Lecture 12

Performance Measurement
Advanced Communication

Announcements

- No class on 11/17 and 11/19
- Makeup lectures
  - Friday 11/20. 3:00 to 4:20
  - Weds 12/2 5-7PM
- CSE 260 Symposium
  - Week 10: Tues, Weds, Thurs

Parallel print function

Parallel print function

- Debugging output can be hard to sort out on the screen
- Many messages say the same thing

```
Process 0 is alive!
Process 1 is alive!
        ...
Process 15 is alive!
```

- Compare with

```
Processes[0-15] are  alive!
```

- Parallel print facility

  http://www.llnl.gov/CASC/ppf

Summary of capabilities

- Compact format list sets of nodes with common output

```
PPF_Print( MPI_COMM_WORLD, "Hello world" );
0-3: Hello world
```

- %N specifier generates process ID information

```
PPF_Print( MPI_COMM_WORLD, "Message from %N\n" );
Message from 0-3
```

- Lists of nodes

```
PPF_Print(MPI_COMM_WORLD,
          (myrank % 2)
           ? "[%N] Hello from the odd numbered nodes!\n"
           : "[%N] Hello from the even numbered nodes!\n")
```

```
[0,2] Hello from the even numbered nodes!
[1,3] Hello from the odd numbered nodes!
```

Practical matters

- Installed in `$(PUB)/lib/PPF`
- Use a special version of the arch file called

  `arch.ppf.pgi`
- Each module that uses the facility must

```
#include "ptools_ppf.h“
```

- Look in `$(PUB)/Examples/PPF` for example programs `ppfexample_cpp.C` and `test_print.c`

Send_Recv

- Instead of Send and Recv, we can use MPI_Sendrecv_replace( ) to simplify the coding and improve performance
- Sends then receives a message using a single buffer

```
int MPI_Sendrecv_replace
    ( void *buf,
     int count,  MPI_Datatype
datatype,
     int dest,   int sendtag,
     int source, int recvtag,
     MPI_Comm comm, MPI_Status
*status )
```

Parallel Matrix Multiplication

Parallel matrix multiplication

- Assume p is a perfect square
- Each processor gets an $n/\sqrt{p} \times n/\sqrt{p}$ chunk of data
- Organize processors into rows and columns
- Process rank is an ordered pair of integers
- Assume that we have an efficient serial matrix multiply

p(0,0)	p(0,1)	p(0,2)
p(1,0)	p(1,1)	p(1,2)
p(2,0)	p(2,1)	p(2,2)

Loop reordering

- The simplest formulation of matrix multiply is the so called "ijk" formulation, named after the order of the loops

```
for i:= 0 to n-1,  j:= 0 to n-1, k:= 0 to n-1
        C[i,j] += A[i,k] * B[k,j]
```

- Now consider the "kij" formulation

```
for k:= 0 to n-1,  i:= 0 to n-1, j:= 0 to n-1
        C[i,j] += A[i,k] * B[k,j]
```

Formulation

- The matrices may be non-square

```
for k := 0 to n3-1
    for i := 0 to n1-1
        for j := 0 to n2-1
            C[i,j] += A[i,k] * B[k,j]
```

```
C[i,:] += A[i,k] * B[k,:]
```

- The two innermost loop nests compute

n3 *outer products*

```
for k := 0 to n3-1
        C[:,:] +=  A[:,k] • B[k,:]
```

where • is outer product

Outer product

- Recall that when we multiply
  an m×n matrix by an n × p matrix…
  we get an m × p matrix
- Outer product of *column vector* a^T and *vector* b = matrix C
  an m × 1 times a 1 × n

  a[1,3] • x[3,1]

$$(a,b,c) * (x,y,z)^T \equiv \begin{pmatrix} ax & ay & az \\ bx & by & bz \\ cx & cy & cz \end{pmatrix}$$

Multiplication table with rows formed by a[:] and the columns by b[:]

