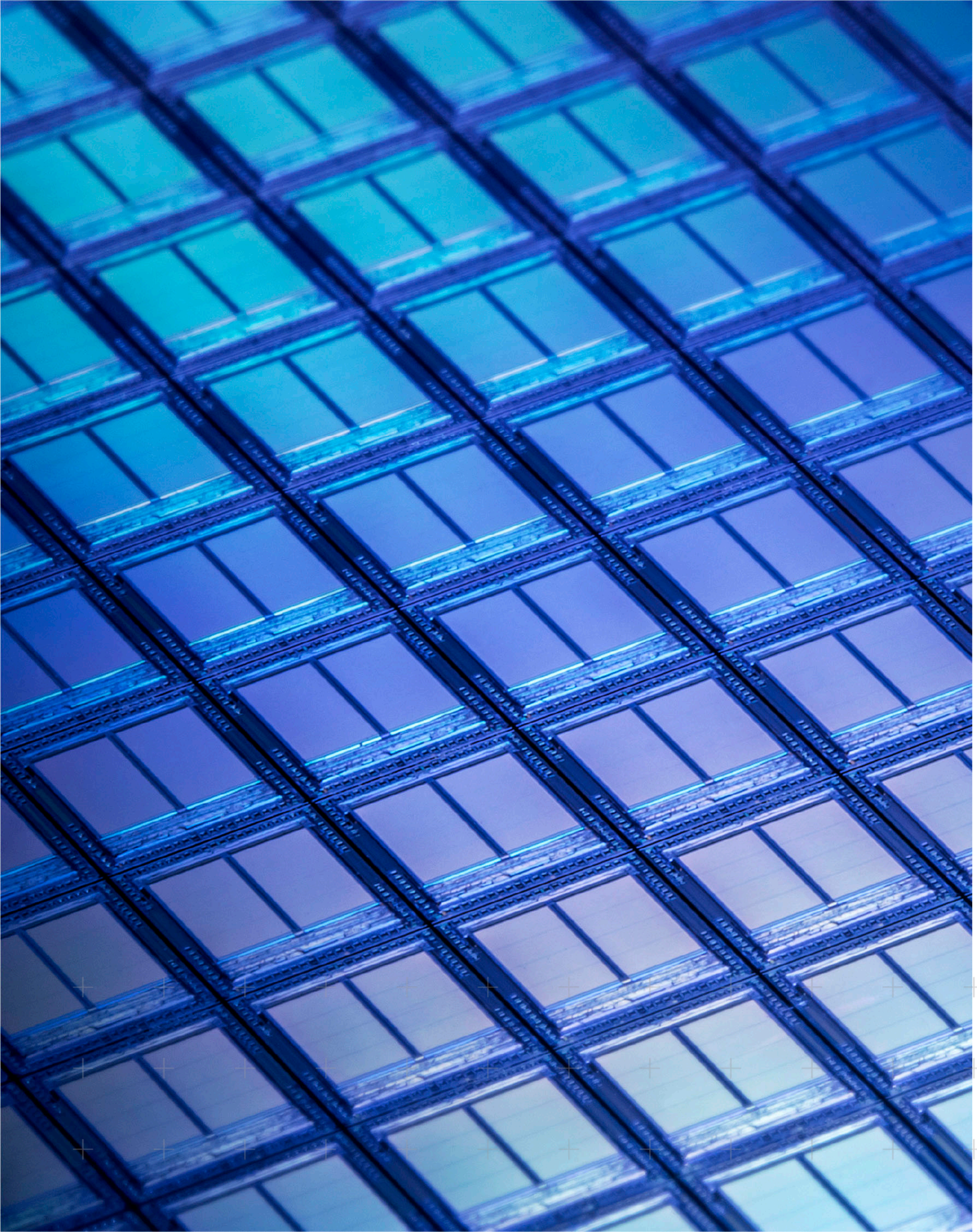


Minimizing Copy Overhead While Sharing GPUs in a Box

Mark Roulo
Sep 2021

IITM HPC workshop

Presentation Overview

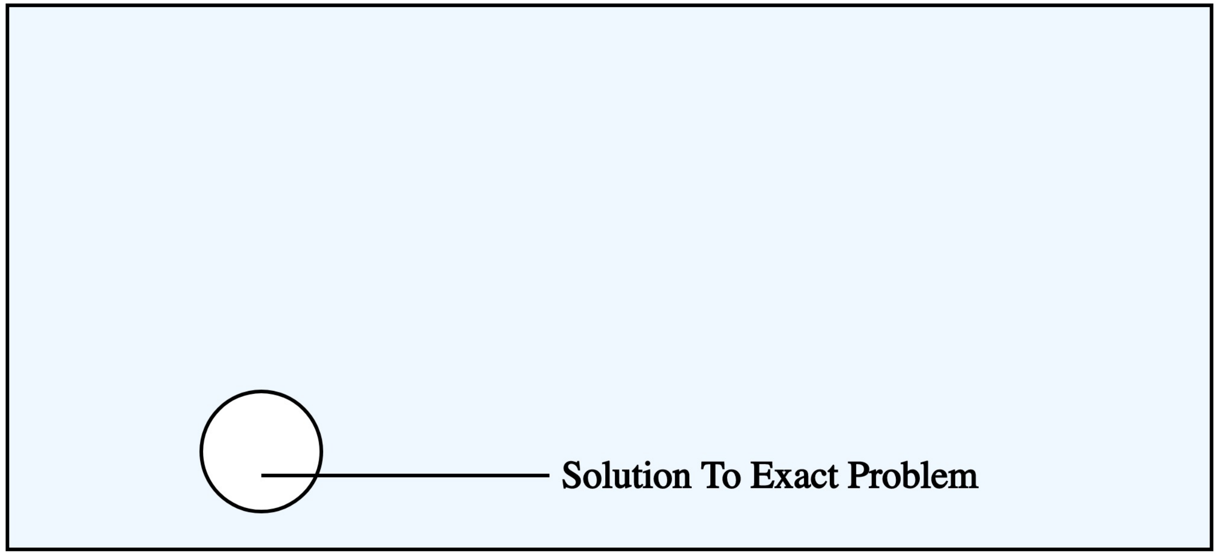
- What Problem Are We Trying To Solve?
- Design Philosophy
- Technologies Behind the Solution
- Putting the Pieces Together

The Problem

- We have 32+ CPU processes that each want access to GPU/DL resources
- DL "sessions" (e.g. TensorFlow) can be used by only one process
