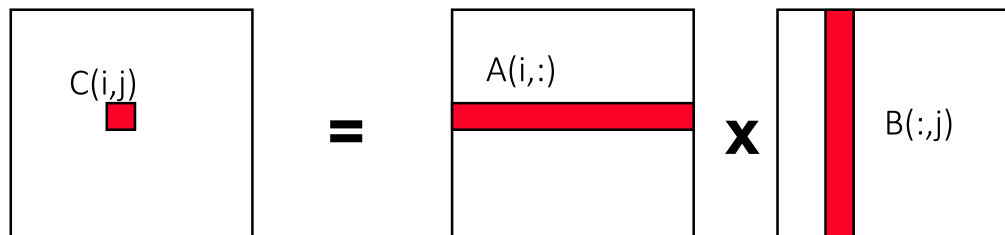
Australian National University

The OpenCL Memory Hierarchy

Slides taken from *Hands On OpenCL* by Simon McIntosh-Smith, Tom Deakin, James Price, Tim Mattson and Benedict Gaster under the "attribution CC BY" creative commons license.

Optimizing matrix multiplication

- MM cost determined by FLOP/s and memory movement:
  - $2*n^3 = O(n^3)$ FLOP/s
  - Operates on $3*n^2 = O(n^2)$ numbers
- To optimize matrix multiplication, we must ensure that for every memory access we execute as many FLOP/s as possible.
- Outer product algorithms are faster, but for pedagogical reasons, let's stick to the simple dot-product algorithm.

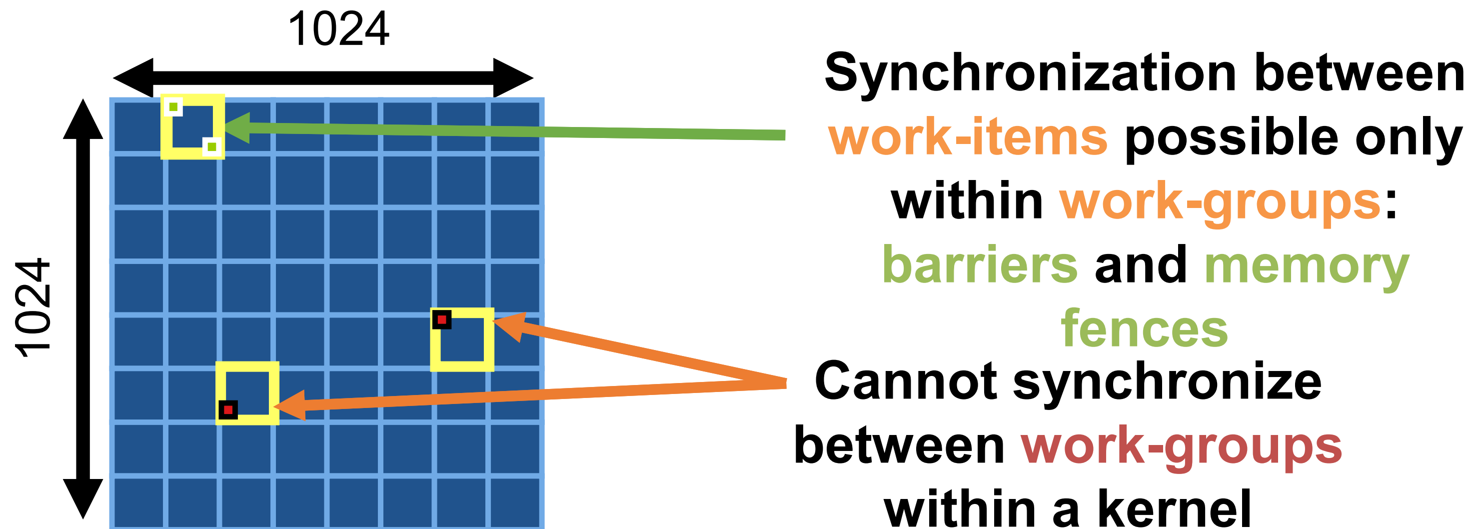


Dot product of a row of A and a column of B for each element of C

- We will work with work-item/work-group sizes and the memory model to optimize matrix multiplication