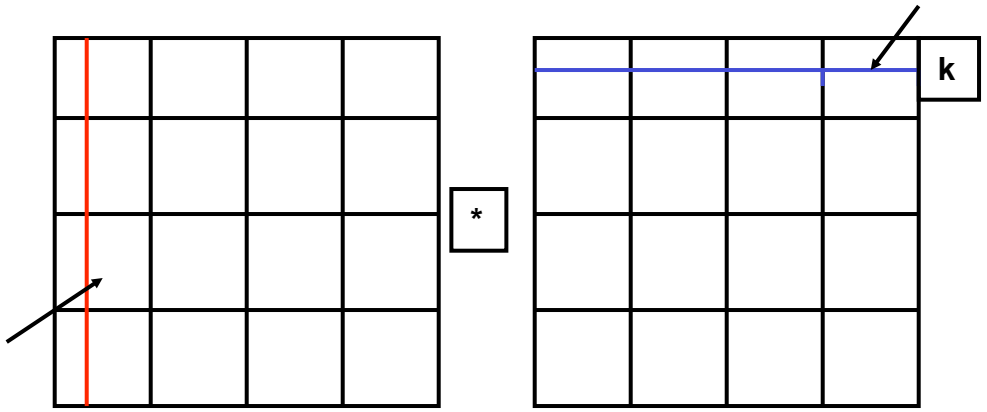
- The SUMMA algorithm computes n partial outer products:

```
for k := 0 to n-1
    C[:,:] +=  A[:,k] • B[k,:]
```

Serial algorithm

- Each row k of B contributes to the n partial n partial outer products :

```
for k := 0 to n-1
    C[:,:] +=  A[:,k] • B[k,:]
```

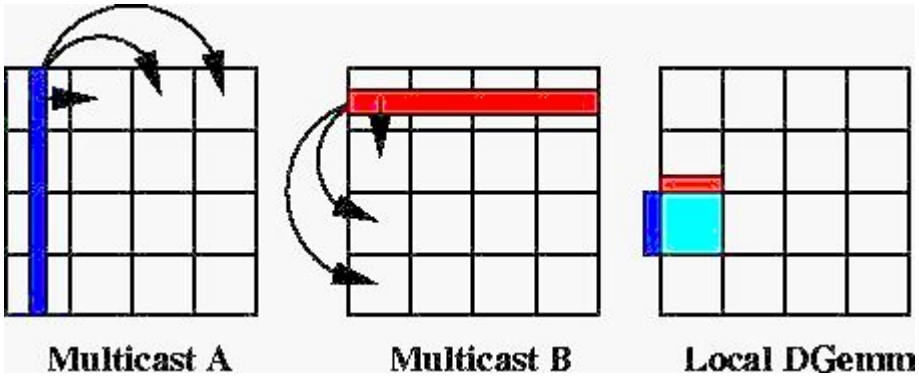


Parallel algorithm

- Set up a processor geometry P = px × py
- Blocked multiply, panel size = b << N/max(px,py)

```
for k := 0 to n-1 by b
    multicast A[ : , k:k+b-1 ]  along processor rows
    multicast B[ k:k+b-1, :  ]  along processor columns
    C +=  A[:,k:k+b-1 ] * B[k:k+b-1,: ]  // Local MM
```

- Each row and column of processors independently participate in a broadcast of a panel
- Owner of the panel changes with k



What is the performance?

```
for k := 0 to n-1 by b
// Tree broadcast:  lg √p (α + bβn/√p)
//  For long messages:  2((√p -1)/ p)bβn
    multicast A[ :          , k:k+b-1 ] along rows
    multicast B[ k:k+b-1, :   ]         along columns
// Built in matrix multiply: 2(n/√p)^2 b
    C += A[:,k:k+b-1 ] * B[k:k+b-1,: ]
```

