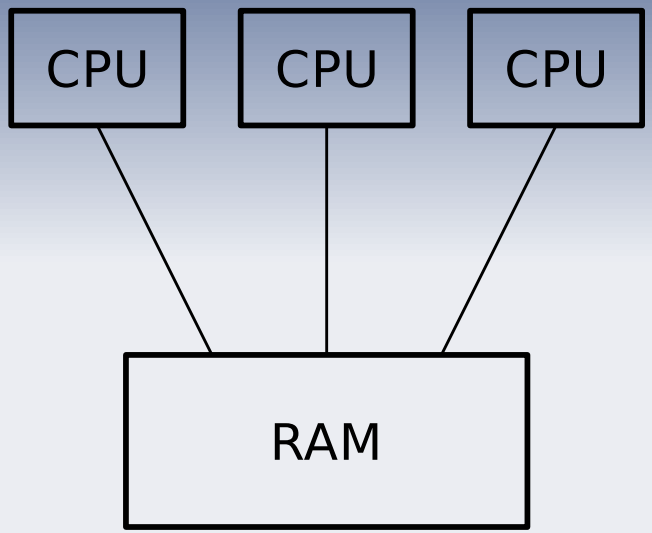
High-performance processing and development with Madagascar

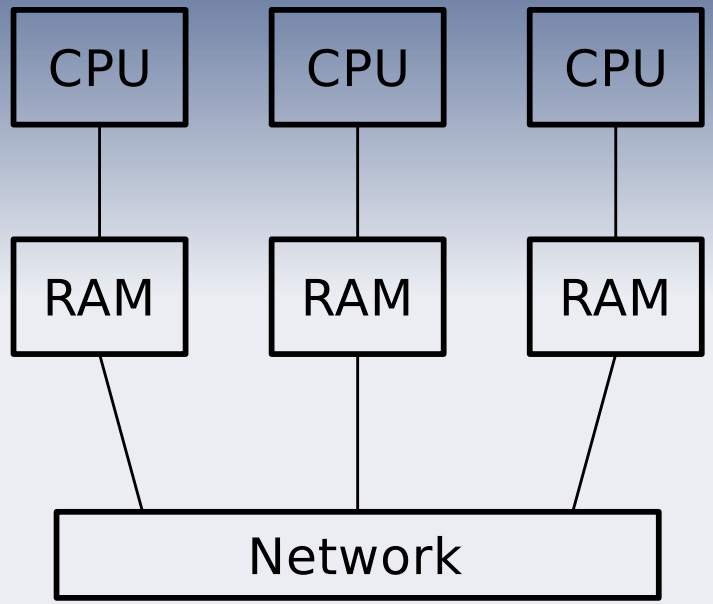
July 24, 2010

Madagascar development team

Outline



Shared memory



Distributed memory

- Vast majority of modern HPC systems are hybrid
- Rise of GPUs added another level of hardware and software complexity

Application frameworks for HPC

	Type	Supported via
OpenMP	Shared	Compiler pragmas
MPI	Distributed	Libraries[a]
GPU/CUDA	Special hardware	Extra compiler + SDK[b]

[a] Executables need special environment to run: remote shell + launcher (mpirun)

[b] Executables need proprietary drivers to run GPU kernels, but can be recompiled in emulation mode

- Most big clusters have job batch systems. On such systems, programs cannot be run directly, but have to be submitted to a queue.

Design goals for HPC support

- Support for all combination of systems
- Fault tolerance and portability of SConstructs

How?

- Utilize explicit (data) parallelism whenever possible: keep basic programs simple and sequential, handle parallelization inside **Flow**() statements automatically
- Define system-dependent execution parameters outside of SConstructs

How to define parallel Flow()

Flow('target','source','workflow',[n,m],reduce='cat')
Process 'source' independently along **n**th dimension of length **m**, concatenate all results into 'target'.