An N-dimensional domain of work-items

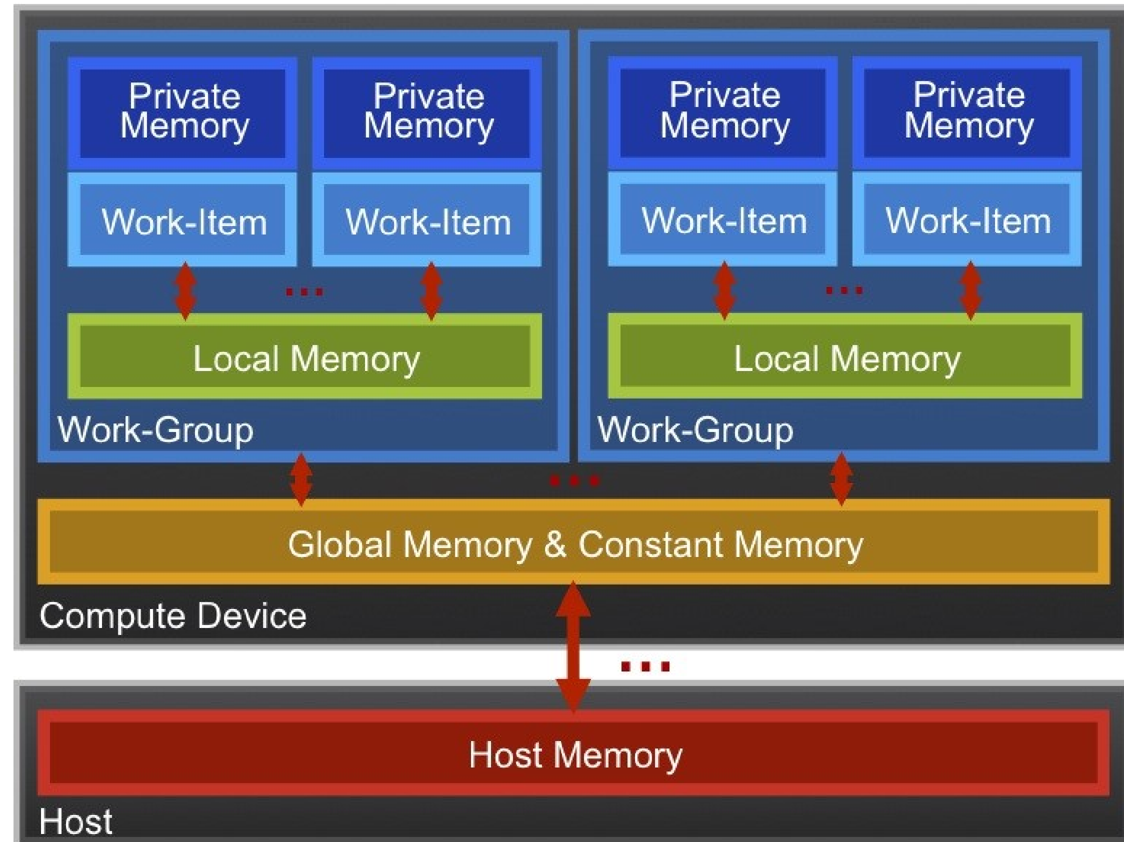
- Global Dimensions:
  - 1024x1024 (whole problem space)
- Local Dimensions:
  - 128x128 (**work-group**, executes together)



- Choose the dimensions that are best for your algorithm

OpenCL Memory model

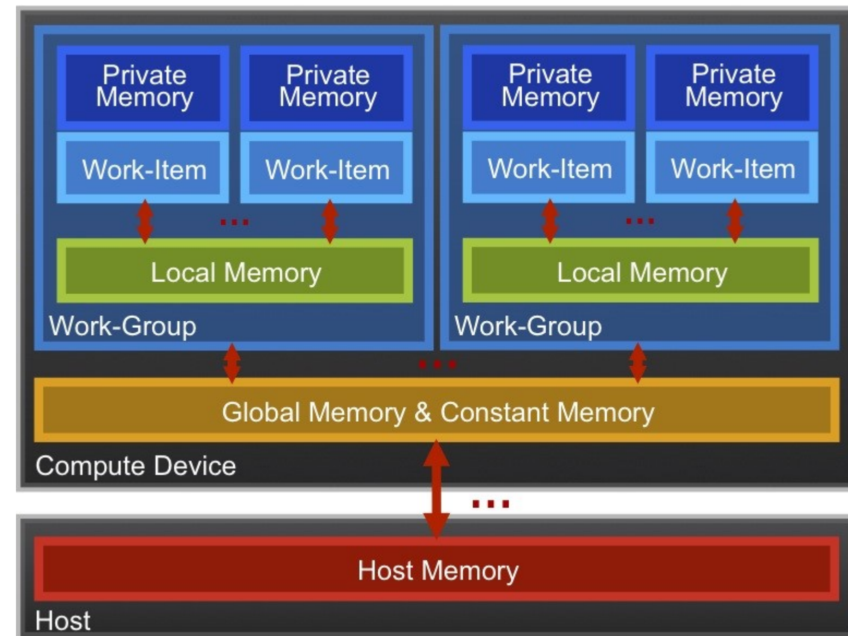
- Private Memory
  - Per work-item
- Local Memory
  - Shared within a work-group
- Global / Constant Memory
  - Visible to all work-groups
- Host memory
  - On the CPU



Memory management is **explicit**:
You are responsible for moving data from
host → global → local *and* back

OpenCL Memory model

- Private Memory
  - Fastest & smallest: O(10) words/WI
- Local Memory
  - Shared by all WI’s in a work-group
  - But not shared between work-groups!
  - O(1-10) KBytes per work-group
- Global / Constant Memory
  - O(1-10) GBytes of Global memory
  - O(10-100) KBytes of Constant memory
- Host memory
  - On the CPU - GBytes