- Total running time: $\sim 2n^3/p + 4\beta bn/\sqrt{p}$

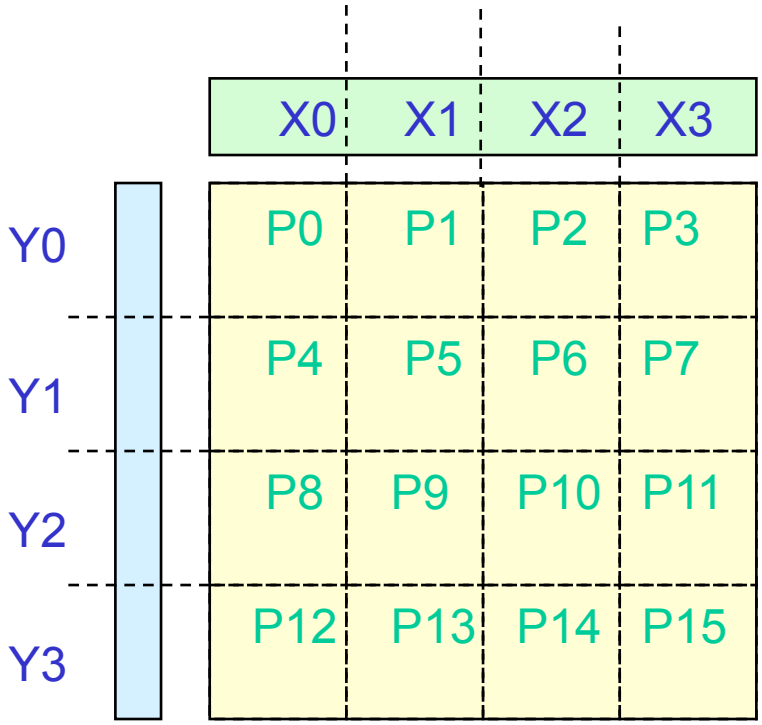
Highlights of SUMMA

- **Performance**
  - Running time = $2n^3/p + 4\beta bn/\sqrt{p}$
  - Efficiency = $O(1/(1 + \sqrt{p}/n^2))$
- **Generality:** non-square matrices, non-square geometries
- Adjust b to **tradeoff latency cost** against **memory**
  - b small ⇒ less memory, lower efficiency
  - b large ⇒ more memory, high efficiency
- Low **temporary storage**
  - grows like $2bn/\sqrt{p}$
- A variant used in SCALAPACK

  R. van de Geign and J. Watts,
  "SUMMA: Scalable universal matrix multiplication algorithm,"
  Concurrency: Practice and Experience, **9:**255-74 (1997) www.netlib.org/lapack/lawns/lawn96.ps

Communication domains

- Summa motivates MPI ***communication domains***
- Derive communicators that naturally reflect the communication structure along rows and columns of the processor geometry



Communication domains

- A communicator is name space specified by an MPI communicator
- Messages remain within their domain
- Communication domains simplify the code, by specifying subsets of processes that may communicate
- A processor may be a member of more than one communication domain

Splitting communicators

- Each process computes a key based on its rank
- Derived communicators group processes together that have the same key
- Each process has a rank relative to the new communicator
- If a process is a member of several communicators, it will have a rank within each one

Splitting communicators for Summa

- Create a communicator for each row and column
- Group the processors by row

  key = myid div √P
- Thus, if P=4
  - Processes 0, 1, 2, 3 are in one communicator because they share the same value of key (0)
  - Processes 4, 5, 6, 7 are in another (1), and so on

MPI support

- **MPI_Comm_split( )** is the workhorse

```
MPI_Comm_split(MPI_Comm comm,
                        int splitKey,
                             int rankKey,
                             MPI_Comm*
newComm);
```

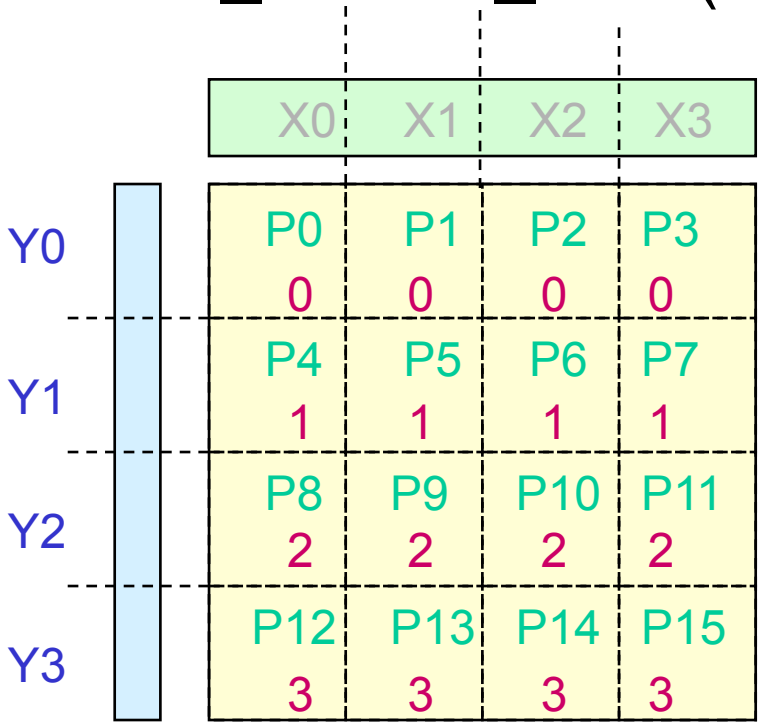
- A collective call
- Each process receives a new communicator, `newComm`, which it shares in common with other processes having the same `splitKey` value