How to run

$ export **RSF_THREADS**='**16**'
$ export **RSF_CLUSTER**='hostname1 **8** hostname2 **8**'
$ pscons # or
$ scons -j **16 CLUSTER**='hostname1 **8** hostname2 **8**'

What happens inside

1. Source file gets split into several independent temporary files
2. SCons executes same workflow on cluster nodes for each file independently through remote shell (ssh)
3. SCons assembles independent outputs into one target

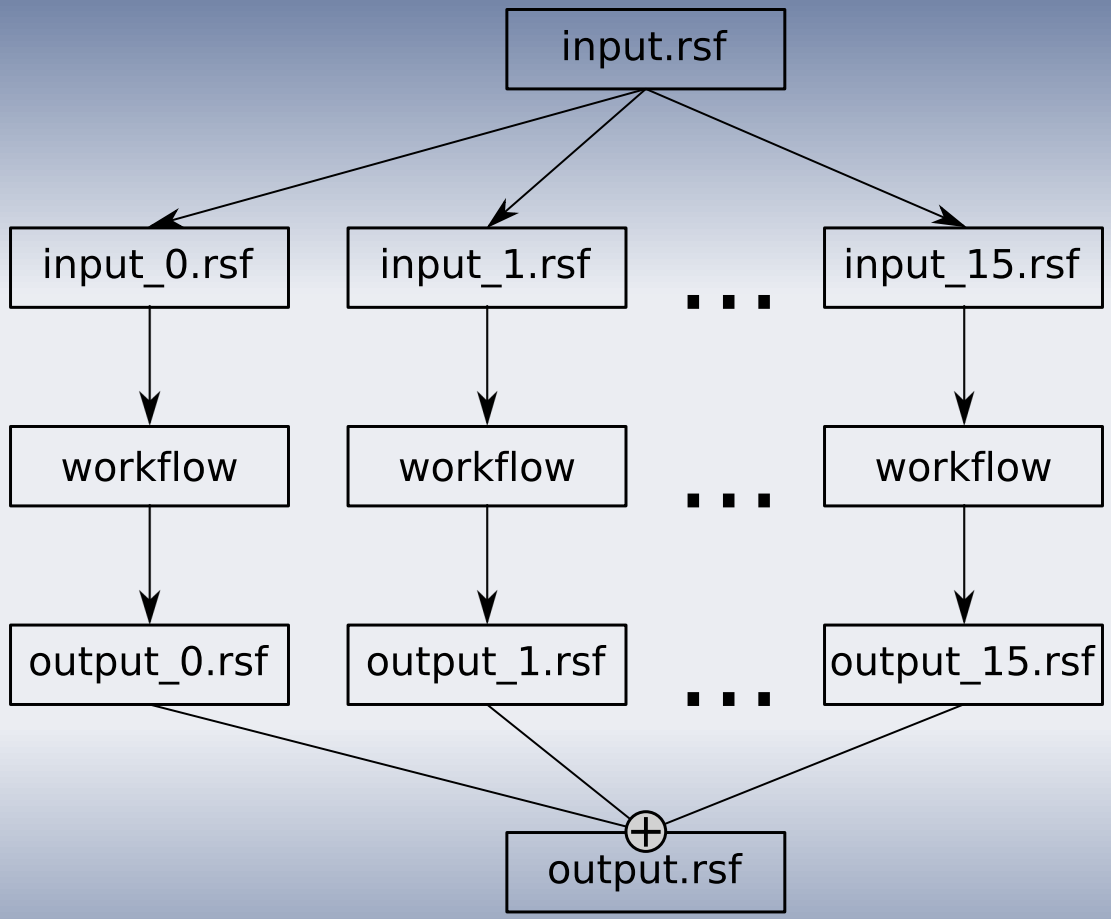


Figure: Parallel workflows with PSCons

Scratch directory for tmp files

$ export **TMPDATAPATH**=/local/fast/file/system

Monitor running programs

$ sftop

Kill remote programs

$ sfkill sfprogram

Fallbacks

If remote shell is not available directly, then it is possible to try to bypass it.

Directly through MPI

Flow('target','source','workflow',[n,'mpi',m],reduce='cat')
Try to invoke MPI executable environment directly with **mpirun** -np **m**.