- We don't get very many sessions per GPU
  - DL sessions can require a lot of GPU memory (eg 8+ GB/session)
  - So maybe 2-3 sessions per GPU
  - This is a lot less than 32+
- Standard "sharing" technologies use network copying + marshalling
- We don't want to take the hit from copying+marshalling

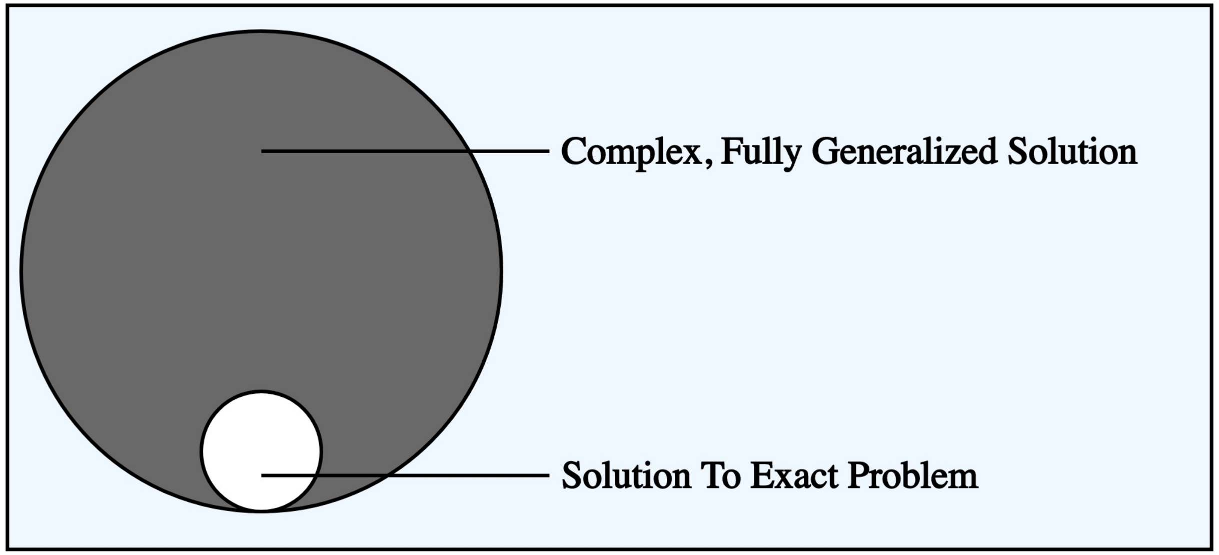
Understand that all of our problems go away if we are willing to run slower!!!!