Memory management is **explicit**:
O(1-10) GBytes/s bandwidth to discrete GPUs for Host <-> Global transfers

Private Memory

- Managing the memory hierarchy is one of ***the*** most important things to get right to achieve good performance

- Private Memory:
  - A **very scarce** resource, only a few tens of (32-bit) words per work-item at most
  - If you use too much it spills to global memory or reduces the number of Work-Items that can be run at the same time, potentially harming performance*
  - Think of these like registers on the CPU

* Occupancy on a GPU

Local Memory*

- Tens of KBytes per Compute Unit
  - As multiple Work-Groups will be running on each Compute Unit, this means only a fraction of the total Local Memory size is available to each Work-Group
- Assume O(1-10) KBytes of Local Memory per Work-Group
  - Your kernels are responsible for transferring data between Local and Global/Constant memories. There are optimized library functions to help
  - E.g. async_work_group_copy(), async_work_group_strided_copy(), …
- Use Local Memory to hold data that can be reused by all the work-items in a work-group
- Access patterns to Local Memory affect performance in a similar way to accessing Global Memory
  - Have to think about things like coalescence & bank conflicts

* Typical figures for a discrete GPU

Local Memory

- Local Memory doesn't always help…
  - CPUs don't have special hardware for it
  - This can mean excessive use of Local Memory might slow down kernels on CPUs
  - GPUs now have effective on-chip caches which can provide much of the benefit of Local Memory but without programmer intervention
  - So, your mileage may vary!

The Memory Hierarchy

	Bandwidths	Sizes
Private memory	O(2-3) words/cycle/WI	O(10) words/WI
Local memory	O(10) words/cycle/WG	O(1) KBytes/WG
Global memory	O(800-1,000) GBytes/s	O(10) GBytes
Host memory	O(10) GBytes/s	O(10-100) GBytes

Speeds and feeds approx. for a high-end discrete GPU, circa 2018

Memory Consistency

- OpenCL uses a **relaxed consistency** memory model; i.e.
  - The state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times.
- For each work-item:
  - Memory has load/store consistency to the work-item's private view of memory, i.e. it sees its own reads and writes correctly
- Between work-items in a work-group:
  - Local memory is consistent at a **barrier**.
- Global memory is consistent within a work-group at a barrier, **but *not* guaranteed across different work-groups!!**
  - This is a common source of bugs!
- Consistency of memory shared between **commands** (e.g. kernel invocations) is enforced by **synchronization** (barriers, events, in-order queue)

Work-Item Synchronization

Ensure correct order of memory operations to local or global memory (with flushes or queuing a memory fence)

- ***Within*** a work-group:

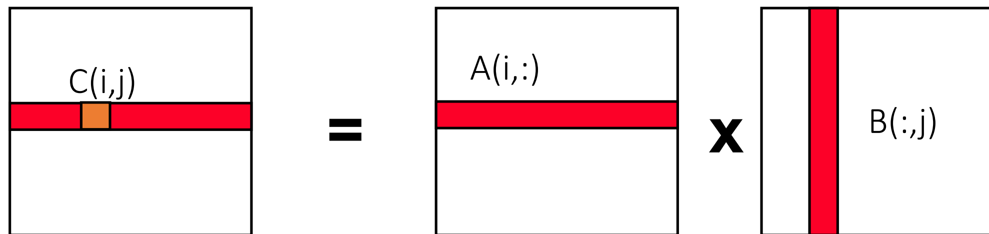
```
void barrier()
```

  - Takes optional flags
    CLK_LOCAL_MEM_FENCE and/or CLK_GLOBAL_MEM_FENCE
  - A work-item that encounters a barrier() will wait until ALL work-items in its work-group reach the barrier()
  - Corollary: If a barrier() is inside a branch, then the branch must be ***uniform***, i.e. taken by either:
    - ALL work-items in the work-group, OR
    - NO work-item in the work-group

- Between different work-groups:
  - No guarantees as to where and when a particular work-group will be executed relative to other work-groups
  - Cannot exchange data, or have barrier-like synchronization between two different work-groups! (Critical issue!)
  - **Only solution**: finish executing the kernel and start executing another

Optimizing matrix multiplication

- There may be significant overhead to manage work-items and work-groups.
- So let's try having each work-item compute a full row of C



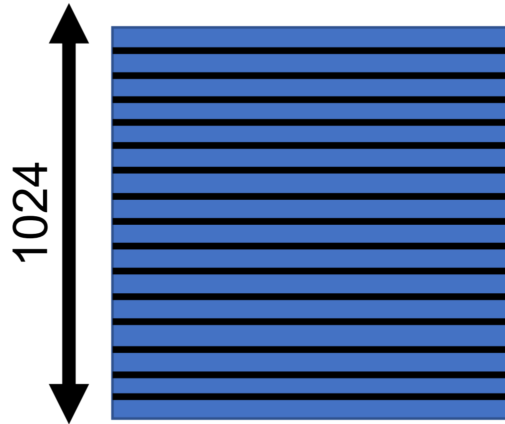
Dot product of a row of A and a column of B for each element of C

An N-dimension domain of work-items

- Global Dimensions: 1024 (1D)

  Whole problem space (index space)

- Local Dimensions: leave to the run-time



- **Important implication**: we will have a lot fewer work-items and work-groups. Why might this matter?