Directly through OpenMP

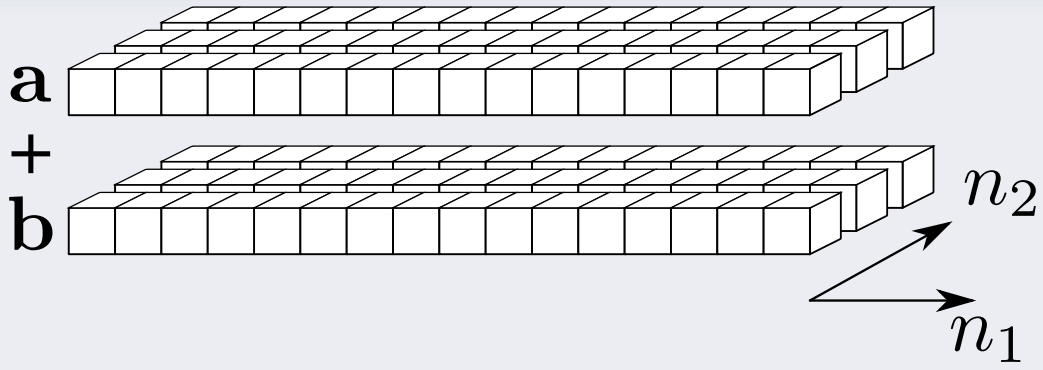
Flow('target','source','workflow',[n,'omp'],reduce='cat')
Try to run the workflow in OpenMP environment locally.

Note

These options are not as portable as the general one.

'Hello World' program of HPC world

$$\mathbf{c} = \mathbf{a} + \mathbf{b}.$$



Approaches

- First, the obvious one - **pscons**
- Then, let us assume that this problem is not data parallel and explicit parallellism has to be expressed at the level of source code

Obvious solution

```
from rsf.proj import *

Flow('a',None,'math n1=512 n2=512 output="x1+x2"')
Flow('b',None,'math n1=512 n2=512 output="-x1-x2"')

Flow('c','a b','math b=${SOURCES[1]} output="input+b"',
                split=[2,512])
End()
```

How to run

$ pscons

Note

- This approach will work on any type of system and on a single-CPU machine as well
- Splitting inputs and collecting outputs add overhead

vecsum.job - job file for running inside LSF batch system

```
#!/bin/bash

#BSUB -J vecsum          # Job name
#BSUB -o vecsum.out      # Name of the output file
#BSUB -q normal          # Queue name
#BSUB -P rsfdev          # Account
#BSUB -W 0:05            # runtime
#BSUB -n 16              # number of CPUs

export RSF_THREADS=16
export RSF_CLUSTER=$LSB_MCPU_HOSTS
pscons
```