Design Philosophy (1)



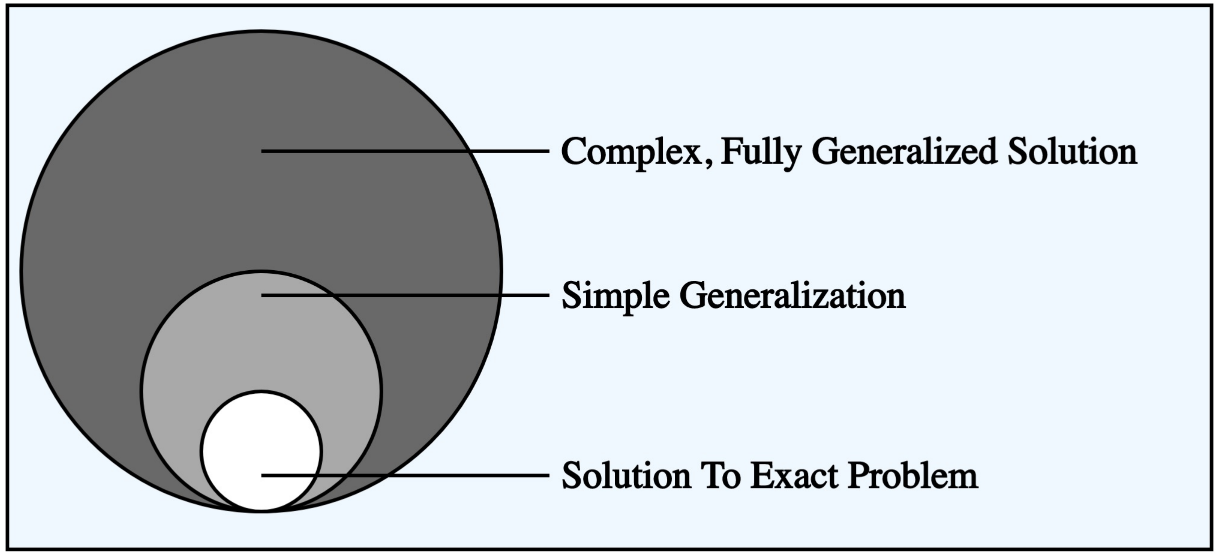
- Solve only today's problem. Not good. You don't want to revisit this in a year.

Design Philosophy (2)