Establishing row communicators

MPI_Comm rowComm;

MPI_Comm_split(MPI_COMM_WORLD, myRank / √P, myRank, &rowComm);

MPI_Comm_rank(rowComm,&myRow);



- Ranks are unique within a communicator
- Ordered according to `rankkey` = myRank / √P

More on Comm_split

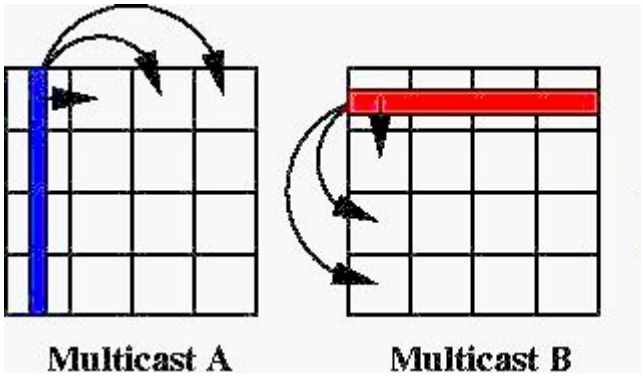
```
MPI_Comm_split(MPI_Comm comm,
               int splitKey,
               int rankKey,
               MPI_Comm* newComm);
```

- Ranks are assigned arbitrarily among processes sharing the same `rankKey` value
- May exclude a process by passing the constant `MPI_UNDEFINED` as the `splitKey`
- A special `MPI_COMM_NULL` communicator will be returned
- Alternative is to enumerate all nodes in a set and call `MPI_Group_incl()` and `MPI_Comm_create()`

Panel Broadcast

- Each row/column calls Bcast, a multicast
- Contributing row/column circulates across and down ward

```
Foreach step in 0 to n by panel
   Ring Bcast(current column, comm_col)
   Ring Bcast(current row,    comm_row)
  DGEMM ( )
```



Ring BCast

```
RING_Bcast( double *buf, int count,
                    MPI_Datatype type, int root,
                    MPI_Comm comm )
MPI_Comm_rank( comm, &rank );
MPI_Comm_size( comm, &np );
if ( rank ≠ root)
   MPI_Recv( buf, count, type, (rank-1+np) mod np,
                    MPI_ANY_TAG, comm, &status );
if ( ( rank +1 ) mod np ≠ root )
   MPI_Send(buf, count, type, (rank+1)%np, 0, comm );
```

Occupancy Calculator

1.) Select Compute Capability (click):	1.3

2.) Enter your resource usage:	
Threads Per Block	256
Registers Per Thread	8
Shared Memory Per Block (bytes)	2048

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	**1024**
Active Warps per Multiprocessor	**32**
Active Thread Blocks per Multiprocessor	**4**
Occupancy of each Multiprocessor	**100%**

Physical Limits for GPU:	**1.3**
Threads / Warp	32
Warps / Multiprocessor	32
Threads / Multiprocessor	1024
Thread Blocks / Multiprocessor	8
Total # of 32-bit registers / Multiprocessor	16384
Register allocation unit size	512
Shared Memory / Multiprocessor (bytes)	16384
Warp allocation granularity (for register allocation)	2