How to run

```
$ bsub <vecsum.job
```

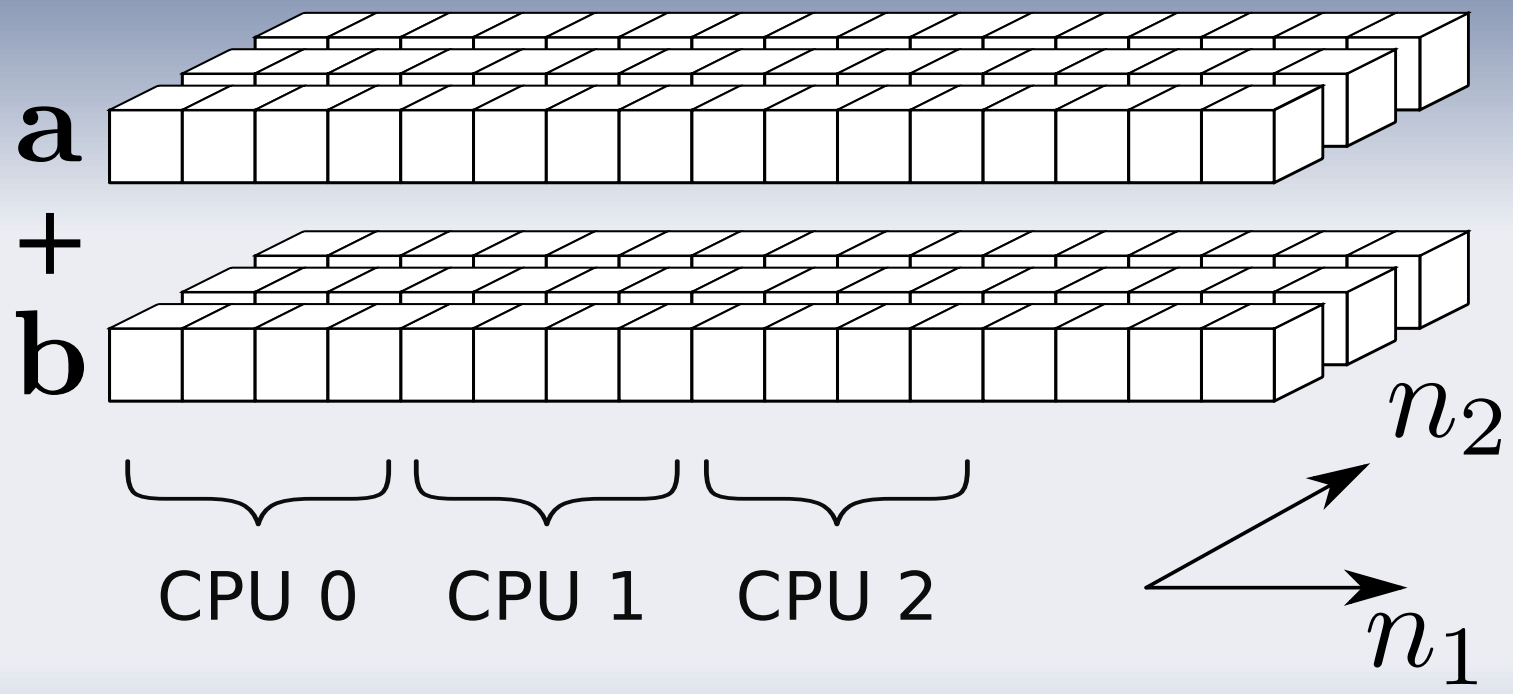


Figure: Summation of vectors with OpenMP

omphello.c

```
#include <rsf.h>

int main (int argc, char *argv[]) {
    int n1, n2, i, j;
    float *a, *b, *c;

    sf_file ain, bin, cout = NULL;

    sf_init (argc, argv);
    ain = sf_input ("in"); /* Input vector a */
    bin = sf_input ("b"); /* Input vector b */
    if (SF_FLOAT != sf_gettype (ain) ||
        SF_FLOAT != sf_gettype (bin))
        sf_error ("Need float");
    /* Vector size */
    if (!sf_histint (ain, "n1", &n1)) sf_error ("No n1=");
    /* Number of vectors */
    n2 = sf_leftsize (ain, 1);
```

omphello.c

```
    /* Output vector */
    cout = sf_output ("out");
    /* Vectors in memory */
    a = sf_floatalloc (n1); b = sf_floatalloc (n1);
    c = sf_floatalloc (n1);
    /* Outer loop over vectors */
    for (i = 0; i < n2; i++) {
        sf_floatread (a, n1, ain);
        sf_floatread (b, n1, bin);
        /* Parallel summation */
#pragma omp parallel for private(j) shared(a,b,c)
        for (j = 0; j < n1; j++)
            c[j] = a[j] + b[j];
        sf_floatwrite (c, n1, cout);
    }
    sf_fileclose (ain); sf_fileclose (bin);
    sf_fileclose (cout);
    return 0;
}
```

How to compile

Like any regular Madagascar program.

How to run

$ scons/pscons

Note

- Even if OpenMP is not supported, the source code will get compiled as sequential
- It can be combined with upper-level parallelization in SConstruct
- For a good real-life example, look into RSFSRC/user/psava/sfawefd.c