Complex, Fully Generalized Solution
Solution To Exact Problem

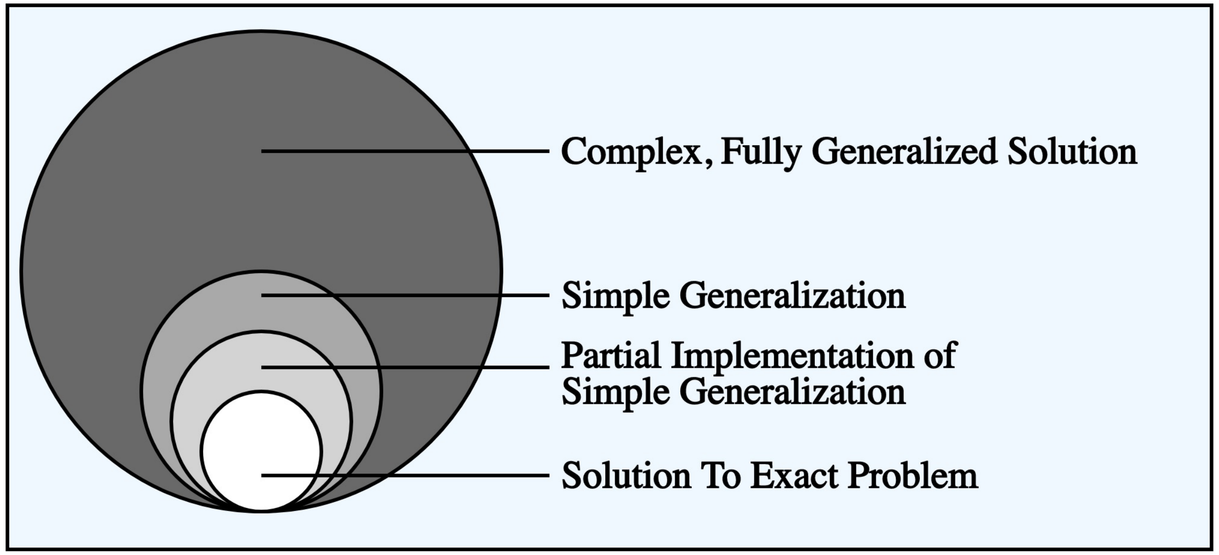
- Solve only today's problem. Not good. You don't want to revisit this in a year.
- Design and build the fully generalized solution. Not good. Lots of wasted effort. You are not Facebook or Google.

Design Philosophy (3)


Complex, Fully Generalized Solution
Simple Generalization
Solution To Exact Problem

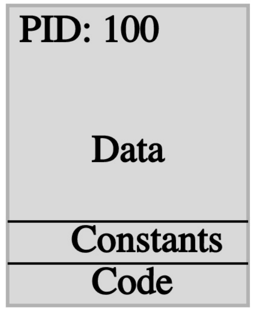
- Solve only today's problem. Not good. You don't want to revisit this in a year.
- Design and build the fully generalized solution. Not good. Lots of wasted effort. You are not Facebook or Google.
- Design up to the "natural", simple generalization to your problem.
  Don't implement all of that generalization.

Design Philosophy (4)

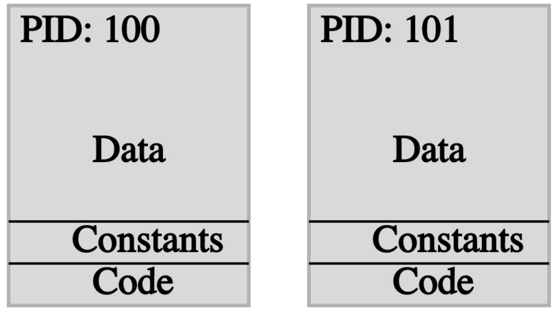

Complex, Fully Generalized Solution
Simple Generalization
Partial Implementation of Simple Generalization
Solution To Exact Problem

- Solve only today's problem. Not good. You don't want to revisit this in a year.
- Design and build the fully generalized solution. Not good. Lots of wasted effort. You are not Facebook or Google.
- Design up to the "natural", simple generalization to your problem.
  Don't implement all of that generalization.
- Implement your best guess of the most likely to be needed bits of the easy generalization.

A Linux Process



Linux Processes After Forking

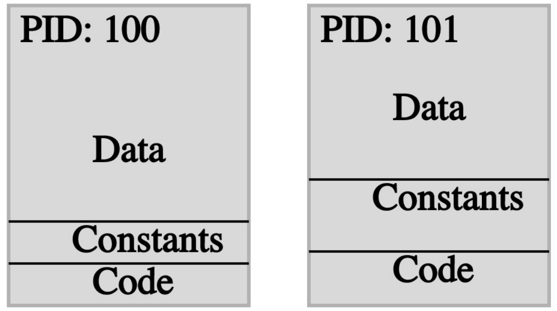


Right after fork() the code, constant values and mutable data values are (almost) identical.