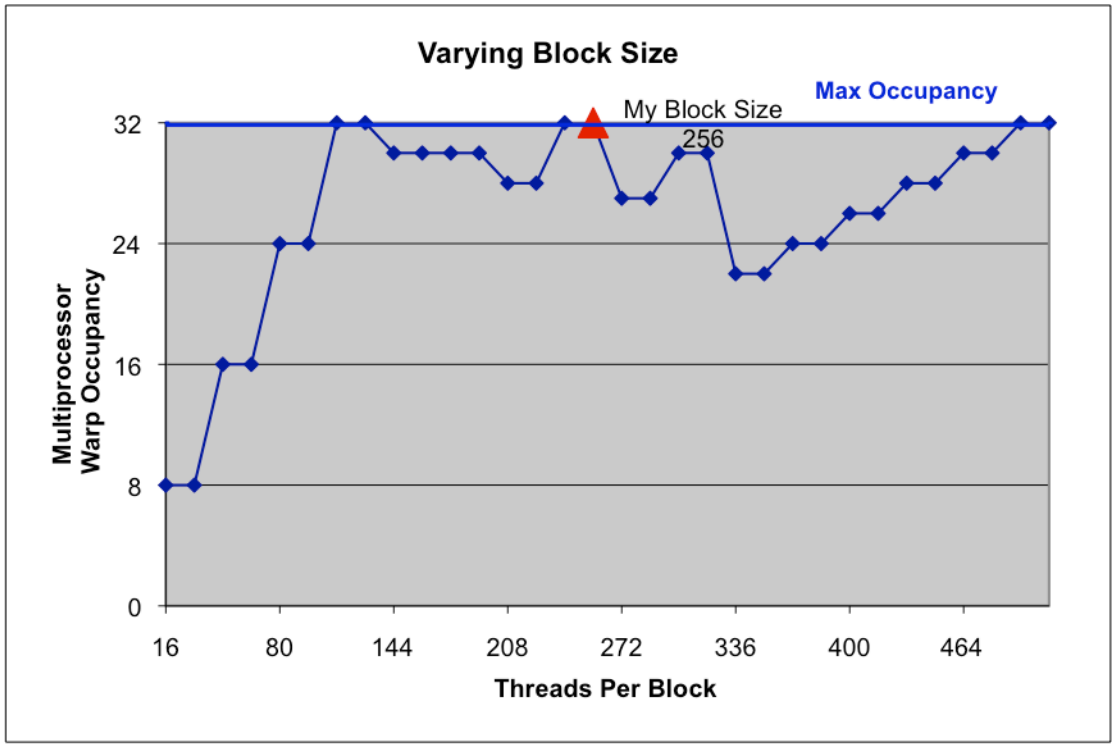
Allocation Per Thread Block	
Warps	8
Registers	2048
Shared Memory	2048

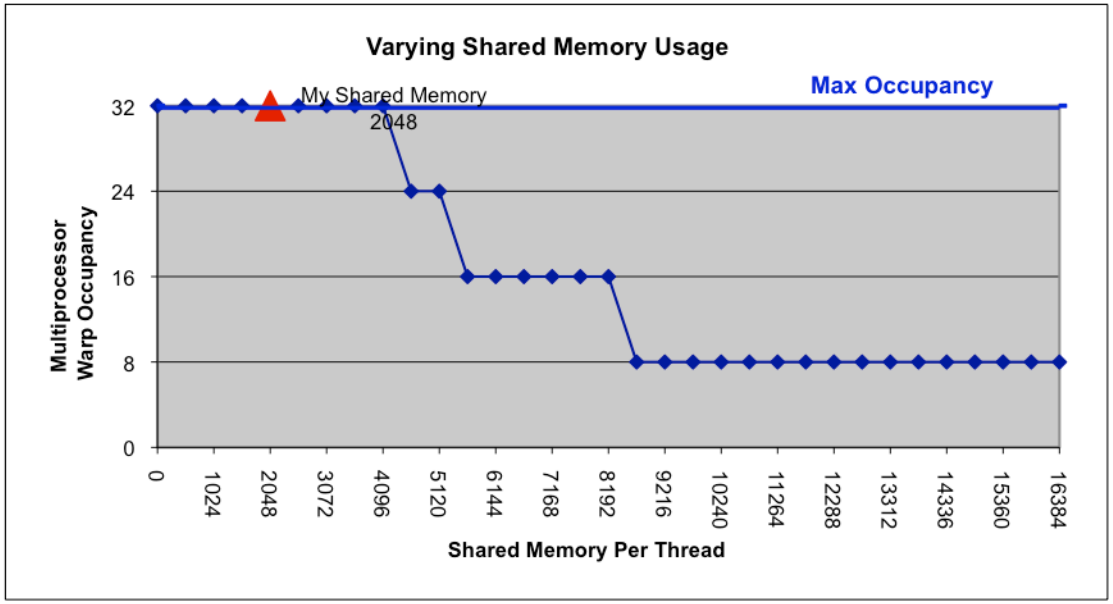
These data are used in computing the occupancy data in blue