Reduce work-item overhead

Do a whole row of C per work-item

```
__kernel void mmul(const int N,
 __global float *A, __global float *B, __global float *C)
{
    int k, j;
    int i = get_global_id(0);
    float tmp;
    for (j = 0; j < N; j++) {
        // N is width of rows in C
        tmp = 0.0f;
        for (k = 0; k < N; k++)
            tmp += A[i*N+k] * B[k*N+j];
        C[i*N+j]  = tmp;
    }
}
```

Matrix multiplication host program (C++ API)

```
int main(int argc, char *argv[])
{
  std::vector<float> h_A, h_B, h_C; // matrices
  int N; // A[N][N],B[N][N],C[N][N]
  int i, err;
  int size; // num elements in each matrix
  double start_time, run_time; // timing data
  cl::Program program;

  N     =
  size  =
  h_A   =
  h_B   =
  h_C   = std::vector<float>(size);

  initmat(N, h_A, h_B, h_C);

  // Compile for first kernel to setup program
  program = cl::Program(C_elem_KernelSource, true);
  Context context(CL_DEVICE_TYPE_DEFAULT);
  cl::CommandQueue queue(context);
  std::vector<Device> devices =
      context.getInfo<CL_CONTEXT_DEVICES>();
  cl::Device device = devices[0];
  std::string s =
      device.getInfo<CL_DEVICE_NAME>();
  std::cout << "\nUsing OpenCL Device "
            << s << "\n";

  // Setup the buffers, initialize matrices,
  // and write them into global memory
  cl::Buffer d_a(context, begin(h_A), end(h_A), true);
  cl::Buffer d_b(context, begin(h_B), end(h_B), true);
  cl::Buffer d_c(context, CL_MEM_WRITE_ONLY,
                          sizeof(float) * size);

             <int,cl::Buffer, cl::Buffer, cl::Buffer>
             krow(program, "mmul");

  zero_mat(N, h_C);
  start_time = wtime();

  krow(cl::EnqueueArgs(queue
                       cl::NDRange(N)),
       N, d_a, d_b, d_c);

  run_time  = wtime() - start_time;

  cl::copy(queue, d_c, begin(h_C), end(h_C));

  results(N, h_C, run_time);
}
```

Changes to host program:

1. 1D ND Range set to number of rows in the C matrix

Matrix multiplication performance

- Matrices are stored in global memory.

Case	GFLOP/s	
	CPU	GPU
C(i,j) per work-item, all global	111.8	70.3

This hasn't helped.

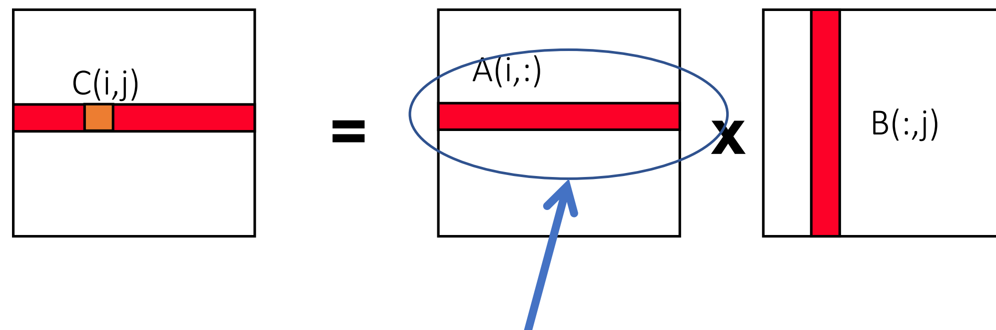
Device is NVIDIA® Tesla® P100 GPU with 56 compute units, 3,584 PEs
Device is 2x Intel® Xeon® CPU, E5-2695 v4 @ 2.1GHz

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Third party names are the property of their owners.

Optimizing matrix multiplication

- Notice that, in one row of C, each element reuses the same row of A.
- Let’s copy that row of A from global memory into private memory of the work-item that’s (exclusively) using it, to avoid the overhead of loading it from global memory for each C(i,j) computation.



Matrix multiplication: OpenCL kernel (3/3)

```
__kernel void mmul(
    const int N,
    __global float *A,
    __global float *B,
    __global float *C)
{
  int k, j;
  int i = get_global_id(0);
  float Awrk[1024];
  float tmp;
        for (k = 0; k < N; k++)
            Awrk[k] = A[i*N+k];
        for (j = 0; j < N; j++) {
           tmp = 0.0f;
           for (k = 0; k < N; k++)
               tmp += Awrk[k]*B[k*N+j];
           C[i*N+j] = tmp;
        }
}
```

Setup a work array for A in private memory and copy into it from global memory before we start with the matrix multiplications.