This commonality includes pointers to values in the process (so *p will have the same address in both processes).

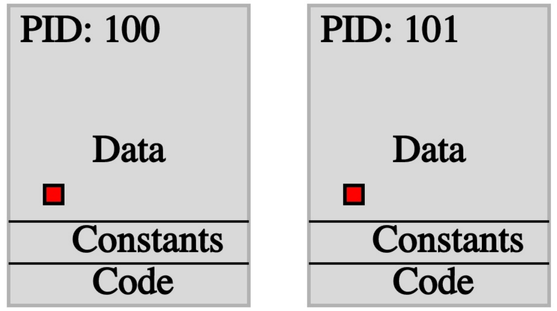
But each process has a separate copy, so changes will be private to the process making the change.

Linux Processes After Fork and Exec



The code, constant values and mutable data are all different now!

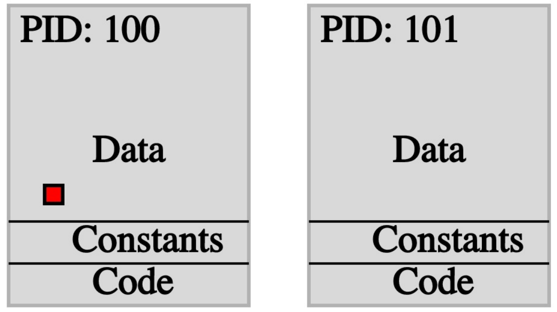
Private Shared Memory (1)



This is the same as forking (common values, but copies so changes are local)

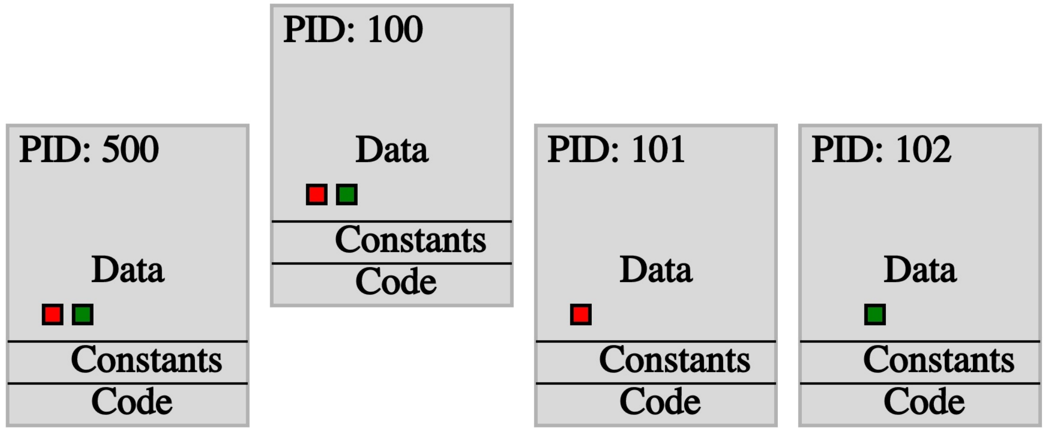
Except that the private shared memory is SHARED. Changes made by one process are visible to the other process.