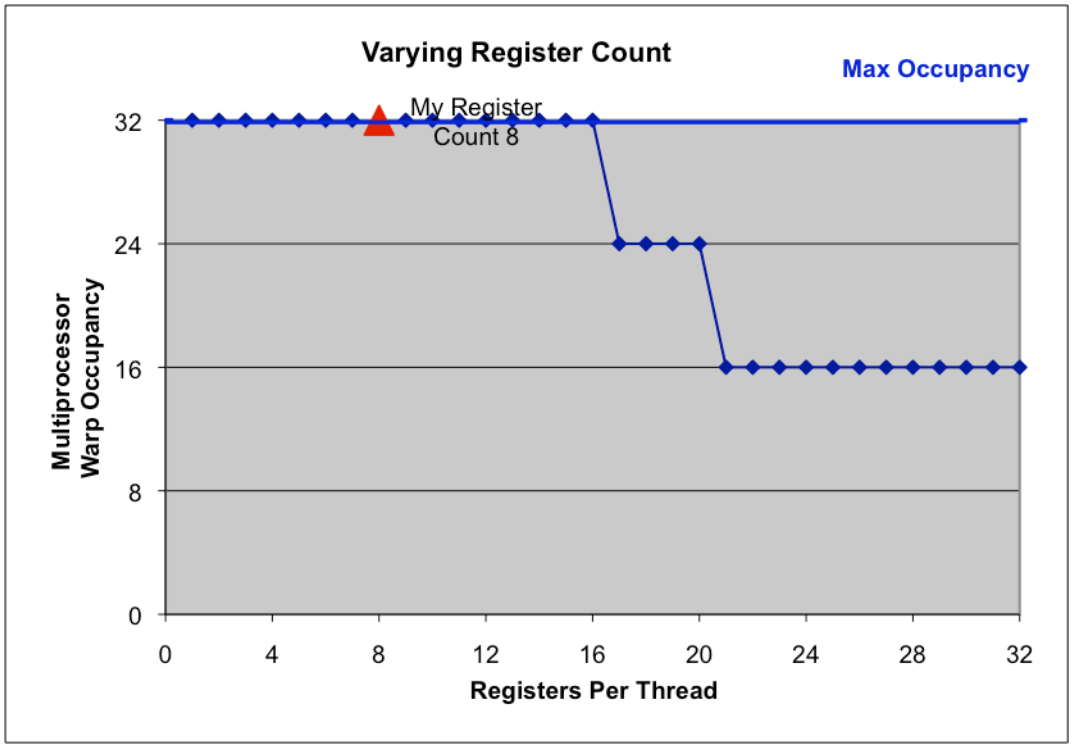
Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Multiprocessor	4
Limited by Registers / Multiprocessor	8
Limited by Shared Memory / Multiprocessor	8

Thread Block Limit Per Multiprocessor highlighted **RED**

CUDA Occupancy Calculator	
Version:	1.5


Varying Block Size
Max Occupancy
My Block Size
256
32
24
16
8
0
Multiprocessor
Warp Occupancy
16
80
144
208
272
336
400
464
Threads Per Block


Varying Shared Memory Usage
Max Occupancy
My Shared Memory
2048
32
24
16
8
0
Multiprocessor
Warp Occupancy
0
1024
2048
3072
4096
5120
6144
7168
8192
9216
10240
11264
12288
13312
14336
15360
16384
Shared Memory Per Thread


Varying Register Count
Max Occupancy
My Register
Count 8
32
24
16
8
0
Multiprocessor
Warp Occupancy
0
4
8
12
16
20
24
28
32
Registers Per Thread

Threads	Warps/Multiprocessor
256	**32**
16	8
32	8
48	16
64	16
80	24
96	24
112	32
128	32
144	30
160	30
176	30
192	30
208	28
224	28
240	32
256	32
272	27
288	27
304	30
320	30
336	22
352	22
368	24
384	24
400	26
416	26
432	28
448	28
464	30
480	30
496	32
512	32

Registers	Warps/Multiprocessor
8	32
1	32
2	32
3	32
4	32
5	32
6	32
7	32
8	32
9	32
10	32
11	32
12	32
13	32
14	32
15	32
16	32
17	24
18	24
19	24
20	24
21	16
22	16
23	16
24	16
25	16
26	16
27	16
28	16
29	16
30	16
31	16
32	16

Shared Mei	Warps/Multiprocessor
2048	32
0	32
512	32
1024	32
1536	32
2048	32
2560	32
3072	32
3584	32
4096	32
4608	24
5120	24
5632	16
6144	16
6656	16
7168	16
7680	16
8192	16
8704	8
9216	8
9728	8
10240	8
10752	8
11264	8
11776	8
12288	8
12800	8
13312	8
13824	8
14336	8
14848	8
15360	8
15872	8
16384	8