(Actually, this is using *far* more private memory than we'll have and so Awrk[] will be spilled to global memory)

Matrix multiplication performance

- Matrices are stored in global memory.

Case	GFLOP/s	
	CPU	GPU
C(i,j) per work-item, all global	111.8	70.3
C row per work-item, A row private	9.6	24.9

Mixed, and still slower than naïve

Device is NVIDIA® Tesla® P100 GPU with 56 compute units, 3,584 PEs
Device is 2x Intel® Xeon® CPU, E5-2695 v4 @ 2.1GHz

Third party names are the property of their owners.

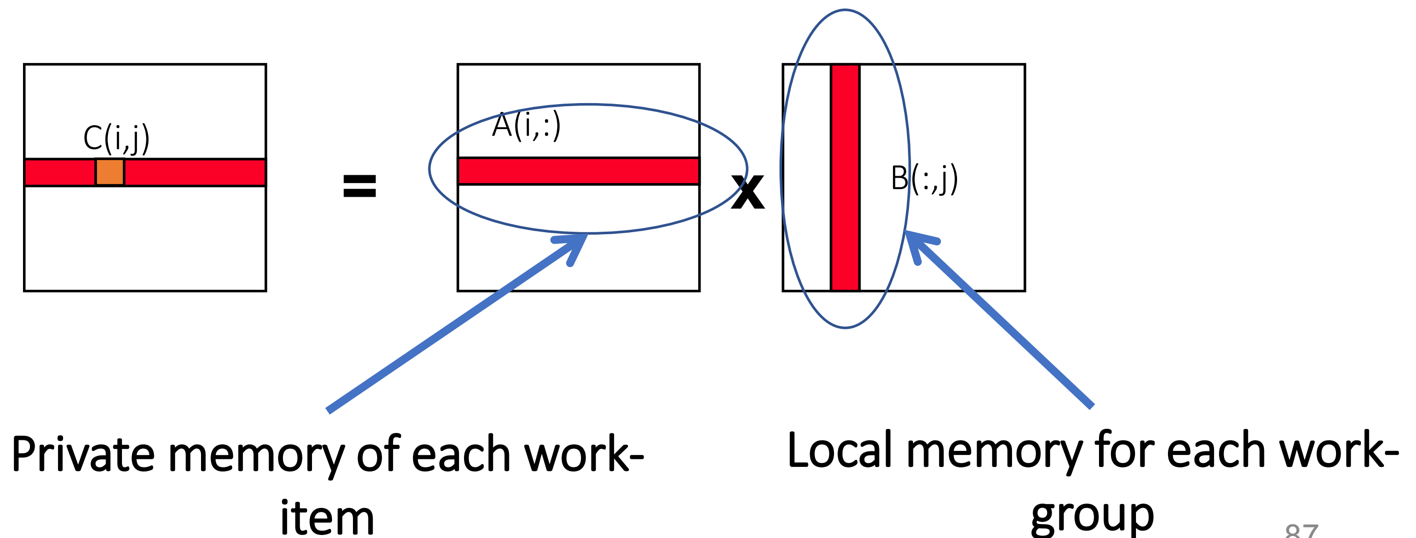
These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Why using too much private memory can be a good thing

- In reality private memory is just hardware registers, so only dozens of these are available per work-item
- Many kernels will allocate too many variables to private memory
  - So the compiler already has to be able to deal with this
- It does so by *spilling* excess private variables to (global) memory
- You still told the compiler something useful – that the data will only be accessed by a single work-item
- This lets the compiler allocate the data in such as way as to enable more efficient memory access

Optimizing matrix multiplication

- We already noticed that, in one row of C, each element uses the same row of A
- Each work-item in a work-group also uses the same columns of B
- So let's store the B columns in **local** memory (which is shared by the work-items in the work-group)



Row of C per work-item, A row private, B columns local

```
__kernel void mmul(
    const int N,
    __global float *A,
    __global float *B,
    __global float *C,
    __local float *Bwrk)
{
  int k, j;
  int i = get_global_id(0);
  int iloc = get_local_id(0);
  int nloc = get_local_size(0);
  float Awrk[1024];

  float tmp;
  for (k = 0; k < N; k++)
    Awrk[k] = A[i*N+k];
  for (j = 0; j < N; j++) {
    barrier(CLK_LOCAL_MEM_FENCE);
    for (k=iloc; k<N; k+=nloc)
      Bwrk[k] = B[k*N+j];
    barrier(CLK_LOCAL_MEM_FENCE);
    tmp = 0.0f;
    for (k = 0; k < N; k++)
      tmp += Awrk[k] * Bwrk[k];
    C[i*N+j] = tmp;
  }
}
```

Pass in a pointer to local memory. Work-items in a work-group start by cooperatively copying the columns of B they need into the work-group's local memory.