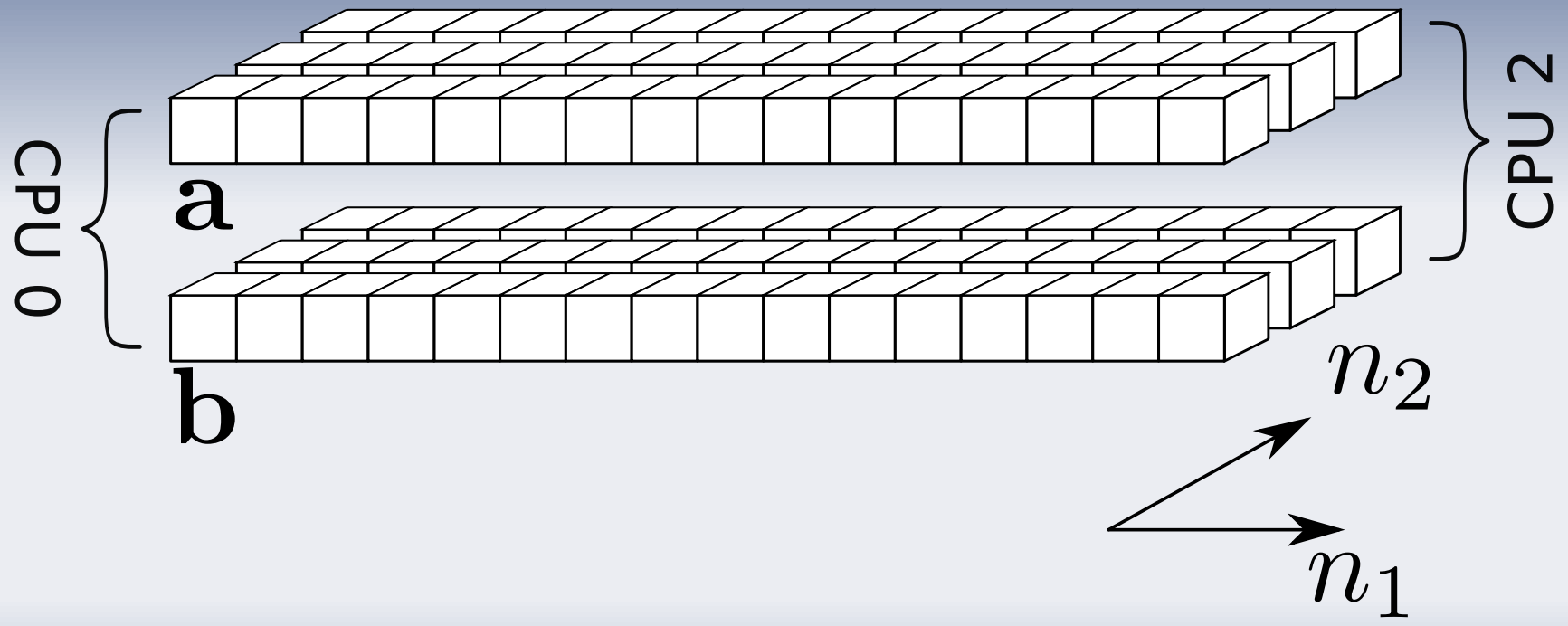


Figure: Summation of vectors with MPI

mpihello.c

```c
#include <rsf.h>
#include <mpi.h>

int main (int argc, char *argv[]) {
    int n1, n2, nc, esize, i, j, k = 0;
    float *a, *b, **c;

    sf_file ain, bin, cout = NULL;

    int cpuid; /* CPU id */
    int ncpu; /* Number of CPUs */
    MPI_Status mpi_stat;

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &cpuid);
    MPI_Comm_size (MPI_COMM_WORLD, &ncpu);

    sf_init (argc, argv);
```

mpihello.c

```
ain = sf_input ("input"); /* Input vector a */
bin = sf_input ("b"); /* Input vector b */
if (SF_FLOAT != sf_gettype (ain) ||
    SF_FLOAT != sf_gettype (bin))
    sf_error ("Need float");
/* Size of an element */
if (!sf_histint (ain, "esize", &esize))
    esize = sizeof(float);
/* Vector size */
if (!sf_histint (ain, "n1", &n1)) sf_error ("No n1=");
/* Total number of vectors */
n2 = sf_leftsize (ain, 1);
/* Only the first CPU will do output */
if (0 == cpuid) {
    cout = sf_output ("out");
    sf_putint (cout, "n1", n1);
    sf_putint (cout, "n2", n2);
    sf_warning ("Running on %d CPUs", ncpu);
}
```

mpihello.c

```
    a = sf_floatalloc (n1);
    b = sf_floatalloc (n1);
    /* How many vectors per CPU */
    nc = (int)(n2/(float)ncpu + 0.5f);
    c = sf_floatalloc2 (n1, nc);
    /* Starting position in input files */
    sf_seek (ain, n1*cpuid*esize, SEEK_CUR);
    sf_seek (bin, n1*cpuid*esize, SEEK_CUR);
    for (i = cpuid; i < n2; i += ncpu, k++) {
        /* Read local portion of input data */
        sf_floatread (a, n1, ain);
        sf_floatread (b, n1, bin);
        /* Parallel summation here */
        for (j = 0; j < n1; j++)
            c[k][j] = a[j] + b[j];
        /* Move on to the next portion */
        sf_seek (ain, n1*(ncpu - 1)*esize, SEEK_CUR);
        sf_seek (bin, n1*(ncpu - 1)*esize, SEEK_CUR);
    }
```

