v)*length[e];
      }
      …
      dm_stripe[k]   += (u1-v1)*length;
    }
  }
}
```

20x speedup on modest EMP dataset

20x V100 GPU vs Xeon CPU + 4x from general optimization

Using fp32 adds additional boost, especially on gaming and mobile GPUs

	E5-2680 v4 CPU		GPU	GPU	GPU	GPU	GPU
	Original	New	V100	2080TI	1080TI	1080	Mobile 1050
fp64	800	193	12	59	77	99	213
fp32	-	190	9.5	19	31	36	64

140x speedup on cutting edge 113k sample

140x V100 GPUs vs Xeon CPUs + 4.5x from general optimization

Per chip (in minutes)	128x CPU E5-2680 v4		128x GPU V100	4x GPU V100	16x GPU 2080TI	16x GPU 1080TI
	Original	New				
fp64	415	97	14	29	184	252
fp32	-	91	12	20	32	82

Aggregated (in chip hours)	128x E5-2680 v4 CPU		128x GPU V100	4x GPU V100	16x GPU 2080TI	16x GPU 1080TI
	Original	New				
fp64	890	207	30	1.9	49	67
fp32	-	194	26	1.3	8.5	22

140x speedup on cutting edge 113k sample

Largish CPU cluster

Single node

GPUs vs Xeon CPUs + 4.5x ne ation

Per chip (in minutes)	128x CPU E5-2680 v4 Original	128x CPU E5-2680 v4 New	128x GPU V100	4x GPU V100	16x GPU 2080TI	16x GPU 1080TI
fp64	415	97	14	29	184	252
fp32	-	91	12	20	32	82

Aggregated (in chip hours)	128x E5-2680 v4 CPU Original	128x E5-2680 v4 CPU New	128x GPU V100	4x GPU V100	16x GPU 2080TI	16x GPU 1080TI
fp64	890	207	30	1.9	49	67
fp32	-	194	26	1.3	8.5	22

22x speedup on consumer GPUs

22x 2080TI GPUs vs Xe

Consumer GPUs slower than server GPUs but still faster than CPUs

(Memory bound)

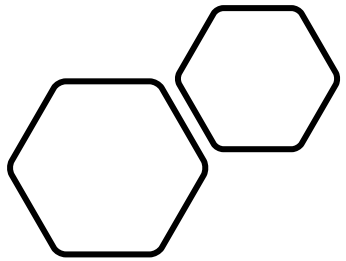
Per chip (in minutes)	128x CP E5-2680 v					
	Original	New				TI
fp64	415	97	14		184	252
fp32	-	91	12	20	32	82

Aggregated (in chip hours)	128x E5-2680 v4 CPU		128x GPU V100	4x GPU V100	16x GPU 2080TI	16x GPU 1080TI
	Original	New				
fp64	890	207	30	1.9	49	67
fp32	-	194	26	1.3	8.5	22

Desiderata

- Support for array of pointers
  - Was able to work around it, but annoying
- Better multi-GPU support
  - Currently handled with multiple processes + final merge
- (Better) AMD GPU support
  - GCC theoretically has it, but performance in tests was dismal
- Non-Linux support
  - Was not able to find an OpenACC compiler for MacOS or Windows
- Tensor compute?

Conclusions



- OpenACC made porting UniFrac to GPUs extremely easy
  - With a single code base
- Some additional optimizations were needed to get maximum benefit
  - But most were needed for the CPU-only code path, too
- Performance on NVIDIA GPUs great
  - But wondering what to do for AMD GPUs and GPUs on non-linux systems

KNIGHTLAB

Acknowledgments

This work was partially funded by US National Science Foundation (NSF) grants OAC-1826967, OAC-1541349 and CNS-1730158, and by US National Institutes of Health (NIH) grant DP1-AT010885.

UC San Diego

SDSC
SAN DIEGO SUPERCOMPUTER CENTER

PRP PACIFIC RESEARCH PLATFORM