Matrix multiplication host program (C++ API)

Changes to host program: Pass local memory to kernels.

1. This requires a change to the kernel argument lists … an arg of type LocalSpaceArg is needed.
2. Allocate the size of local memory
3. Update argument list in kernel functor

```
int main
{
  std::ve
  int N;
  int i,
  int si
  double
  cl::Program program;

  N     = ORDER;
  size  = N*N;
  h_A   = std::vector<float>(size);
  h_B   = std::vector<float>(size);
  h_C   = std::vector<float>(size);

  initmat(N, h_A, h_B, h_C);

  // Compile for first kernel to setup program
  program = cl::Program(C_elem_KernelSource, true);
  Context context(CL_DEVICE_TYPE_DEFAULT);
  cl::CommandQueue queue(context);
  std::vector<Device> devices =
      context.getInfo<CL_CONTEXT_DEVICES>();
  cl::Device device = devices[0];
  std::string s =
      device.getInfo<CL_DEVICE_NAME>();
  std::cout << "\nUsing OpenCL Device "
            << s << "\n";
```

```
                                                        ue);
                                                        ue);
                              CL_MEM_WRITE_ONLY,
                              sizeof(float) * size);
cl::LocalSpaceArg localmem =
                    cl::Local(sizeof(float) * N);
cl::KernelFunctor<int, cl::Buffer, cl::Buffer,
                cl::Buffer, cl::LocalSpaceArg>
               rowcol(program, "mmul");
zero_mat(N, h_C);
start_time = wtime();

rowcol(cl::EnqueueArgs(queue, cl::NDRange(N)),
       N, d_a, d_b, d_c, localmem);

run_time  = wtime() - start_time;

cl::copy(queue, d_c, begin(h_C), end(h_C));

results(N, h_C, run_time);
```

Matrix multiplication performance

- Matrices are stored in global memory.

Case	GFLOP/s	
	CPU	GPU
C(i,j) per work-item, all global	111.8	70.3
C row per work-item, A row private	9.6	24.9

Device is NVIDIA® Tesla® P100 GPU with 56 compute units, 3,584 PEs
Device is 2x Intel® Xeon® CPU, E5-2695 v4 @ 2.1GHz

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Third party names are the property of their owners.

Making matrix multiplication *really* fast

- Our goal has been to describe how to work with private, local and global memory. We've ignored many well-known techniques for making matrix multiplication fast
  - The number of work-items should be a multiple of the fundamental machine "vector width". This is the wavefront on AMD, warp on NVIDIA, and the number of SIMD lanes exposed by vector units on a CPU
  - To optimize reuse of data, you need to use *blocking* techniques
    - Decompose matrices into tiles such that three tiles just fit in the fastest memory
    - Copy tiles into local memory
    - Do the multiplication over the tiles
  - We have provided a very fast yet still quite simple block matrix multiply solution in OpenCL. This uses blocking with block sizes mapped onto the GPU's warp/wavefront size. We'll come back to this later in the advanced section

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
  int i, j, k;
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      for (k = 0; k < N; k++)
        C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{ // assume N % block_size = 0
  int i, j, k;
  int NB = N/block_size;
  for (ib = 0; ib < NB; ib++)
    for (i = ib*NB; i < (ib+1)*NB; i++)
      for (jb = 0; jb < NB; jb++)
        for (j = jb*NB; j < (jb+1)*NB; j++)
          for (kb = 0; kb < NB; kb++)
            for (k = kb*NB; k < (kb+1)*NB; k++)
              C[i*N+j] += A[i*N+k]*B[k*N+j];


}
```

Break each loop into chunks with a size chosen to match the size of your fast memory

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{ // assume N % block_size = 0
  int i, j, k;
  int NB = N/block_size;
  for (ib = 0; ib < NB; ib++)
    for (jb = 0; jb < NB; jb++)
      for (kb = 0; kb < NB; kb++)
        for (i = ib*NB; i < (ib+1)*NB; i++)
          for (j = jb*NB; j< (jb+1)*NB; j++)
            for (k = kb*NB; k < (kb+1)*NB; k++)
              C[i*N+j] += A[i*N+k]*B[k*N+j];


}
```

Rearrange loop nest to move loops over blocks “out” and leave loops over a single block together