mpihello.c

```
    if (0 == cpuid) { /* Collect results from all nodes */
        for (i = 0; i < n2; i++) {
            k = i / ncpu; /* Iteration number */
            j = i % ncpu; /* CPU number to receive from */
            if (j) /* Receive from non-zero CPU */
                MPI_Recv (&c[k][0], n1, MPI_FLOAT, j, j,
                          MPI_COMM_WORLD, &mpi_stat);
            sf_floatwrite (c[k], n1, cout);
        }
        sf_fileclose (cout);
    } else { /* Send results to CPU #0 */
        for (i = 0; i < k; i++) /* Vector by vector */
            MPI_Send (&c[i][0], n1, MPI_FLOAT, 0, cpuid,
                      MPI_COMM_WORLD);
    }
    sf_fileclose (ain); sf_fileclose (bin);
    MPI_Finalize ();
    return 0;
}
```

How to compile

Look into RSFSRC/system/main/SConstruct for mpi.c

How to run

```
$ export MPIRUN=/path/to/my/mpirun # If it is not standard
$ scons
```

Note

- No standard input in the program
- Pipes are not allowed in **Flow**()

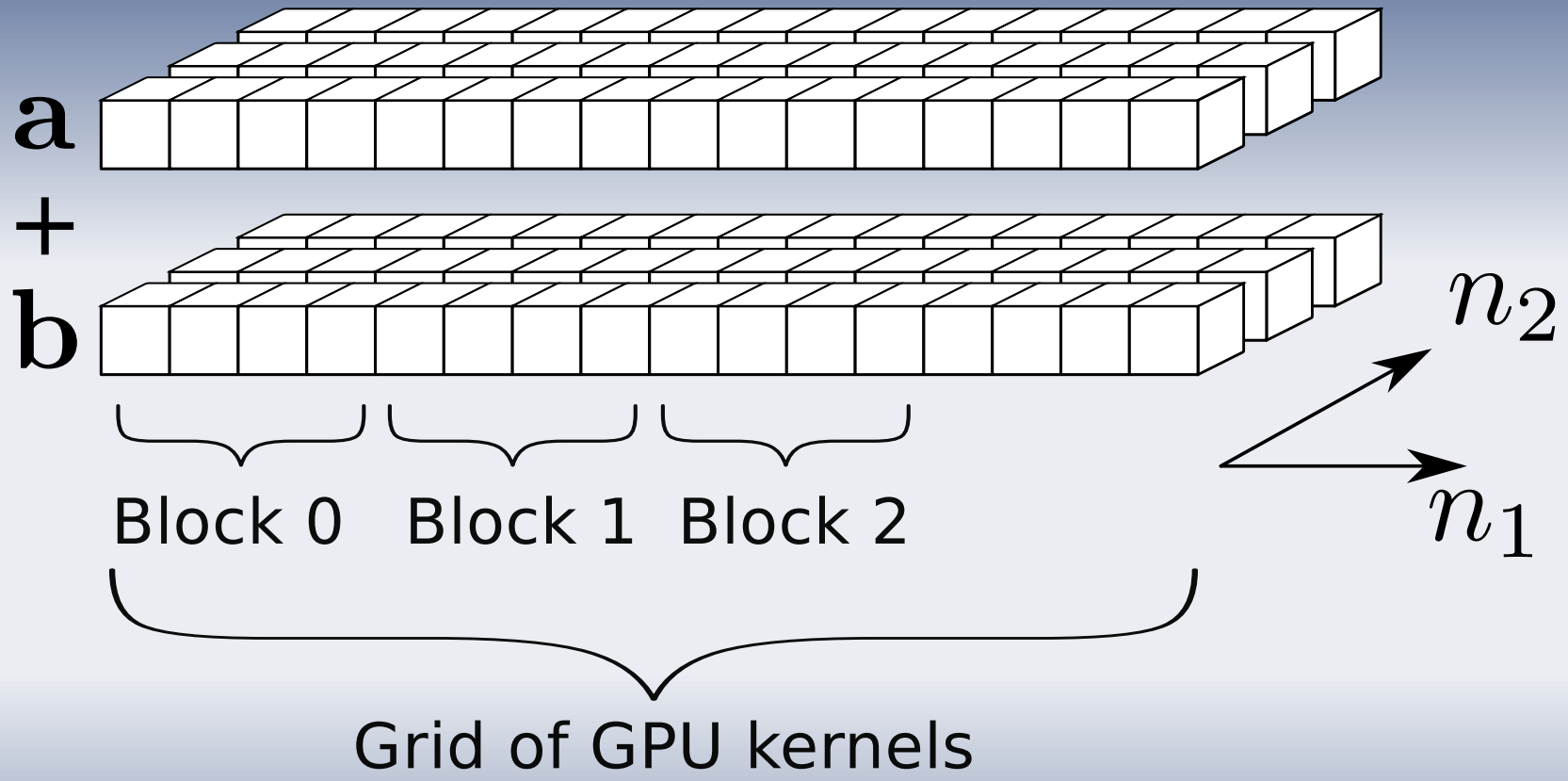


Figure: Summation of vectors on GPU

gpuhello.c

```
#define BLOCK_SIZE 128
__global__ void gpu_vec_sum (float *a, float *b,
                             float *c) {
    const unsigned int j = blockIdx.x*blockDim.x +
                           threadIdx.x;
    c[j] = a[j] + b[j];
}
int main (int argc, char* argv[]) {
    int n1, n2, esize, i;
    float *a, *b, *c;
    sf_file ain, bin, cout = NULL;
    dim3 dimgrid (1, 1, 1); /* GPU grid */
    dim3 dimblock (BLOCK_SIZE, 1, 1); /* GPU block */
    float *d_a, *d_b, *d_c; /* GPU pointers */

    sf_init (argc, argv);
    cuInit (0); /* Use first GPU device */
    sf_check_gpu_error ("Device initialization");
    cudaSetDevice (0);
```

gpuhello.c

```
ain = sf_input ("in"); /* Input vector a */
bin = sf_input ("b"); /* Input vector b */
if (SF_FLOAT != sf_gettype (ain) ||
    SF_FLOAT != sf_gettype (bin))
    sf_error ("Need float");