Private Shared Memory (2)



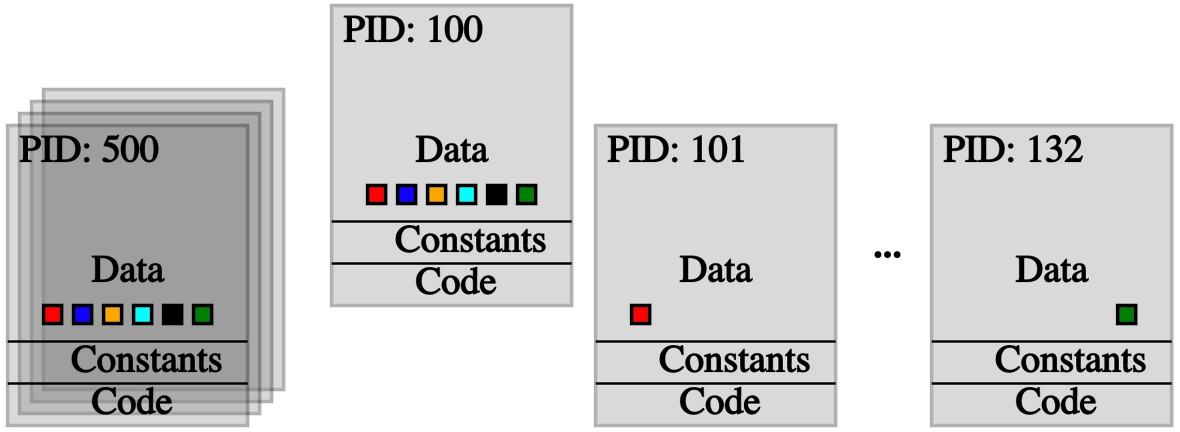
Private shared memory can be closed/freed. Then the process no longer has access to the shared memory.

Private Shared Memory (3)



We can arrange for each forked (child) process to have access to only one shared memory block. The parent can acccess all shared memory blocks.

Private Shared Memory (4)



- Each child process has access to one (and only one) shared memory block to share data with helper processes.
- Each helper process has access to a DL/GPU session.
- Each helper process can access any of the child shared memory blocks.

Transferring Control

- Child processes send a message to the parent, which dispatches to <u>any</u> available helper.
- Messages are queued up if no helper is available.

The messaging API looks like this:

-
  ```
  typedef struct shm_msg_t
  {
      shm_func_ptr_t  func_ptr;
      void            *params;
  } shm_msg_t;
  ```
- `void shm_send_msg(shm_msg_t msg);`
- `shm_msg_t shm_read_msg();`

Transferring Control (2)

An example function might look like this:

```
void helper_func_1(void *data)
{
    printf("   Helper %2d [pid:%8d]: %s\n",
           shm_get_helper_id(),
           getpid(),
           (char*)data));
}
```

Transferring Control (3)

- Messages are small (16 bytes; 64 bytes after adding some overhead)
- Messages only sent within a single OS domain.
- We use Posix pipes to send the messages.
- A single thread in the parent process manages all the message dispatch.
  - Single-threaded dispatch, so a simple loop with no locking.
  - Use Linux `poll()` API to avoid busy waiting
  - ~150,000 round-trip messages/sec on my MacBook Pro laptoo.
  - Our inference graphs take about 4 - 20 ms to run

(*) poll() is a poorly named API. It is an improved version of select() and does not busy wait!

Managing the Shared Memory

- `void *shm_malloc(size_t size);`
- `void shm_free(void*);`

This is only called from the child processes.
The helpers can access the shared memory, but do not allocate out of it.

Putting It All Together

- Someone launches the parent process.
- The parent sets up the shared memory regions.
  The parent sets up the process-to-process pipes.

```
int shm_launch( int child_cnt,
                shm_main_bundle_t main_bundles[],
                int shm_block_size,
                int helper_cnt);
```

- The parent forks the helpers.
- The parent forks the children (and closes the correct shared memory regions).
- Children manage the memory

```
void *shm_malloc(size_t size);
void shm_free(void*);
```

- Children and helpers run, sending messages back and forth.

```
void shm_send_msg(shm_msg_t msg);
shm_msg_t shm_read_msg();
```

Overview of Some Design Decisions

- Why Helper Processes?
- Why Single Dispatch Thread in Parent?
- Why Allocate Only in Child?
- Why Homogeneous Helper Processes?
- Constraint: Approach Doesn't Work on Windows
- Constraint: One Message Outstanding At a Time Per Child

Thank you