Matrix multiplication: sequential code

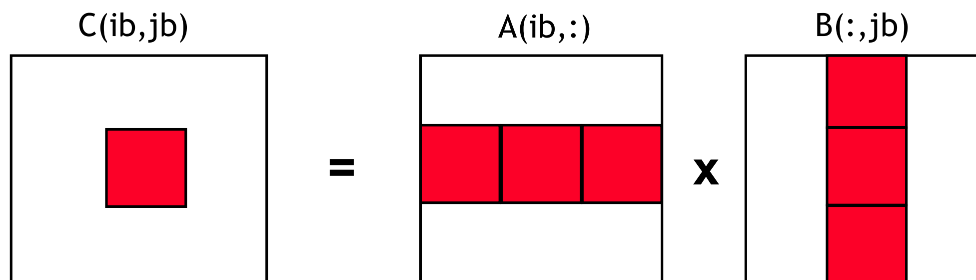
```
void mat_mul(int N, float *A, float *B, float *C)
{ // assume N % block_size = 0
  int i, j, k;
  int NB = N/block_size;
  for (ib = 0; ib < NB; ib++)
    for (jb = 0; jb < NB; jb++)
      for (kb = 0; kb < NB; kb++)
        for (i = ib*NB; i < (ib+1)*NB; i++)
          for (j = jb*NB; j< (jb+1)*NB; j++)
            for (k = kb*NB; k < (kb+1)*NB; k++)
              C[i*N+j] += A[i*N+k]*B[k*N+j];

}
```

This is just a local matrix multiplication of a single block

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{ // assume N % block_size = 0
  int i, j, k;
  int NB = N/block_size;
  for (ib = 0; ib < NB; ib++)
    for (jb = 0; jb < NB; jb++)
      for (kb = 0; kb < NB; kb++)
        sgemm(C, A, B, …)        // C_ib,jb = A_ib,kb * B_kb,jb
}
```



Note: sgemm is the name of the level three BLAS routine to multiply two matrices

Blocked matrix multiply: kernel

```
#define blksz 16
__kernel void mmul(
                const unsigned int N,
                __global float* A,
                __global float* B,
                __global float* C,
                __local  float* Awrk,
                __local  float* Bwrk)
{
   int kloc, Kblk;
   float Ctmp=0.0f;

   //  Compute element C(i,j)
   int i = get_global_id(0);
   int j = get_global_id(1);

   // Element C(i,j) is in block C(Iblk,Jblk)
   int Iblk = get_group_id(0);
   int Jblk = get_group_id(1);

   // C(i,j) is element C(iloc, jloc)
   // of block C(Iblk, Jblk)
   int iloc = get_local_id(0);
   int jloc = get_local_id(1);
   int Num_BLK = N/blksz;

   // Upper-left-corner and inc for A and B
   int Abase = Iblk*N*blksz;    int Ainc  = blksz;
   int Bbase = Jblk*blksz;      int Binc  = blksz*N;

   // C(Iblk,Jblk)=(sum over Kblk) A(Iblk,Kblk)*B(Kblk,Jblk)
   for (Kblk = 0;  Kblk<Num_BLK;  Kblk++)
   {
      // Load A(Iblk,Kblk) and B(Kblk,Jblk).
      // Each work-item loads a single element of the two
      // blocks which are shared with the entire work-group

      Awrk[jloc*blksz+iloc] = A[Abase+jloc*N+iloc];
      Bwrk[jloc*blksz+iloc] = B[Bbase+jloc*N+iloc];

      barrier(CLK_LOCAL_MEM_FENCE);

      #pragma unroll
      for (kloc=0; kloc<blksz; kloc++)
         Ctmp+=Awrk[jloc*blksz+kloc]*Bwrk[kloc*blksz+iloc];

      barrier(CLK_LOCAL_MEM_FENCE);
      Abase += Ainc;      Bbase += Binc;
   }
   C[j*N+i] = Ctmp;
}
```

Blocked matrix multiply: kernel

It's getting the indices right that makes this hard

```
#define blksz 16
__kernel void mmul(
                const unsigned int N,
                __global float* A,
                __global float* B,
                __global float* C,
                __local  float* Awrk,
                __local  float* Bwrk)
{
   int kloc, Kblk;
   float Ctmp=0.0f;

   //  Compute element C(i,j)
   int i = get_global_id(0);
   int j = get_global_id(1);

   // Element C(i,j) is in block C(Iblk,Jblk)
   int Iblk = get_group_id(0);
   int Jblk = get_group_id(1);

   // C(i,j) is element C(iloc, jloc)
   // of block C(Iblk, Jblk)
   int iloc = get_local_id(0);
   int jloc = get_local_id(1);
   int Num_BLK = N/blksz;

   // Upper-left-corner and inc for A and B
   int Abase = Iblk*N*blksz;     int Ainc  = blksz;
   int Bbase = Jblk*blksz;       int Binc  = blksz*N;

   // C(Iblk,Jblk)=(sum over Kblk) A(Iblk,Kblk)*B(Kblk,Jblk)
   for (Kblk = 0;  Kblk<Num_BLK;  Kblk++)
   {
      // Load A(Iblk,Kblk) and B(Kblk,Jblk).
      // Each work-item loads a single element of the two
      // blocks which are shared with the entire work-group

      Awrk[jloc*blksz+iloc] = A[Abase+jloc*N+iloc];
      Bwrk[jloc*blksz+iloc] = B[Bbase+jloc*N+iloc];

      barrier(CLK_LOCAL_MEM_FENCE);