/* Size of an element */
if (!sf_histint (ain, "esize", &esize))
    esize = sizeof(float);
/* Vector size */
if (!sf_histint (ain, "n1", &n1)) sf_error ("No n1=");
/* Number of vectors */
n2 = sf_leftsize (ain, 1);
/* Output vector */
cout = sf_output ("out");
/* Vectors in CPU memory */
a = sf_floatalloc (n1); b = sf_floatalloc (n1);
c = sf_floatalloc (n1);
```

gpuhello.c

```
/* Vectors in GPU memory */
cudaMalloc ((void**)&d_a, n1*esize);
cudaMalloc ((void**)&d_b, n1*esize);
cudaMalloc ((void**)&d_c, n1*esize);
sf_check_gpu_error ("GPU mallocs");
/* Kernel configuration for this data */
dimgrid = dim3 (n1/BLOCK_SIZE, 1, 1);
/* Outer loop over vectors */
for (i = 0; i < n2; i++) {
    sf_floatread (a, n1, ain); /* Input */
    sf_floatread (b, n1, bin);
    cudaMemcpy (d_a, a, n1*esize, /* a -> GPU */
                cudaMemcpyHostToDevice);
    cudaMemcpy (d_b, b, n1*esize, /* b -> GPU */
                cudaMemcpyHostToDevice);
    sf_check_gpu_error ("Copying a&b to GPU");
    /* Parallel summation on GPU */
    gpu_vec_sum<<<dimgrid, dimblock>>>(d_a, d_b, d_c);
```

gpuhello.c

```
        /* Parallel summation on GPU */
        gpu_vec_sum<<<dimgrid, dimblock>>>(d_a, d_b, d_c);
        sf_check_gpu_error ("Kernel execution");
        cudaMemcpy (c, d_c, n1*esize, /* GPU -> c */
                    cudaMemcpyDeviceToHost);
        sf_check_gpu_error ("Copying c from GPU");
        sf_floatwrite (c, n1, cout); /* Output */
    }
    sf_fileclose (ain);
    sf_fileclose (bin);
    sf_fileclose (cout);
    return 0;
}
```

How to compile

Look into RSFSRC/user/cuda/SConstruct

How to run

$ scons/pscons

Note

- Can be launched on a multi-GPU cluster by **pscons**
- Can be piped like a sequential Madagascar program
- You will need a dedicated GPU to run this code efficiently
- CUDA is proprietary

Takeaway message

Exploit data parallelism when possible. It keeps programs simple and SConstructs portable.