      #pragma unroll
      for (kloc=0; kloc<blksz; kloc++)
         Ctmp+=Awrk[jloc*blksz+kloc]*Bwrk[kloc*blksz+iloc];

      barrier(CLK_LOCAL_MEM_FENCE);
      Abase += Ainc;       Bbase += Binc;
   }
   C[j*N+i] = Ctmp;
}
```

Load A and B blocks, wait for all work-items to finish

Wait for everyone to finish before going to next iteration of Kblk loop.

Blocked matrix multiply: Host

```
#define DEVICE
CL_DEVICE_TYPE_DEFAULT

int main(void)

{  // Declarations (not shown)

  size = N * N; blksz = 16;

  std::vector<float> h_A(size);

  std::vector<float> h_B(size);

  std::vector<float> h_C(size);


  cl::Buffer d_A, d_B, d_C;


  // Initialize matrices and setup

  // the problem (not shown)


  cl::Context context(DEVICE);

  cl::Program program(context,
      util::loadProgram("mmul.cl",
         true));
```

```
cl::KernelFunctor
      <int, cl::Buffer, cl::Buffer, cl::Buffer,
       cl::LocalSpaceArg, cl::LocalSpaceArg >
         mmul(program, "mmul");


d_A   = cl::Buffer(context, begin(h_A), end(h_A),true);

d_B   = cl::Buffer(context, begin(h_B), end(h_B),true);

d_C   = cl::Buffer(context,
            CL_MEM_WRITE_ONLY, sizeof(float) * size);


cl::LocalSpaceArg Awrk =
              cl::Local(sizeof(float) * blksz * blksz);
cl::LocalSpaceArg Bwrk =
              cl::Local(sizeof(float) * blksz * blksz);

mmul(cl::EnqueueArgs( queue,
            cl::NDRange(N,N), cl::NDRange(blksz, blksz)),
         N, d_A,  d_B,  d_C, Awrk, Bwrk);


cl::copy(queue, d_C, begin(h_C), end(h_C));


// Timing and check results (not shown)
}
```

Blocked matrix multiply: Host

```
#define DEVICE
CL_DEVICE_TYPE_DEFAULT

int main(void)

{  // Declarations (not shown)

  size = N * N; blksz = 16;

  std::vector<float> h_A(size);

  std::vector<float> h_B(size);

  std::vector<float> h_C(size);

  cl::Buffer d

  // Initialize matrices and setup

  // the problem (not shown)

  cl::Context context(DEVICE);

  cl::Program program(context,
      util::loadProgram("mmul.cl"
        true
```

Setup local memory with blocks of A and B (16 by 16) that should fit in local memory.

```
cl::KernelFunctor
      <int, cl::Buffer, cl::Buffer, cl::Buffer,
       cl::LocalSpaceArg, cl::LocalSpaceArg >
         mmul(program, "mmul");

d_A   = cl::Buffer(context, begin(h_A), end(h_A),true);

d_B   = cl::Buffer(context, begin(h_B), end(h_B),true);

d_C   = cl::Buffer(context,
             CL_MEM_WRITE_ONLY, sizeof(float) * size);

cl::LocalSpaceArg Awrk =
             cl::Local(sizeof(float) * blksz * blksz);
cl::LocalSpaceArg Bwrk =
             cl::Local(sizeof(float) * blksz * blksz);

mmul(cl::EnqueueArgs( queue,
           cl::NDRange(N,N), cl::NDRange(blksz, blksz)),
         N, d_A,  d_B,  d_C, Awrk, Bwrk);

cl::copy(queue, d_C, begin(h_C), end(h_C));
```

One work-item per element of the C matrix organized into 16 by 16 blocks.

Matrix multiplication performance

- Matrices are stored in global memory.

Case	GFLOP/s	
	CPU	GPU
Sequential C (not OpenCL)	0.85	N/A
C(i,j) per work-item, all global	111.8	70.3
C row per work-item, all global	61.8	9.1
C row per work-item, A row private	9.6	24.9
C row per work-item, A private, B local	12.3	55.4
Block oriented approach using local	138.0	1,801.8
	11.5% of peak	**21.2% of peak**

Device is NVIDIA® Tesla® P100 GPU with 56 compute units, 3,584 PEs
Device is 2x Intel® Xeon® CPU, E5-2695 v4 @ 2.1GHz

Third party names are the property of their owners.

P100 peak is ~8.5 TFLOP/s single precision.
E5-2695 peak is ~1.2 TFLOP/s s.p.

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Matrix multiplication performance

- Block sizes are crucial to performance

Case	GFLOP/s	
	CPU	GPU
Block oriented approach using local, 8x8	67.5	1,511.3
Block oriented approach using local, 16x16	109.6	1,801.8
Block oriented approach using local, 32x32	134.9	1,796.0
Block oriented approach using local, 64x64	138.0	N/A
Vendor SGEMM (MKL / NVIDIA CuBLAS)		5,550.3

Device is NVIDIA® Tesla® P100 GPU with 56 compute units, 3,584 PEs
Device is 2x Intel® Xeon® CPU, E5-2695 v4 @ 2.1GHz

Third party names are the property of their owners.

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.